Table of Contents:

1. General Software Construction (Theory)........ 3
2. The Shell........ 3
- How it works? High Level........ 3
- Brief Overview of OS Kernel........ 4
- Shell Control Characters & Important Commands........ 4
- Signal Flavors........ 5
- POSIX (Portable Operating System Interface) model........ 5
- Directories........ 6
- Inode Number........ 6
- Hard Links........ 6
- Symbolic Links (Symlinks)........ 7
- Expansion Order in Shell Commands........ 8
- Quoting in the Shell........ 8
- Useful Tips........ 9
- Practice Shell Scripts........ 9

3. Emacs (Editor Macros)........ 12
- Overview........ 12
- Important Commands........ 12
  - Buffer/Window Commands........ 12
  - Shell Commands Within Emacs........ 12
  - Special Character Commands........ 12
  - Help Commands........ 12
  - Searching........ 13
- Emacs Buffers........ 13
- Emacs Modes........ 13
- Emacs Regular Expressions........ 14

4. Emacs Lisp........ 15
- Overview........ 15
- 3 Ways to Deal w/ Scripting Langauges........ 15
- Elisp Objects........ 15
- Comparison in Elisp........ 16
- Practice Elisp Scripts........ 16

5. Python........ 18
- Sequences........ 18
  - Operations on Sequences........ 18
  - Operations on Mutable Sequences........ 19
  - Operations on Lists........ 19
  - Operations on Strings........ 19
- Dictionaries........ 19
  - Operations on Dictionaries........ 20
- Callables........ 20
  - Functions........ 20
  - Classes........ 20
- Dunder Methods in Classes........ 20
- Modules........ 21
- Packages........ 22
  - pip........ 22
  - Version numbers........ 22
  - Mixed Python........ 22
- Practice Python Scripts........ 23

6. Network Software Construction........ 25
- Web Application Types........ 25
- Problems with Client Server Applications........ 26

○ Performance Issues (Latency & Throughput)....26
○ Correctness Issues....27
● Network Configuration Types....27
○ Circuit Switching....27
○ Packet Switching....27
● Packets....28
● Protocols (Overview)....28
● Protocol Layers....29
○ Internet Layer....29
○ Transport Layer....30
○ Application Layer....30
● Hypertext Transfer Protocol (HTTP)....31
○ HTTP (v1)....31
● HTTP 2....32
● HTTP 3....32
● SGML (Standard Generalized Markup Language)....33
● HTML (HypterText Markup Language)....34
● JavaScript....34
○ JXS....34
● Browsers (Browser Rendering Pipeline)....34
7. Version Control....35
● Backups....35
● Features of Version Control System....36
● Intro to Git....36
● Distributed Version Control System (DVCS)....37
● Branches....38
● Tags....38
● Merging....38
● Rebasing....38
● Git Internals....39
● Git Objects....40
● Hashing....41
● Compression....42
○ Huffman Coding....42
○ Adaptive Compression w/ Huffman Coding....42
○ Dictionary Compression....43
○ Adaptive Dictionary Compression....43
8. C Programming & Debugging....43
● C vs. C++....44
● Compilation & Execution Environments....44
● How to Debug?....45
4. Defensive Programming....46
● GDB....47
○ Intro to GDB....47
○ Get Started....48
○ GDB Commands....48
○ Extending GDB....49
○ Remote Debugging....49
○ Breakpoints....49
○ Reverse Execution....50
● Performance Optimization....50
● Dynamic (Runtime) Analysis....51
● Static (Compile-time) Checking....52
9. Software Security....52

- Network Attacks....53
- Cross Site Scripting (XSS)....53
- Prototype Pollution....53
- Testing Security....53
  - Risk Assessment....54
  - Trusted Computing Base....54

1. General Software Construction (Theory)

- things that you do not consider in CS-31, but is still necessary for software to work
  - data design
  - performance/efficiency
  - reliability → persistence (persistent data survives reboots)
    - you need programs to work even when parts, or underlying hardware, doesn't work
    - in the RW, reliability is often taken for granted; customer assumes that your code is reliable → up to us to think about things that could go wrong and write reliable code that functions properly even when hardware or other software component doesn't work
  - security & privacy
    - your customer won't know about what could go wrong; this will be our job
  - dependencies/configuration → is the underlying software configured/set up properly? what does your software assume as part of its functioning?
  - understandability/legibility → determines whether its difficult or not to maintain?
- 2 requirements for getting up to speed
  - Core — How to think when scripting? How code runs? Fundamental execution model for the language
    - Thinking logic in other programming languages and then converting to the chosen language does not work, or at least is much less efficient
    - Figure out how the core developers of the language think
  - Ecosystem — How developers glue together pieces of code into a functioning application?

2. The Shell

- How it works? High Level
  - Operating System (OS) runs on hardware (x86 for SeasNet); apps run on top of OS
    - Many OS get extremely long (millions of lines of source code); leads to understanding issues and performance issues
  - We want to break up OS into smaller pieces each independently understandable and maintainable
  - Shell →
    - Idea is to strip down the operating system to as small as we can make it → OS kernel
    - around OS kernel is very thin layer (also talks to the hardware); lets you type commands at it and spits out the result → figures out what kernel API component you want to use, does it, and outputs the result
    - thin layer (shell) → **sh.c**
      - very simple program – reads commands and implements them using low level system calls
      - supposed to be an interface that you can use by typing into it, and seeing output on your screen (interactive); doesn't do much on its own → relies on OS kernel

```
while (true)
{
        c = read_cmd ('stdin');
        if (c == "exit") break;
        { issue low level kernel calls to do the cmd; }
}
```

    - there is a low level system call that the kernel knows abt that **tells it that you want to run another program**

```
        execlp("/usr/bin/sort", "sort", 0);
```

        - `$ sort` - type this in the shell, the shell will execute the above low-level system call, it'll drop into the kernel, and the kernel will arrange for the sort program to run
        - shell is a meta program; **a program for starting/setting up other programs** – `/bin/sh` → part of the OS, implements the shell

- Brief Overview of OS Kernel
    - OS kernel (CS 111 material) rests on top of the hardware
    - Kernel starts up when you boot the system; always running → don't often notice it
    - Kernel is boss (master control program); can kill off shell, running programs, etc.; the kernel typically doesn't want to kill off the shell, but it can
        - Kernel can have a bug that causes it to kill every process in the system?? What would this look like?? This happened to Eggert once.

```
true.c
int main() {} //equivalent to ... return 0; ... exit (0);
```

    - simplest c program; someone needs to tell the kernel that we want this program to run → `execlp(..., "simple", 0);`
    - Once the program hits the return/dies, it informs the kernel → **point being that even the simplest programs must interact with the kernel in some capacity**
    - Applications and the shell itself actually execute machine/hardware instructions (most of the time); sometimes must communicate with the kernel (which can do things that machine/hardware does not know how to do, e.g. reading from stdin)
        - The shell is the same way; shell interacts directly with the hardware to figure out which system call to make for example

- Shell Control Characters & Important Commands
    - `$ ^D` → whatever program is reading from the terminal will get an end of file; if typing into the shell (which is reading from the terminal), then the shell will log out; useful when a program reads from standard input
    - `$ ^C` → kills everything that is currently running
    - `$ ^Z` → **suspends** whatever program is currently running and moves it to the background; shell will take over → you can run whatever command you want after
        - `$ fg` → brings to the foreground; with no argument, will bring back the most recently suspended program), `$ fg %integer_val`
        - `$ jobs` → tells what all jobs are in the background
        - Once you have some tabbing functionality, the need for suspension is far less; can be cases where you don't, or ssh is not connecting, where this is needed
    - `$ man {cmd}` → **will give the documentation for a command)!!!! is very important for this course, which is about finding out about how stuff works**
        - `$ info {cmd}` → will drop you into even longer winded documentation. **How do you find out about more commands, when you don't know the name of the command?**
        - When the documentation is insufficient, **read the source code** for the different functions (e.g. kill.c)
        - Most commands have a `--help` option → will print out a **brief** summary of how you can invoke the command (to remind you of options); `man` will provide more details
    - `$ ps` → process status; lists all the running programs/processes that you're interested in, as well as their process id)
        - process vs. program → program is static file, process is an actually running program
        - **He mentioned reading up the options for ps (lot of them); determined by user preference**
    - `$ kill {PID/process id}` → sends a signal to the process to quit/die; like ^C, even if the process is not attached to the terminal, etc.
        - Some processes can ignore this command? Most do not, and die on command

        - `$ kill -KILL {PID}` or `$ kill -9 {PID}` → will kill the process **no matter what**; processes cannot ignore – very dangerous, because many programs have clean-up that they have to perform (that they typically do on their own when killed normally)
    - `$ ls` → lists the files in the current directory; without any options, just lists the names of the file without any more details, one file per line
        - `$ ls -l` → lists all files AND DETAILS (includes meta information about the file) – file type (directory or regular file), permissions, hard links, user, group, number of bytes, time last modified, and file name
        - `$ ls -i` → outputs the inode number, # that uniquely identifies the file in the file system
        - `$ ls -d` → if I input name of a directory, don't tell me all the names in the directory, just tell me about the directory itself
        - `$ ls -a` → prints all files in the directory, even hidden files and hidden directory
            - By convention, if you just type ls, it would list file names starting with . and ..; there may be quite a few of these
            - `$ echo *` → will match to all files in the current directory except the ones starting with .; if you wanted to print all files, you'd need `$ echo .* *`
    - `$ cd` → changes the working directory
        - `$ cd /` → the working directory is the root of the entire file system rather than where I am
        - `$ cd -` → goes back ONE traversal to where I was
    - `$ ln` modifies . → creates a file named <file2> in the working directory; you may think this is a command to create file2, but it actually modifies the working directory
    - `$ mv` (which renames files) → works the same way; modifies the working directory (doesn't change either of the files)
    - `$ rm` is the same way → modifies the working working directory; removes the directory entry from ., doesn't change the file at all
    - `$ touch {filename}` → creates an empty file
    - `$ chmod` → changes permissions for specific files
        - A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Omitted digits areassumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and restricted deletion or sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.
    - `$ echo`

        Echo the STRING(s) to standard output.

        -n do not output the trailing newline

        -e enable interpretation of backslash escapes

        -E disable interpretation of backslash escapes (default)

        --help display this help and exit

        --version
            output version information and exit

- Signal Flavors
    - `^C` → INT (interrupt) signal → 3
        - `kill -3 {PID}` → equivalent to sending ^C to the process. **Is this only with `$ kill` or is this with other processes as well??**
    - TERM (termination) signal → happens when you type `kill` with no other signal flavor (default for this process)
    - KILL signal → 9
    - HUP (hang-up) → signal sent to your program when you disconnect (e.g. when your ssh connection dies and programs are still running)
    - `^Z` → SUSP (suspend) signal
    - **Dozens of these signal flavors** → used by the user and the kernel to control programs and to tell them that something bad has happened and to change their behavior
        - Programs may clean up in different ways depending on the signal they receive; distinction is rather subtle

- POSIX (Portable Operating System Interface) model
  - each file has …
    - permissions/mode
      - **File permissions: 9 bits (e.g. `rw-r--r--`)**
        - rw- → owner of the file's permissions (ex. can read and write, but cannot execute)
        - r-- → other members of the owner's group (ex. can read the file, but nothing else)
        - r-- → everyone else (ex. can read the file, but nothing else)
    - file type (the first character of the permissions output; `d` → directory, `-` → regular file)
    - user (in my case, my SeasNet ID)
    - group (`csugrad` for me)
    - size (bits or bytes?)
    - timestamp (last modified date and time)
    - inode number
  - $PWD → prints the working directory; the shell automatically updates this for you
  - $PATH → this is a list of file paths separated by colons; the way the shell finds specific files is it searches recursively in each of the directories listed from left to right

- Directories
  - directories are a special kind of file; **map file names to file**
    - file name = sequence of bytes (256 is the limit); has to be non-empty, but cannot contain slash or NULL
    - **file name is NOT part of the file, part of the directory**
    - directories can map to other directories — gives the tree structure of the file system
  - root directory = /; distinct from the home directory (~) which you are in when you log in
  - directory maps **file names** to **inode numbers**!
  - By convention (to help navigate), each directory always has at least 2 directory entries
    - . → points the directory itself
    - .. → points to the directory's parent directory → root directory is a special case; it is its own root directory
- Inode Number
  - inode numbers assigned as the system is created; thus, files in the home directory tend to have much higher numbers than files in root directory
  - inode numbers are not actually machine addresses (or where the file is stored in memory), but it acts like it (pointer)
    - Why does it need it? Has to do with persisance; system tries to keep track of everything internally via inode numbers, has to expose those details to you on occasion when you're doing delicate things like renaming files, etc.
  - when I run `$ ls -li` in the root directory, I see **multiple files w/ the same inode number**
    - you can have multiple file systems on the same computer (is routine in linux); can happen if you have multiple flash drives on your system → going to have multiple trees
    - **this results in duplicate inode numbers**, as each file system has its own rules on inode num assignment
    - Is there a way to output on a file by file basis which file system it is a part of/stored on?
      - mount table (`$ mount`) shows all of the different file systems on the OS; Linux only shows 1 big tree, does not show all of the individual file systems without specific commands
      - There is a way to do this on a file-by-file basis → **CS 111 content!**
    - **file system** = collection of files sitting on a secondary storage device; typically either a flash drive or a hard disk
      - actual data on drive = bytes; represents the file system

- Hard Links
  - `$ ln <file1> <file2>` → creates a new file that is a **HARD** link to another file; editing one also edits the other, and they have the same inode number (are the same file)
    - Create new directory entries for each directory entry for the file!!
    - `$ ls -l` output will show the number of links pointing to a file (the last bit in the file type/permissions string)
    - `$ rm` **does not delete the file**; it deletes the name of the file — if there are multiple names for the file (multiple hard links) then `rm` is not effective (file can be easily recovered)
      - there is no index that will tell you all the names of a file; the only way is to walk through entire file system and check
    - when you remove all names for a file, then no one can access the file via directories; file system can reclaim the storage space
  - `$ ln` modifies . → creates a file named <file2> in the working directory; you may think this is a command to create file2, but it actually modifies the working directory
  - `$ mv` (which renames files) → works the same way; modifies the working directory (doesn't change either of the files)
  - `$ rm` is the same way → modifies the working working directory; removes the directory entry from ., doesn't change the file at all
  - Not allowed for directories
- Symbolic Links (Symlinks)
  - **Symlinks are NOT regular files** (unlike hard links), are a new type of file
  - Contents = the name of a file that is referencing, rather than the file itself
  - When it encounters a symbolic link, it splices in the contents of the symlink and keeps going
    - If a symlink starts with a slash, it starts from the root (absolute symlinks); if it starts with anything else, it stays where you are and keeps going from that first directory or file (relative symlinks)
    - Relative symlinks are always interpreted relative to the directory you found it in
  - Symbolic link still creates a directory entry
  - Very few commands do not follow symbolic links if they are the the final item in the path (ls and ln are two of them)
  - Symlinks are how Emacs does autosave …
    - It creates a symbolic link (hidden in ls because it starts with .) → creates a new file with the active buffer, and a symbolic link (not as a link to a file) to the metainformation about edited file
    - Keeps track of who is editing the file, what machine they're editing the file on, timestamp, process ID, etc. → this is how Emacs locks me out if I try to edit files on different Emacs sessions
- Shell Globbing Patterns
  - Shell globbing patterns are a bit different from regular expressions
    - Still have bracket expressions: [a-z0-9] → matches lower case letter or digit (same as in Emacs)
      - → means 0 or more characters, except at the start of a pattern that doesn't match dot → in Emacs, this would be .* (0 or more characters)
  - ? → matches any single character, where in Emacs, . matches any single characters

- Expansion Order in Shell Commands

  0. Overall structure of the program – `|| | && ; ;; if then else while for`
     0.5. Variable expansion – `$x` expands to be the value of variable x

  1. Tilde expansion
     - `~ ~/ ~eggert ~eggert/` → at start of word, expanded into directory that they stand for; work independently of whether the file you are mentioning actually exist
       `$ echo ~eggert/ouchy` → `/u/cs/fac/eggert/ouchy`;
       `$ echo ~yathish` → `/u/cs/ugrad/yathish`
  2. Command substitution
     - `$ (command)` → treats the output as if it was something you actually typed
       - `$ ls -ld $(find . -type d | grep -v var)` → runs `ls -ld` on every subdirectory in your home directory that does not have the word var in the name (`grep -v` → excludes all that match the search)
     - **What is the difference between this and the ` characters in the shell script??**
     - Similar to `$ command | sh` → executing a command and running its output in the shell
       - Difference is that sh is a subshell; you lose access to the output when the shell exits → command substitution keeps everything within the context of the current shell
  3. Arithmetic expansion
     `x=5`
     `y=$((x+10))` → y will be 15
     - Awkward way to do arithmetic; shell really isn't supposed to be used for arithmetic
  4. Field splitting
     - results of previous expansions are separated into words; look for spaces between words – quoted string does not experience field splitting
     - field splitting is controlled by a builtin variable in the shell called IFS
       - `$IFS` → typically contains space, tab, newline → the bounders/delimiters between parsed commands; you can change this
  5. **Pathname expansion (Globbing)**
     - if you see a word that is not quoted that has `* ? [...]`, then it uses those patterns to find files whose names match those patterns
  6. **Redirection** – changes where the I/O goes for the command
     - `i>` redirects output (1 is default) –> File descriptors: 0 = stdin, 1 = stdout, 2 = stderr
     - `<` redirects input (0 is default)
     - `>>` output, but append to a file rather than over-writing the file
     - `<>` file is available for either input or output; the underlying command can decide whether to read or write or both (shell doesn't care)
     - `i>&j` where i and j are integers; redirect i to be the same as j
       `foo >out.txt 2>&1` – redirects stderr to go to the same place that stdout goes to
       `foo 2>&1 | sort` – in addition to sending the standard output to the next command, send standard error to the same place
     - `i>&-` close the output
       - `>/dev/null` → you can write to this as much as you want, OS will simply discard
       - `>/dev/full` → if you try to write, OS will give an error that there is no space left

- Quoting in the Shell
  - Bunch of special characters that need to be escaped when put into quotation marks: $, (), etc.
  - Can escape them by placing a backslash in front: \$, \&, etc.
  - Also can use single quotation marks: ' → the string will extend until the next single quotation mark, no matter what
  - Double quotation marks – difference between double and single quotation marks: $ and ` are still special in double quoted strings

- Useful Tips
    - && → conditional AND, runs the first command and if it succeeds runs the second command
    - || → conditional OR, runs the first command and if it fails runs the second command
    -
- Practice Shell Scripts

Assignment #1:

Use the find command to find all directories that are located under (or are the same as) the directory /usr/local/cs, and that were modified after the directory /usr/local/cs/lib was modified. Sort the directory names alphabetically and list just the first five names, or list them all if there are fewer than five names.

```
$ find /usr/local/cs -newer /usr/local/cs/lib -type d | sort -d | head -n 5
```

How many regular files are in the Python source code distribution located in the directory matching the globbing pattern /usr/local/cs/src/python/Python-*? Do not count directories or symbolic links or anything else; just regular files.

```
$ find /usr/local/cs/src/python/Python-* -type f | wc -l #wc -l counts the number of output
lines
```

How many of the files in /usr/local/cs/bin are symbolic links?

```
$ find /usr/local/cs/bin -type l | wc -l
```

What is the oldest file in the /usr/lib64 directory? Use the last-modified time to determine age. Specify the name of the file without the /usr/lib64/ prefix. Don't ignore files whose names start with ".", but do ignore files under subdirectories of /usr/lib64/. Consider files of all types, that is, your answer might be a regular file, or a directory, or something else.

```
$ ls /usr/lib64 -atr | head -n 1 #-a gives all files, including ., -t sorts them newest to
oldest by last modified time, -r reverses this order giving oldest first)
```

Use the ps command to find your own login shell's process, all that process's ancestors, and all its descendants.

```
#!/bin/sh

echo "Log-in Shell:"
ps -o "pid,args" | head -n 2
MY_PID=`ps -o "pid" --no-headers | head -n 1`
MY_PPID=`ps -o "ppid" --no-headers | head -n 1`

echo -e "\nAncestor Processes: "
#ps -o "pid,args" --no-headers -p $MY_PPID
while [ $MY_PPID -ne 0 ]
do
    ps -o "pid,args" --no-headers -p $MY_PPID
    MY_PPID=`ps -o "ppid" --no-headers -p $MY_PPID`
done

echo -e "\nDescendant Processes: "
get_descendants()
{
    if [ ! -z "$1" ]
    then
```

```
        ps -o "pid,args" --no-headers --ppid $1

        for f in $(ps -o "pid" --no-headers --ppid $1)
        do
            get_descendants $f
        done
    fi
}
get_descendants $MY_PID
```

Practice Midterms:
Write a shell script to take a backup of your home directory. Place it into '~/backup'. Back up everything under your home directory except for '~/backup' itself.

```
#!/bin/sh
mkdir ~/backup
for FILE in $(ls ~ | grep -v backup)
do
        cp -r ~/$FILE ~/backup/$FILE
        #recursively copy files from subdirectories (-r)
done
```

The New York Times Spelling Bee is a daily puzzle that
gives you six letters plus one special letter. You solve it by spelling as many English words as you can, such that each word:

- contains at least four letters,
- contains at least one instance of the special letter, and
- consists entirely of letters taken from the seven-letter set.

For example, Tuesday's puzzle used the six letters "g", "h", "l", "m", "o", "y" plus the special letter "t", and solutions for it include the English words "hotly", "loot", and "mythology". However, "hot" is not a solution because it's too short, and "loom" is not a solution because it lacks "t". Write a shell command that solves Tuesday's puzzle by outputting solution words, one word per output line. Limit the output to words consisting entirely of lower-case letters. Your solution can use the file /usr/share/dict/linux.words.

```
grep -E '^[tghlmoy]{4}$' </usr/share/dict/linux.words | grep t >test.txt
```

Create a bash script that applies Caesar cipher; all letter shifted three to the right. Create another that sets … to be the character '@'

```
echo "$1" | tr 'a-zA-Z' 'd-za-cD-ZA-C'
echo "$1" | tr 'a-wX-Z' 'd-zA-C' | tr 'x-zA-W' '@'
```

Worksheets:
Write a script named "list-files.sh" that behaves exactly as the ls command (invoked with no arguments), except it prints out files separated by spaces instead of tabs, and terminates its output with a newline?

```
#!/bin/sh
for FILE in *
do
        echo -n "$FILE " #suppresses echo's builtin add trailing new lines functionality (-n)
        echo
done
```

Make a calculator that can handle addition, subtraction, multiplication, and division using bash

```
#!/bin/sh
if [ "$1" = "add" ] #this is how string equality works
then
    OUTPUT=$(($2+$3)) #this is how integer arithmetic is done, e.g. $((COUNT+1))
    echo "$2+$3 = $OUTPUT"
fi
if [ "$1" = "subtract" ]
then
    OUTPUT=$(($2-$3))
    echo "$2-$3 = $OUTPUT"
fi
if [ "$1" = "multiply" ]
then
    OUTPUT=$(($2*$3))
    echo "$2*$3 = $OUTPUT"
fi
if [ "$1" = "divide" ]
then
    OUTPUT=$(($2/$3))
    echo "$2/$3 = $OUTPUT"
fi
if [ "$1" = "help" ]
then
    echo "./calculator.sh OPTION ARG1 ARG2"
    echo "OPTIONS: add, substract, multiple, divide"
fi
```

Securing Emails: When either of you send an email, the last three lines of the email will contain the sender, the recipient, and the date. You use the SHA-1 hash function to convert the lines into hashes, and also randomly shuffle the order of these hashes. Write a shell script.

```
#!/bin/sh

var1=$(echo "from $2" | sha1sum) #sha1sum reads text from standard input
var2=$(echo "to $3" | sha1sum)
var3=$(echo "date $4" | sha1sum)

echo -e "$var1\n$var2\n$var3" | shuf >> $1
```

Write a shell script called verify.sh that verifies an email ends with the desired hashes.

```
#!/bin/sh

var1=$(echo "from $2" | sha1sum)
var2=$(echo "to $3" | sha1sum)
var3=$(echo "date $4" | sha1sum)
new_shah=$(echo -e "$var1\n$var2\n$var3" | sort)

file_shah=$(tail -n 3 $1 | sort)

echo "$new_shah"
echo "$file_shah"

if [ "$new_shah" = "$file_shah" ]
then
    echo "Verified!"
else
```

```
    echo "Not Verified; Possibly Fraudulent!"
fi
```

3. Emacs (Editor Macros)

- Overview
    - Emacs = text editor + IDE (integrated development environment)
    - Emacs is one of the simplest IDE’s out there; looking at the source code is more feasible than fancier development environments
    - When you type `$ emacs` , emacs will take over your screen → will give you a terminal interface
        - Does this by sending control characters to your virtual terminal that says clear screen and replace the characters on the screen with new ones …
    - Characters you type in emacs go to EMACS, NOT THE SHELL; the shell is waiting for emacs to exit

- Important Commands
    - Buffer/Window Commands
        - `C-x C-c` → Exit Emacs; if any unsaved changes, will prompt you to save active buffers
        - `C-x C-s` → Save the current buffer; for buffers associated with a file, write that buffer’s contents out to the file; replace the file’s contents with the buffer’s contents
            - `C-x s` → save ALL buffers, and do so interactively; Emacs will wander through all buffers, will see which ones don’t match their files’ contents, and ask if you want to save each in sequence
                - One way to make sure that buffers – caches of files – match the files
        - `C-x C-f` → the inverse of the above, short for “find file”; copying data from the file into the buffer → visiting a file in Emacs; can also be used to create new files; reads from a file into a newly created buffer
        - `C-x b` → switch between buffers
        - `C-g` → terminate a command mid-keystroke (in the event of a mistake) – keyboard quit; return to top level
        - `C-x 0` → dismiss the current window; if another window is open, leaves only that window
            - `C-x 1` → look only the current buffer
            - `C-x 2` → split the window (splits the current buffer between the two windows; editing one will edit the other, since there is only 1 active buffer)
        - `C-x o` → switch the cursor to the other window on the screen
        - Most of the characters on keyboard are bound to `self-insert-command` → will take the character that you type and paste it into the current buffer
            - `C-h k` for these will not work inside read-only buffers (such as the help buffers)
    - Shell Commands Within Emacs
        - `M-x shell RET` (enter) → will run a shell under emacs’ control, which will display; the difference from before is that emacs and the emacs sub-shell process are running in parallel, while the original shell is waiting for emacs to exit
        - `M-! <cmd> RET` → run individual shell commands without opening a new shell buffer
        - `M-| <cmd> RET` : run the command with the input being the current region
            - Region – everything between the **point** (where the cursor currently is) and **mark** (where you want the cursor to go) (think of this as highlighting the text between)
            - `C-@ or C-SPC` allows you to set the mark
    - Special Character Commands
        - `C-x 8 RET {UNICODE or NAME OF CHAR}` → insert any character given hex code/Unicode or character name
        - `TAB` → autocomplete
    - Help Commands
        - `C-x =` → tell me about the current character (the character at the cursor position)
            - `C-u C-x =` → gives more extensive character description; opens up a new buffer with character information

        - `C-h k RET {keystroke}` → gives help (additional documentation/information) about a particular keystroke
        - `C-h m` → explains the mode; tells the possible sequences you can type
        - `C-h b` → list all keybindings
    - Searching
        - `C-s` → search forward for a fixed string; useful when you don't want symbols in the input to be interpreted as regular expressions
        - `M-s` → search forward for match for regular expression (`C-M-s` on my terminal for some reason)
        - `C-w, M-w, C-y` → cut, copy, and yank/paste
        - Searching for non-ASCII characters …
            - Can search for individual non-ASCII characters using `C-s C-x 8 RET {CHAR NAME}`
            - More general search can be done using `M-s RET [^[:ascii:]]`
    - Python in Emacs
        - `M -x run-python` → will run a Python interpreter buffer in Emacs
        - `C -h f RET run-python` → will provide documentation for the run-python function

- Emacs Buffers
    - One way to build Emacs or text editors is to make all variables persistent; if things crash, we haven't lost all of our work → **PROBLEM: Too slow; flash drives are extremely slow compared to RAM (orders of magnitude slower)**
        - Do you run out of memory? No. Takes up the same amount of memory; still have to allocate space for it if its not persistent. Do you run out of secondary storage? No. On most machines, you have way more secondary storage than RAM → you have flash drive to burn
        - Whenever you change the variable, you'd have to write it out to flash; wear out the flash drive faster → unbearably slow for customers
    - Emacs **buffers** are "copies" of files that are **sitting in RAM** (not persistent); managed by emacs → emacs can change it, but if crash happens, the buffer is lost
        - Different from **files** (persistent storage), managed by the operating system
    - Both buffers and files both contain **character sequences** and **byte sequences** (there is a difference between the two)
        - On modern computers, byte = 8 bits (can represent any number from 0 to 255); can represent any character of a character set that is at most 256 characters → not enough
        - each character is represented as a sequence of bytes; ASCII chars are 1-byte characters, non-ASCII are multi-byte
- Emacs Modes
    - Emacs is a **modeful editor** → the action that it takes when you type a character depends on the context/mode Emacs is in
    - Many possible modes
        - Text mode → useful when you're just editing ordinary text; doesn't have to be ASCII text
        - C++ mode → when editing a C++ file; will indent according to indenting rules, etc.
        - Elisp mode
        - Shell mode → enters when you enter a subsidiary shell in Emacs
        - Fundamental mode → even simpler than text mode; doesn't even know how to format paragraphs, etc.
        - `dired` (DIRectory EDit) mode → allows emacs to edit directories/files!!
    - If you change the save the file extension using `C-x C-f`, Emacs will guess what mode is most apt for the file – from the extension, but also sometimes from the contents (you can change it, but is not recommended, especially w/ common file extensions .txt, .cpp, .py, etc.)

- Emacs Regular Expressions
    - Each regex is a pattern! Each pattern matches any of a possible set of strings
    - Possible to build very fast searches for regular expression using simple techniques; if you try to use fancier searches, you will find that code gets very complicated or very slow
        - Maximizes power of searching capabilities while keeping search program concise
    - When you do a search → tries to find the first match, but if there are several different matches that all start at the same spot, it finds the longest match (longest-leftmost match)

PATTERN	**STRINGS THAT CAN BE MATCHED**	**NOTES**
\	escape character	Can escape any special character in a regex; has no special meaning inside the character set
^Q^J	new line character	^Q → quote the next line, ^J → new line character; searching for new lines
a (ordinary character)	a (matches itself)	Simplest regular expression
. (period)	any single character other than newLine	.* matches any sequence of characters other than and until the newline (e.g. a.*b)
p1p2 (concatenation of two patterns)	any string s1s2 where p1 matches s1, p2 matches s2	
p*	0 or more occurrences of p	When preceded by brackets (e.g. [aeiou]*), it matches any sequence of lower case values. ab*c will match “ac”, matching 0 occurrences of b.
p+	1 or more occurrences of p	Equivalent to pp*
p1 \\| p2	OR (anything matched by p1 or matched by p2)	
[abc]	any single character inside the square bracket	Example: [xy] is equivalent to \(x\\|y\)
[a-eq-z0-9]	{“a”-“e”, “q”-“z”, 0-9}	- (ASCII minus) shows a continuous subset between two bounds; switching between ranges without spaces **(SPACES ARE SIGNIFICANT IN REGULAR EXPRESSIONS)**. If you want to put a closing square bracket into the set, you have to put it in front []a-z]. If you want to put minus sign, then you immediately follow it with closing bracket: []az-]
[^a-e]	^ → negates the set; matches everything that is NOT in the set	This particular regex, or [^a-z0-9] would match a newline character. [^^] matches any character except “^”. Doing negation in general in regular expressions can be an NP-complete problem.
^p	p matching at line start	^ outside of square brackets is VERY different from inside square brackets. **a^ does not match anything**, because it must be an a, and then the start of a line, which cannot happen without a newLine (can be very useful)

p$	p matching at end of the line	
[[:alpha:]]	matches anything within the known character set.	[:alpha:] = alphabetic characters, [:ascii:] = set of ASCII characters; there are many more (e.g. alphanumeric, digit, etc.)

4. Emacs Lisp

- Overview
    - Once you get to the documentation, put curser on file name, then press ENTER
        - within source code, put cursor on a function C -h f : can search source code for specific functions
    - Within *scratch* buffer, you can redefine functions, change variables, etc.
        - `C -x b RET *scratch*` = change to the scratch buffer
    - Emacs Source code ~ 90% Elisp, 10% C

    - Shell's goal is to be a small language to set up the programs you really want to run → program to configure other programs
    - Elisp is an **extension language** → Emacs is still in charge, but you attach small behavior changes in specific, defined circumstances
        - The problems that you'd solve in Elisp are quite different from what you'd solve in the shell

    - `(global-set-key “\C-cq” ‘occur)` : creates a shorthand for an existing Elisp function occur → in this case the searching for regular expressions
        - Can bind all sorts of functions to new keystrokes 🤩
        - The single quotation mark is required, otherwise it will execute the function or try to read as a variable
        - Assigning a keybinding overwrites the previous keybinding; can only have one to a function (only valid for the one session)
        - If you want to tailor your Emacs permanently, create a file called **~/.emacs.d/init.el** (contains all sorts of customized startup code)

- 3 Ways to Deal w/ Scripting Langauges
    - foo.el = typically what source code looks like (.el = Emacs Lisp)
    - foo.elc = Elisp byte code
        - `M -x byte-compile-file RET <bytecodefile>`
        - **Byte code interpreter** is an efficiency scheme; is much more efficient to run than interpreting the lisp source code directly
        - Advantage of **interpreting source code directly** = most easily debuggable
        - **Compiling x86-64 machine code** : fastest, but the most cancerous to debug
            - Assembly is a printable representation of machine code; easier for humans to read. Byte code is not a printable representation (considered lower level)
        - There is a set of subroutines inside Emacs that knows how to translate byte code into machine code (mini-compilers)
        - This part is machine dependent; on ARM, will be different
    - After compilation, the x86-64 CPU instruction pointer is assigned to the machine code → executed
- Elisp Objects
    - numbers → there is no limit on the size of integers, integer overflow
        - Emacs internally uses multiple precision arithmetic → represents large numbers as long bit-strings of a bunch of 64-bit words, taking as many words as it needs to represent the number than it stops
        - Arithmetic is going to be slower in Elisp that it is in C/C++
        - Setting global variables: (setq var_name value)
    - floating point numbers

- strings
- symbols
  - ‘var_name (single apostrophe before) will print out the symbol var_name, but will not evaluate it or call its value if its a variable
  - When would you need this without the values they stand for?? If you want to perform symbolic computation, you want to write a program that deals with aspects of your program
    - Symbols are used when you need the function name (code that deals with code)
    - Why would a string not suffice?
      - Has to do with making things go fast; in order to compare strings you have to look at their contents (two different strings can have the same contents)
      - Symbols are unique; if two symbols are spelled the same way, they have to be the same object → use symbols so that you have higher performance code when you want to run programs that are introspective or look at specific functions
- pairs/conses → need to understand this better!!
  - like tuples; pointer to a piece of storage that contains two objects
  - build a pair using function cons — short for construct → (cons val1 val2)
  - Elisp representation of pairs → (val1 . val2), with a period in the middle
  - Can chain cons’s together → (a b c . d) → cons (-n (cons val1 val2))
    - This is how lists are represented in Elisp, except the last value has to be an empty list (nil) → built using (cons -n (cons val1 nil))
- Part of the attitude in Elisp is that you should not try to overwrite/modify existing objects other than buffers
  - Emacs is all about modifying buffers, so this is perfectly acceptable
  - Good programming style in Elisp is if you need a list that’s like the list you have, but is perhaps a little longer, then **build a longer list**

```
(setq m ‘(x y z w))
(set l ‘(a b c))
(setq p (append l m)) → append is the standard function for this; different
from in Python
```

- Comparison in Elisp
  - There are different flavors of comparison → help you compare different things quickly
  - Numeric Comparison: (= 3 3.0) → not identical, but just numerically equal
  - Content Comparison: (equal x y) → compares contents of objects
    - equal can be expensive → it has to look at contents; e.g. recursively descending through lists, etc.
  - Pointer Comparison: (eq x y) → compares the machine addresses/identities of the object

- Practice Elisp Scripts

  Assignment 2

  Compute 2**(607 - 1) * (2**607 - 1) in the *scratch* buffer, by using the expt and other functions. Then compute number of bits in base 2 notation (take log base 2 of the number)

```
A. (* (expt 2 (- 607 1)) (- (expt 2 607) 1))
B. (+ (- 607 1) (log (- (expt 2 607) 1) 2))
```

  Design and implement a command M-x gps-line that acts like M-x what-line except that it says “Line 27/106” in contexts where M-x what-line would merely say “Line 27”

```
(defun gps-line ()
  "Prints the current line of the cursor in the buffer/total lines in the buffer"
  (interactive)
  (let ( (start (point-min))
        (n (line-number-at-pos))
        (counter (how-many "\n" (point-min))) )
    (if (= start 1)
       (message "Line %d/%d" n counter)
      (save-excursion
       (save-restriction
         (widen)
         (message "line %d/%d (narrowed line %d"
                (+ n (line-number-at-pos start) -1) counter n))))))
```

Worksheets
Can you adjust your Emacs settings to set tab indenting to 4 spaces?

```
(defun tabbing-indent ()
       (interactive)
       (insert "    "))
(global-set-key "\t" 'tabbing-indent)
```

Write a function called is-even that takes one argument and returns whether it is even.

```
(defun is-even(n1)
  (interactive)
  (= (% n1 2) 0))
(is-even 4) #returns t (True)
(is-even 3) #returns nil (False)
```

Write a function called is-current-line-even that takes no arguments, and uses the message function to print out whether the current line number is even or odd.

```
(defun is-current-line-even ()
  (interactive)
  (if (= (% (line-number-at-pos) 2) 0)
       (message “Even”)
     (message “Odd”)))
```

Write a function is-in that takes in an integer and a list of integers as parameters and returns whether or not the integer is in the list.

```
(defun is-in (val lst)
  (interactive)
  (if (member val lst)
      t
    nil))
(is-in 3 ‘(1 2 3)) #returns t

(defun is-in (val lst)
  (interactive)
  (let ((i) (output nil))
    (while lst
      (if (= val (pop lst)) #(car lst) will return head of cons; (cdr lst) will return
remainder of linked list
          (setq output t)))
    output))
(is-in 4 '(1 2 3)) #returns nil
```

5. Python

- Python vs. C++
    - BIG DIFFERENCE between Python and C++ : objects/classes are kept around at runtime, can print out type class objects
    - In C++, classes exist during development and during compilation — to tell it what code to run — but not at runtime
        - major difference between scripting vs. compiled languages
    - Part of the attraction of Python is convenience …
        - `>> [ty(val) for ty,val in zip(types, s.split(‘,’))]`; rather simple list comprehension, that would take 5-10 lines to execute in C++
- Python Overview
    - Python — derived from Perl and ABC; aggregates all of the prior “little languages,” but strips away the ambiguity? There is a clear way to write/formatting
        - Like Perl in functionality, can do a lot more than ABC can; more like ABC in that we want a solid, systematic, simple core
    - Every value in Python (or Elisp) is an object (object is a pointer to a piece of storage that represents the object’s value)
    - Each object has …
        - type (even a type () is an object of class <type>)
        - value
        - identity → machine address for the object
            - using the “is” command O(1), can compare identities between objects
            - This is not as straightforward as pointers in C++; Python interpreters can decide (on a case-by-base basis) to store equivalent string and small integer values in the same identity to save memory
            - Reassigning the value of one of the variables will result in different identity → integers are immutable, reassigning the value of the variable does not change the integer or the address of the integer
            - id (var) will return the identity of the variable
                - mostly used for debugging
                - “%x” % id(“xyz”) displays the memory address in hexadecimal
- Builtin Types for Python
    - Numbers (includes booleans)
    - Sequences (Lists, Strings, Tuples, Buffers)
    - Mappings (Dictionaries)
    - Callables (Functions, Classes)
- Assignment operators in Python do not create copies (e.g. e = {}, f = e); creating pointers internally to the same data structure → in this example, changing e also changes f
- Sequences
    - Sequence = an object that contains a sequence of other objects indexed by the numbers 0 to n-1
    - Some sequences are mutable (lists, buffers), others are immutable (strings, tuples)
    - Immutable sequence sequence can’t change; you can’t modify its contents, it can’t get longer/shorter
        - Immutability – applies only to the sequence itself; an immutable sequence can contain mutable objects that can be changed, and a mutable sequence can contain immutable objects that cannot be changed
    - Operations on Sequences
        - `s[i]` → gives you the ith element of the sequence; -len (s) <= i < len(s)
            - s[-1] → returns last item in the sequence; can use negative indices to count backwards from the end of the sequence
        - `len (s)` → gives you the number of items in the sequence
        - `s [i:j]` → gets subsequence starting from i up to (but not including) j
            - Does not change the original list; just returns a subsequence, that can be modified independently of the original

                - `len(s[i:j]) = j-i`
                - `s[i:]` → start at i and go to the end of the sequence; not quite the same as `s[i:-1]`, which does not include the last element of the sequence
                    - `s[:j]` → start from first element and go up to but not including j
                - When i=j, s[i]=[], empty sequence
            - `min (s), max(s)` → returns minimum and maximum element of a sequence
                - for this to work, the elements of a sequence must be comparable; ex. integers to strings cannot be compared → will produce a runtime error
                - min/max relies on an underlying comparison operator
            - `list (s)` → takes sequence, converts it to a list, and creates a new list with the same elements as the original; input sequence does not have to be a list (e.g. string, etc.)
    - Operations on Mutable Sequences
        - `s[i] = v` → replacing ith value of the squence with the value v
        - `s[i:j] = s1` → replacing the subsequence of s with the members of s1 (in order)
            - The new length of s can be larger or smaller than the original length depending on s1
            - `del s[i:j]` is equivalent to `s[i:j] = []` → deletes everything starting from i and up to (not including) j
        - `del s[i]` → deletes the ith value of the array in the list
            - O(n) - May not be cheap; have to move all values of the list after i up by 1
    - Operations on Lists
        - `l.append(v)` → adds the value v to the end of the list l; adds an extra element
            - Equivalent to `l[len(l):len(l)] = [v]`
            - O(1) amortized → if you do a function N times, total cost is O(N)
                - When you run append, occasionally it will be slower → Python is going to have to do internal memory management
                - Guaranteed if you do it a lot of times, internal memory management issues won't cost you too much; roughly proportional to the amount of times you run append
                - This is b/c of how lists are implemented internally in C Python (source code)
                    - lists have headers (describe how large the list is & how many spaces are allocated for it) → allocated space exceeds current length
                - If code is O(1), then it's also O(1) amortized → append is not O(1); reason is if length is equal to allocated space, Python has to allocate new storage
                    - Copies all elements from old storage to new storage, frees old storage, and changes the header to point to new storage → O(N), occasionally slow
        - `l.extend(s)` → appends all elements of s to the end of l (in order)
        - `l.count(v)` → counts occurences of v in l
        - `l.index(s)` → finds an occurence of s in v, and returns the index of that occurence
        - `l.insert(i,v)` → inserts value v just before the ith element of l, l[i]
        - `l.pop(i)` → returns l[i] and del l[i]
            - l.pop([i]) → would not write this; this is meta-notation in Python documentation → means that i is optional; if you leave i out, defaults to -1 (last element)
            - `l.pop()` → removes the last element of the list and deletes it → comes from stack data structure
        - `l.reverse()` → modifies all elements of l until they're in reverse order of what they used to be
        - `l.sort()` → sorts element of the list; is O (N logN)
    - Operations on Strings
        - `s.join(t)` → where t is a sequence (list, string, etc.); returns a string with all elements of t, separated by s (e.g. `"/".join("hello")` → 'h/e/l/l/o')
        - `s.replace(old, new [, maxreplace])` → walks through s, looking for instances of old, replaces it with new at most maxreplace times (optional argument)
- Dictionaries
    - Dictionary = mutable 'function' that maps immutable **keys** to **values**

- Keys can be strings, tuples; cannot be mutable objects (e.g. lists) → would make dictionaries very hard to implement (using hash tables)
- Operations on Dictionaries
  - `d[k]` → returns corresponding value for the key k; if there is no k, will produce a runtime error
  - `k in d` → returns true or false depending on whether k is a member of the dicitonary
  - `d[k] = i` → changes value attached to the key k
  - `d.get(k [, default])` → gets the value of the key k (d[k]), and if k is not in the dictionary, returns default (defaults to None)
  - `d.clear()` → deletes everything in dictionary, becomes empty; lists also have this operation
  - `d.copy()` → gives you a copy of the original dictionary
  - `d.items()` → gives you list of (k,v) pairs in the dictionary
  - `d.keys()` → returns list of keys in the dictionary
  - `d.values()` → returns list of values in the dictionary
  - `d.update(d1)` → takes all items in d1 and stuffing them into d, possibly overriding any keys in d that have the same name as what's in d1
    - `d.update(d)` would do nothing → updates d to include all items that are already in d; may take a while, Python isn't optimized for it
  - `d.popitem()` → returns the last (k,v) pair from the directionary and deletes it → in no particular order!! Not like lists, since this is based on hash table not stack
  - `d.setdefault(k,v)` → sets the kth item in d to v, but returns the old value of d[k] and returns v if there wasn't an old value

- Callables
  - Callable = Anything in Python that you can pretend is a function
  - Functions
    - `lambda x,y: x+y` → nameless function; can give this a name … `f = lambda x,y: x+y`
      - Use? Often, Python API's will have callbacks → have to pass a function to another function so that it will call your function → can pass it as a lambda
      - Don't have to give this a name … `(lambda x,y: x+y) (5,7)` → returns 12;

    ```
    def callfun(f,v,):
    return f(v)
    callfun (lambda n: n+1, 23)
    ```

    - Optional parameters:
      - `def f (a, b *c)` – *c → a tuple of trailing/optional arguments
      - `def g (a, b, **c)` – **c → a dictionary of remaining NAMED arguments
      - Can combine the above two notations – `def h (a, b*, c**)` → when calling the functions, the named, unnamed arguments must be in the order that you specified in definition
  - Classes

    ```
    class c (a,b):
          def __init__(self, a, b)
    ```

    - Python allows for multiple parent classes; conventionally, most important parent is first, etc.
    - You can have a class heirarchy → directed, acylic graph
      - If you are trying to find specific methods from the heirarchy, you do preorder **depth-first left-to-right traversal** to search → first class that defines the method gets called
    - `self` parameter → every method of the class gets passed the object on whose behalf the function is being called
      - You would not use self when writing a function that's not in a class (that's not a method)

- Dunder Methods in Classes
  - (Double Underscore/Dunder) __ `method` __ → indicates its part of the reserved space in Python, will treat this method specially → can be called spontaneously
    - Some have reasonable defaults (init, del, repr, etc.); some do not → if you try to call the ones without defaults, there will produce a runtime error

- `__init__` → allows you to set parameters for class objects; Python searches for the init method that matches a class object declaration
- `__dict__` → holds the classes' functions:
  - **Can change functions ... VERY DANGEROUS**
  - `c.__dict__['m'] = lambda self,x: ()`
- `__del__ (self)` → reclaims the storage the object takes up
  - Idea here is doing clean-ups that Python won't do for you; won't worry about dynamically allocated memory (Python is responsible for its own memory)
  - Cleans up resources that the funciton takes up; an example is connections to external databases – You would do the same in C++; its just that in C++ you have some other things to handle
- `__repr __ (self)`
  - default repr: `<__main__.{object name} object at {machine_address}>`
- `__str__ (self)` → both repr and str return string that is a printable representation of the object
  - Can be useful when writing programs where we want to konw what is going on with the object
  - `repr` is for LONG, formal representation; `str` is for shorter, informal representations (for debugging)
    `o1 != o2 → o1.__repr__() != o2._repr()`
  - The goal is that **repr uniquely identifies the object; str does not have that property**
- `__cmp__(self,other)` → compares 2 objects
  - returns -1 (if self < other), 0 (if self = other), 1 (if self > other)
  - this is what makes the min and max functions in Python work.
  - `__lt__ (self, other)` → invokes less than operator; returns true or false
  - `__le__` → less than or equal to; `__gt__` → greater than, etc.
  - These additional functions are necessary when **dealing with objects that cannot be reliably compared**; example is with floating point numbers that have a NaN value (e.g. 0/0, etc.)
- `__nonzero__ (self)` → returns a boolean; used whenever the object is needed in a context where you need to know whether its zero or not; whenever the object is used in a boolean context
  ```
  o = c()
  if o: print "True"
  ```

- Modules
  - Module = typically source is in one file; contains class definitions, function definitions, assignment statements, etc.
    - this collection of code forms a module that you want to load into Python all at once and execute
  - Modules executed through the import statement
    - This is an executable statement → different from C++; can put it anywhere in a script (dynamic construct, rather than dynamic)
    - Can make it conditional. – can execute within if statements, etc.
    - **Self-modifying code?? Your program is reaching into its source code and changing it (common in scripting languages)**
  - `import` statement …
    1. Creates a **namespace** – a dictionary that maps names to values; the keys are the identifiers used in the program
    2. Reads code from Python program and executes it in the context of the namespace
       - Populates the namespace with every **name** defined inside the body of the module; doesn't include print statements or other that simply read to standard output but don't define names
    3. Gives a name to the namespace: binds module name to the namespace
  - A lot of these modules have internal import statements; results in cascade → **the namespace will have sub namespaces**, but most likely don't care about that
  - More complex import statements:
    - `from {some module} import {name1}, {name2}` → acts like import, but the third step is different

- `from {module} import *` → has problems if anything in the module has a naming collision with something in your program; **safer to import specific names or to simply import the entire module** (can explicitly define whether the method is local or called from a module)
- `dir ()` → tells you what the names are in the current namespace
- Top level invocation of a module → `$ python3 {module.py} {args}`
  - like import, it executes the module but, `if __name__ == '__main__'` only runs when the module is called at the top level!! when you import, it will not be executed
  - Commonly used for unit testing
- Looking for modules –
  - `PYTHONPATH = '{dir1}:{dir2}'` → dir1 overlays dir2; look for module in both, if in both, then dir1 wins
  - `PYTHONPATH` is to Python what `PATH` is to shell commands; list of directories with packages in them

- Packages
  - Packages = collections of modules
  - What is the point of having modules and packages?? Why not just have classes and "super-classes" or a single tree structure??
    - Classes are about behavior at runtime. Classes and superclasses behave like each other at runtime → how the program actually runs
    - Packages are about how can you distribute software from the devleoeprs of the software to its users. Its about developers, and how they give code to each other
      - You want different groups of developers to have different packages that they maintain separately
    - **2 VERY DIFFERENT CONCERNS!**
  - pip
    - `pip` command (use `pip3` on Python3) → Manages what packages you have on your current system, and if necessary downloads packages from a standard location on the internet
    - `pip3 list` → Lists out all Python packages installed
    - `pip3 list --outdated` → Consults the internet to find which packages have more recent versions
      - If your program relies on buggy behavior in the current package, when they fix the bug, your program that used to work no longer works if you update spontaneously
      - Bleeding edge (w/ all bug fixes and security fixes) vs. Old (may be more buggy but your code still works)
    - `pip3 show --files` → gives you details about any particular package; can specify specific packages to look at
    - `pip3 list --format {some format (e.g. json)}`

  - Version numbers
    - Typically of the from A.B.C (A = major number, B = minor number, C = patchlevel)
    - If A increases (major changes), that means that the code written with the old version might not be compatible with the new version
    - If B changes, that means that the package has been extended; likely upwards runnable
    - C → minor fixes of bugs
  - Python Versions (Python 3 +)
    - PEP : Python Enhancement Proposal; you improve Python for the next Python Core release → provide source code (typically in Python, but sometimes in C)
      - An example: Making Python friendlier for GPU's; useful for portability for machine learning processes, etc.
      - In Python2, there was a print x+3, in Python3, there is a print function print (x+3) → having print be a function rather than a special case
      - These proposals can even fundamentally change the syntax of the language
  - Mixed Python
    - Python build process :

        - Pure Python (our appraoch)
            - package contains .py files (source code); easily distributed over the network
            - Portable; but it can be slow (not as fast as Rust, C++, or other low level languages)
        - Mixed Python + C++/Rust/etc.
            - Faster, but now the distribution mechanism gets more complex; distributer of mixed package must decide if they are shipping source code (expecting recipient to compile it? + instructions for doing that), or if they send a .so file (need one for every type of system/CPU/OS → each have different calling instructions/machine code)
                - Can have different machine code for different OS on the same architecture
                - By convention, .so files have the architecture and OS they are made for in their file name
            - Python packages will have conditional module executions based on different Python versions/releases, and architectures
    - .so files → Part of the module that is written in some other language; module glues together the machine code produced from that file into Python

- Practice Python Scripts

  Assignment #2
  Use Emacs to write a new script shuf.py in the style of randline.py but using Python 3 instead. Your script should implement the GNU shuf command that is part of GNU Coreutils.

```
"""
Write a random permutation of the input lines to standard output.

Vishal Yathish
COM SCI 35L
"""

import random
import sys
import argparse // can also use sys.argv[1] as above with import sys

class Shuf:
    def __init__ (self, filename=''):
        if filename != '':
            with open (filename, 'r') as f:
                self.lines = f.readlines()

    def function_body (self, arg_list, head, repeat, add_new_lines=False):
        if repeat:
            if head > 0:
                for i in range (head):
                    sys.stdout.write(random.choice(arg_list))
                    if add_new_lines:
                        sys.stdout.write('\n')
            else:
                while True:
                    sys.stdout.write(random.choice(arg_list))
                    if add_new_lines:
                        sys.stdout.write('\n')
        else:
            random.shuffle(arg_list)
            if head > 0:
                for i in range (head):
                    sys.stdout.write(arg_list[i])
                    if add_new_lines:
                        sys.stdout.write('\n')
            else:
                for i in arg_list:
                    sys.stdout.write(i) #writing to standard output
                    if add_new_lines:
                        sys.stdout.write('\n')

    def default (self, head=0, repeat=False):
        self.function_body (self.lines, head, repeat)

    def echo(self, args, head=0, repeat=False):
        self.function_body(args, head, repeat, True)

    def input_range (self, lower_bound, upper_bound, head=0, repeat=False):
        int_range = [str(i) for i in range (lower_bound, upper_bound+1)]
        if (head > upper_bound-lower_bound):
            head = 0
        self.function_body(int_range, head, repeat, True)

    def no_args (self): #reading from standard input
        lines = list(sys.stdin)
        self.function_body(lines, 0, False)

def main():
    usage_msg = """shuf [OPTION]... FILE
             or shuf -e [OPTION]... [ARG]...
             or shuf -i LO-HI [OPTION]...

             Write a random permutation of the input lines to standard output.
             With no FILE, or when FILE is -, read standard input."""

    parser = argparse.ArgumentParser(usage=usage_msg)
    parser.add_argument("-e", "--echo", dest="echo", action="store_true", help="treat each
ARG as an input line")
    parser.add_argument("-i", "--input-range", dest="inputrange", action="store", nargs=1,
                        help="treat each number LO through HI as an input line")
    parser.add_argument("-n", "--head-count", dest="headcount", nargs=1, help="output at
most COUNT lines")
    parser.add_argument("-r", "--repeat-count", dest="repeat", action="store_true",
                        help="output lines can be repeated")
    options, args = parser.parse_known_args()

    if options.echo and bool(options.inputrange):
        parser.error("shuf: cannot combine -e and -i options")

    h = 0
    if options.headcount:
        try:
            h = int (options.headcount[0])
        except ValueError:
            parser.error ("shuf: invalid line count: {}".format(options.headcount[0]))
    if h < 0:
        parser.error("shuf: invalid line count: {}".format(h))

    r=False
    if options.repeat: r=True
    if options.echo:
        generator = Shuf()
        generator.echo(args,h,r)
    elif options.inputrange:
        generator = Shuf()
        try:
            l = options.inputrange[0]
            lower_bound, upper_bound = int(l[0][0:l.index("-")]),
int(options.inputrange[0][l.index("-")+1:])
        except ValueError:
            parser.error(f"shuf: invalid input range: {options.inputrange[0]}")
        generator.input_range(lower_bound, upper_bound, h, r)
    elif args==[] or args[0]=='-':
        generator = Shuf()
        generator.no_args()
    else:
        generator = Shuf(args[-1])
        generator.default(h,r)

if __name__ == "__main__":
    main()
```

Write a Python function rotseq that takes two arguments, a sequence s and an integer n, and returns a list with the same elements as the sequence, except rotated by n items.

```
def rotseq(s, n):
      return [s[(i + n) % len(s)] for i in range(len(s))]
```

Write a Python program ‘dupsout’ that copies to standard output all input lines that are exact duplicates of earlier input lines. Here is an example of input (left column) and output (right column).

```
#!/usr/bin/env python3
import sys
seen = set()
for line in sys.stdin: // if taking specific number of arguments, can use sys.argv
      line = line.rstrip("\n") // strip would remove leading and right whitespace
      if line in seen:
            print(line)
      else:
            seen.add(line)
```

6. Network Software Construction

- Web Application Types
    - Peer-to-peer applications
        - All of the nodes can speak to each other (not always, can be partitioned); none of the nodes are in charge/director (all have equal rights)
        - Work collectively to solve a problem
    - Primary-secondary application
        - Primary node keeps track of all app’s tasks, responsibilities, and scheduling; each secondary node waits for instructions, performs is specific subfunction, and sends result back to the primary
        - Simpler to execute than the peer-to-peer approach; primary node can perform many optimizations
        - Also not very robust, since the primary node is a bottleneck & can have security vulnerabilities
    - Client-server application
        - Server is in responsible for all the data, connected to a database, etc.; clients (which each have their own tasks) send requests to servers and get responses back

        - Downside is performance → classic UI, users want responses back in 10 ms or less (for Eggert 5 ms)
        - Correctness → a lot of the performance tricks involve correctness issues; performing tasks in a different way than the user expects and hoping they don't notice LOL
- Problems with Client Server Applications
    - Performance Issues (Latency & Throughput)
        - **Latency** → Delay due to network transmission; since you have multiple progams runing across multiple machines, and you need to send messages to servers across vast disances, this **takes time**
            - How to Fix?? You don't do what a user wants to do. You do an approximation that you think is good enough → **You pretend to talk to the server, but really do functions locally**
                - Like Emacs having changes stored in a buffer rather than written to the file → Difference is that Emacs tells you that you haven't saved; most client server apps do not want users to know that this local processing occurs
            - **Caching** (most common way) → Client caches server data; acquires a copy of the data, and returns it - this data may be **slightly outdated (stale cache)**, but is typically good enough
                - Issue: **Cache validation** → Many ways to determine whether the cache is valid
                    A. Refreshing the cache; if you do this 1 per min, you know that your cache will be at most 1 min out of date
                    B. Checksum – You take the original data, and you apply a formula to it; Construct a 32 bit value that is a function of its input, **scrambled all the bytes of the input** → come up with a standard function that the client and server agree upon

```
unsigned sum = 0
while ( (c=getchar) != 0)
      sum = (sum << 3) ^ c ^ (sum >> 28)
return sum
```

                    C. Timestamps – Dependent on the context
        - **Throughput** → Number of problems per second that you can solve; in other words, number of requests per second that the server can answer!
            - Typically refers to throughput of the overall system; since server performs most of the processing/computation, is equivalent to **server throughput**
            - How to make the server go faster??
                - Multiple servers running in parallel
                    - In simplest model, each server is in charge of a specific subset of the data → each client knows which server to talk to to acquire specific datapoints
                    - Not necessarily so simple; may not be clear to client ahead of time which server has the data it wants
                    - Can also have multiple threads on the same CPU
                        - Advantage: Can communicate with each other very quickly; store into shared memory
                        - Disadvantage: Have to deal with ppl colliding and dealing with common data
                - Front end w/ parallel back end servers - primary server with a bunch of secondary servers
                    - Out of order requests responses –> You can have a server that handles requests in a different order than they came in
                        - **Correctness Issue** → From POV of outside world, some users get responses faster than others (can have drastic impacts)

                    - Serialization Issue → Answers come back in a different order; do they make sense in the order they come back??
                        - **Explain the results by constructing a serial order for the actions – doesn't even have to be accurate; just need to have a justification**
                - Servers operate on the common database; a database server can be consulted to come up with an explanation (simple examples should not need this). Problem is that the database server becomes a bottleneck → its possible to come up with explanation without the central server (beyond the scope of this course)
    - Correctness Issues
        - As seen above, these come up because you have performance problems, you "cheat" a little; but b/c of this, you now have to worry abt correctness
            - How you deal with this varies based on application or business context (e.g. weather apps vs. bank balance apps)
- Network Configuration Types
    - Circuit Switching
        - Invented 100+; used in telephone systems
        - How it works??
            - **Each phone connected (via wire) to a Central Office** (each org/facility had one, e.g. UCLA – regional coordinator for phones)
            - The Central Office has a big switch that **connects any incoming wire to a bunch of outgoing wires;** was initially a mechanical switch, eventually became an electronic switch
            - The outgoing wires talk to other switches (in different locations), and from there to other switches, etc.
                - There is a length limit to each of the connections (paths between switches); copper wires → signal gets too weak
                - Each of the switches have amplifiers; can take the analog signal coming in from the wire and redistribute it to more distant locations
            - **Essentially a wire circuit (with switches) connecting one phone to another between distant locations**
                - Speech can travel across this within the span of a few milliseconds – speed of light + delay in amplification in each switch
        - Advantages & Disadvantages??
            - Once set up, you have (almost) guaranteed performance (minus freak natural disasters); downside = cost!
                - You have to maintain enough spare wire to sustain the conversation, you have to maintain the connection over dead time (conversation stalled), etc. → wasted capacity
            - Also concern over potential destruction of Central Offices (e.g. Russian attack during the Cold War, etc.)
    - Packet Switching
        - First proposed by Paul Baran of the RAND corporation (UCLA alum); 1962 (actual practice took another ~10 years)
        - How it works??
            - Takes whatever message they want to send and break it up into **packets** (exact packet size depends on the hardware – 1K bytes is a reasonable number) – you take your voice signal → convert from analog to digital
            - Send each packet to the Central Office → from there to neighboring switches; send them in the general direction of the target destination
                - Some switches can get overloaded and drops packets → some packets don't get through

            - For some applications (e.g. bank balances) you cannot tolerate dropping packets; for some, it is acceptable though annoying (e.g. sending audio signal w/ high quality)
        - Packets arrive at destination in perhaps scrambled order → receiving smartphone can descramble the input packets
    - Advantages & Disadvantages??
        - **LESS WASTED CAPACITY** → when you are not talking, you don't send any packets; wires in the network can be utilized by other users
        - **MORE RESILIENT** → destruction of one node/switch will not take down the entire connection; packet transmission routes can reroute once network notices break **(fault tolerance)**
            - Each of the switches has a rough model of what Internet topology looks like – collaborate locally → can determine which routes are good/bad, and which ones travel in the direction of a chosen destination
            - Will have some disruption, as it takes time for topology change to propogate through the network → eventually, network will settle down & packets can get through (at reduced capacity)
        - **Performance will be an issue** → delay between message sending & recieving is much more than circuit switching; smartphones may need to rearrange packets before rendering them into analog
        - **Proper packet receipt and descrambling is also an issue**

- Packets
    - Contents:
        - **Headers** – overhead; used by the network to determine location, etc.
        - Payloads – data that the user wants to send
    - Are exchanged via **protocols!!**
    - Three basic problems ….
        1. **Packets can be lost**
            - You might think that packet lost b/c communication link goes down → this does happen, but is not the main reason
            - Usual reason = switches (routers) gets overloaded; too many incoming packets, doesn't have enough processing power to figure out where to send them → discards packets
        2. **Packets can be recieved out of order**
            - They took different routes, or the server was kind of overloaded and delayed transmitting packets or sent them out of order
        3. **Packets can be duplicated**
            - Recipient can get the same packet twice, even though it was only sent once
            - In today's Internet, **happens b/c intervening network hardware is misconfigured**, or router misconfigured as a bridge
            - Or a router can send a packet to another router, think the line went dead in the middle (falsely) and send a duplicate packet
- Protocols (Overview)
    - Protocols – agreements between the sender and recipient so that when the reciepient sees a packet (particular set of bits), it will know what to do with them
        - Context around the message; lots of reasons a sender might want to send a message (e.g. audio, video, bank balance, etc.)
        - Help recipient decide what to do with the message & helps the sender decide what to send next
        - Protocols are implemented in software running on endpoints or on routers (some implemented on hardware); just conventions!
    - Can design protocols (NOT JUST ONE; an entire suite) to deal with three basic problems with packets!
        - @ Low Level, packets have issues defined above
        - @ High Level, applications (and users) don't want to know or deal with these low level issues

- Protocol Layers
    - Suite of protocols formed into layers … (ordered below in ascending order) → Internet Model (there are other models, e.g. OSI model – division of layers is a contentious point)
        - Link Layer (Lowest Level) → very close to the hardware; don't know much about the Internet at all
            - Link Layer protocols are from one switch/router to the next; inform the switches how to transmit data across individual connections (wires) → point-to-point
            - Hardware oriented/specific -- each form of hardware is different (radio link layer protocols are quite different from wire link layer protocols)
        - Internet Layer → sending individual packets across the Internet
            - Not trying to assemble them or deal with packet drops; just transmitting them from one point in the network to another non-adjacent point
        - Transport Layer → building a data channel between two points; sending multiple packets (e.g. a conversation)
            - Imitating circuit switching?? Not really, as packets are still sent → at this layer of abstraction, what users see are data channels
        - Application Layer (Top Level) → certain class of applications (e.g telephony – audio transmission –, video transmission, bank transactions, etc.)
            - Protocols are designed for a common set of applications
            - Here are where web applications are built!
    - Internet Layer
        - Internet Protocol (IP) → specified by a team led by Jon Postel
            - Talks about what it is in the header, the early part of each packet that controls information abt the packet; what the network needs to operate
            - Header contains …
                - **Length** – number of bytes in a packet
                - **Protocol Number** – IP was designed to have multiple layers on top of it; if you want to have a special purpose protocol for a web-app, has a specific protocol number
                - Source & Destination Address **(IP addresses)** – 32 bit binary numbers; every node on the Internet has a unique ID
                    - By convention, written using a 4 byte decimal sequence separated by dots (e.g. 192.54.239.17)
                - **TTL (Time-To-Live)** (hop count) – limits the number of connections that packets can traverse → after TTL field goes to 0, packet is dropped
                    - Designed to prevent infinite loops; packets traveling between nodes without ever reaching destination
                    - Initial TTL field values are determined by the application
                - **Checksum** – 16 bit, very simple; intended to catch most hardware errors, but not cryptographically secure
                    - In theory, cheksumming should be done @ the Link Layer; since it is the low-level hardware's responsibility to ensure that each link transmits packet correctly → doesn't happen in practice
                    - **<u>End-to-end checksum</u>** → sender computes, recipient checks the checksum; determines whether packet has been corrupted (not a cryptographic concept; geared for reliability)
                        - Trying to defend against routers getting misconfigured
                        - If packet deemed invalid, the safest course of action = discarding
                        - **<u>Important property of the philosophy of the Internet (end-to-end check)!!</u>**
            - How to get around 4 billion IP address limit??
                - Successor to IPv4 → IPv6 (1996) – Has 128 bit IP addresses (represented in hexademical notation)

        - IP addresses are territorial – ranges of IP addresses given out by geographic location (countries)
            - Since US invented the Internet, has bunch more IP addresses than our "fair share" → can survive with IPv4 pretty easily
            - Other countries are running IPv6 – there are ways of communicating over the two styles of network
                - Every modern router knows how to deal with both kinds of packets, and knows how to convert between the two types
        - Because IPv6 has larger IP addresses, has larger headers; less efficient than IPv4 → if you don't need the extra features, then can be a net loss
    - Why would a computer have both IPv4 & IPv6 addresses??
        - Convenient for the same server to have multiple IP addresses – might be on multiple networks (e.g. two wires, one wire-one wireless, etc.) & talking to multiple routers (one might be IPv4, IPv6)
        - Can have a single router that's willing to take v4 and v6 packets, and send those packets; useful for sending signals to ppl in different countries (usually though you go through a gateway)
- Transport Layer
    - User Datagram Protocol (UDP) – thin layer over iP
        - You have an application that wants to operate at a low level; willing to deal with packets being lost – application is robust
    - **Transmission Control Protocol** (TCP) (V. Cerf [UCLA] & Bob Khan [Princeton])
        - You want to go one level up from packets → data streams
        - Provides stream of data that are …
            - Reliable – data is not lost
            - Ordered – data recieved in order of transmission; not scrambled
            - Error Checked – the goal is to do end-to-end checking as many times as possible; not trusting that the lower-levels have checked properly or throughly
                - There is a performance cost; each layer introduces overhead – we're willing to pay the cost to get functionality (more reliability) → configuration problems are a source of A LOT of problems, these consistent checks are important
        - In order to do these, performs these tasks …
            - Divides the data stream into packets
            - Reassembly – when the recipient node gets packets out of order, TCP protocol reassembles these & informs the recipient
            - Retransmission – if recipient notices that a packet has been dropped, TCP sends a packet to the sender notifying that a specific packet was lost, and sender will retransmit
                - Sender then has to be more complicated; whenever it sends a packet has to be able to retransmit it later on
            - Flow Control –
                - Better to not send all packets at once; will overload the network and make it less efficient
                - Make sure that you don't send packets too quickly
    - UDP and TCP are two alternative protocols in the transport layer
    - In theory → intervening routers don't need to know you're using TCP; in practice → routers (smart) are able to identify TCP packet streams, and set up fast links for certain streams
- Application Layer
    - **Hypertext Transfer Protocol (HTTP) –** Basis of the web; **runs atop TCP**
        - If you are trying to transmit a webpage from a server to a client, you don't want parts of it dropped, or duplicated, etc.
    - **Real-time Transport Protocol (RTP)**

            - Runs atop UDP – designed for applications where you have some real-time constraints operating
                - If you are sending audio or video (or some combination), delays in packet arrival can drastically impact user experience (e.g. viewer looking at a blank screen for 3 sec in the middle of watching the Super Bowl)
                - TCP would cause jitter –
                    - If a packet is dropped, recipient has to ask sender for retransmission, etc.
                    - In the meantime, the recipient would have gotten later video frames, but cannot use/display, since TCP insists that the packets must be in order
                - Ex. Zoom uses both …
                    - Web Real-time Control (WebRTC) – for audio and video
                        - deals with packet drops in a different way → make transmission problems visible for the client
                        - lets (e.g) video streams come with little holes in it; recipient will see a blank screen, or large blocks of color rather than fine grain, etc.
                    - Session Initiation Protocol (SIP) – descendant of the old circuit switching technology
                        - At the lower level, built on packet switching; building a model on top of that that spends time setting up an end-to-end connection for more reliable transmission (a core property of circuit switching)
                        - The data transmission would follow a uniform path, as seen in circuit switching; however, it maintains the fault tolerance of packet switching

- Hypertext Transfer Protocol (HTTP)
    - World Wide Web (WWW) (Tim Berners Lee – 1991)
        - Main Idea:
            - Simple protocol for sharing documents (web-pages) → HTTP (v1)
            - Simple format for web-pages
    - HTTP (v1)
        - Simple request-response protocol over TCP; TCP has solved the problem of reliable streams over unreliable packet network
        - Client sends server a message (command & a bunch of arguments) → ships it via TCP to the server

```
GET /abc.html HTTP/1.0
(empty line)
```

        - Server sees this request, and gives a response …

```
200 (comment)
[header data]
(empty line)
[payload → HTML document]
```

        - Once that is done, the connection (data channel) closes → very simple use of the TCP protocol
        - There is a set of 3 digit codes specifying whether request was successful, and if not why (e.g. 200 – successful, 404 – there is no web page by that name, etc.)

```
$ telnet {website url} http RET GET / HTTP/1.0
```

        - Nowadays, this transmission is encrypted; in v1, the HTML doc travels unencrypted over the network (obvious security problems) – if the routers are attacked, wrong info can be displayed, or wrong requests sent
        -
    - HTTP 1.1

```
$ gnutls-cli {website url} RET GET /en-US/ HTTP/1.1
```

        Host: {website url} - GNU Top Level Security Command Line Interface
    - Talk to websites using **HTTPS** – secure variant of HTTP
    - Have to specify what website to host; most web servers nowadays service lots of different website
    - 1.1 improved on this; can keep connection alive, thus improving on performance (takes time to set up/tear down connection)
- HTTP is evolving –
    - 1.1 – You don't have to close the connection
    - HTTP/2 (2015) – aimed to make both HTTP and HTTPS more efficient
        - Header compression – while payload compression is relatively easy, headers had been previously left uncompressed (inefficient)
- HTTP 2
    - 69% of top 1000 websites uses v2; came out in 2015
    - Features:
        - Header compression - meta information about what you're sending
        - Server Push
            - Under HTTP1, client is in charge of the conversation; server doesn't initiate anything
            - w/ HTTP1, client keeps asking the server for updates (leads to excess delay/updates) - if changes occur, client is not notified until next request
            - Server Push = server can initiate a message, not in response to speciifc request for client
                - Clients have to be ready to accept a message at any time
        - Pipeline
            - Attempts to improve performance; client can send several request without waiting for responses
                - client gets answers back as quickly as possible
            - Complexity distadvantage - client has to decide what the follow up requests will be before it gets an answer to original request → does not always work, if you need answer to first question to ask second one
        - Multiplexing
            - Pipelining on steroids → don't insist that the answers come back in the same order that they came out (client gets answers out of order)
            - Protocol gets more complicated; each response needs to have a tag indicating which request they are responding to
            - Allows the server to answer simple questions faster, and take time with more complex questions; server has greater chance to re-order requests
                - Reordering is one of the major approaches servers use to improve throughput; for example, if the database is organized alphabetically, can answer questions in alphabetical order instead of order requests were made
            - Reordering occurs at the servers-side, does not occur in the network — TCP prevents packet scrambling
- HTTP 3
    - Came out in 2022; 39% of top 1000 websites support v3
    - Features:
        - More multiplexing
            - Real problem with v2 and v1 is that is not suitable for video and audio streams; reason its not good is built atop TCP
            - If you send video stream over TCP and packet dropped, protocol holds up all other processes until packet is found (packets must be recieved in order) — causes jitter
            - TCP no longer required for HTTP3 → Built atop another protocol: QUIC
            - Quic = TCP + UDP (adjustable)
                - Can use Quic to set up data steam & send packets

            - If no errors, receipeitn will get packets in order; if dropped, recipient can be notified and choose to proceed without dropped packet
            - More complex, but works better with streams
        - Bit of rivalry between Quic-based protocols and protocols along lines of Web RTC (Zoom)
            - Web RTC focuses just on the problem of streams; does't try to do as much else - Quic tries to be everything if you wanted to set up TCP (more general & complex)
            - Quic - also invented by Google
    - Avoid head-of-line blocking delays
        - Head of line blocking = In TCP, when packet is missing is at head of line, remaining packets won't be delivered until first one is delivered
    - Heterogenous nodes:
        - Little endian machine vs big endian machine → disagree on the ordering of bytes
        - When transmitting integer over network, you assemble each byte and ship bytes; if you have little endian machine sending bytes, big endian will interpret it in the reverse order
        - Need a way to transmit such that even if two machines disagree on ordering, bytes ordered correctly
        - Also need to figure out a way to ship data structures across the network → if you assume that most are trees, you need to find a way to ship "trees on wires"
        - So server can take disassembled, sequential tree and reconstruct in its own memory equivalent tree
            - May be in a different architecture, but the meaning is going to be the same
        - Second major thing that Tim Berners Lee did
            - They alr had a way to put trees on wire (serialization - turning tree into sequence that anyone can understand)
            - SGML – addresses portability problem; sending text/document-representations of tree data structures over the web
            - Berners-Lee → took SGML and changed it for the web; added notions for creating links, audio streams, images & changed the language to lower case (minor syntactic changes) → **HTML** (far more popular than SGML ever was)
- SGML (Standard Generalized Markup Language)
    - Have a single portable representation for books and papers, etc. - Designed by IBM
    - Before this, every publisher had their own format which meant that they couldn't share documents
    - Before the internet, would send this document on floppy disks, etc.

```
<QUOTE TYPE = 'example'>
      Hello from the <ITALICS> other side </ITALICS>.
</QUOTE>
```

    - Angle brackets show meta information - each publisher substitutes in the format that they prefer
    - For each publisher (who perhaps specialize in specific subject matters) → need to have specialized versions of the document that have unique format/notation/features (e.g. chemistry documents)
        - Document type declaration (DTD) → extra document that specifies what things you can put in document and what attributes they can have
        - Specifies syntax for different notations, paired with English language documentation to support and explain
    - Can be represented as nodes in a tree → SGML = a way to textually represent a data structure in a way that can be shipped across a network and reconstructed by the recipient in whatever format fits their needs
    - Downside: Focused on the printing industry (things like fonts), not as good for the net

- HTML (HypterText Markup Language)
    - Each element = a node in the tree; surrounded by tags: <abc> </abc> (identifiers must match)
    - Closing tags are often optional, can be ommited
    - Elements have attributes & content, representing subtree information
        - Can contain ordinary text (or entities) or new elements
        - Entities: &amp = "&", &lt = "<" → this is how you build up symbols without being interpreted as part of HTML (shorthand for text)
    - "HTML syntax" without tying it to web-pages → take HTML and split it into two pieces (syntax and meaning for when looking at webpages - DTD)
        - **XML - Extensible Markup Language** → Serializing tree structured data
        - Can ship other kinds of data across the internet (e.g. map data, timestamps, etc.);
        - XML documents cannot be formatted/read by standard browsers
    - DTD's for HTML
        - Standardization process is too slow; took too long between idea for having element for video streams and mass adoption of varieties (competition)
        - HTML uses living DTD; there is a standard source on the web that says here's what HTML5 is today, and it might change — changes reasonably rapidly (e.g. adding controls to video streams is not that hard)
    - Document Object Model (DOM)
        - Give you a way of accessing tree that you're going to send over the network and access conveniently from object oriented program
            - Tree is in memory, but you intend to either recieve or send tree → can be communicated as HTML
                - You can point to any object in tree and get string representations
            - Also want to be able to traverse, search and modify tree → if you don't like what the user sees on a webpage, you can change/update it
        - Tree can also contain programs → in some elements of text is source code of program that you can run, which can also in turn use DOM to access tree
- JavaScript
    - Like Python: Scripting Language; it's object oriented, but doesn't have classes
    - Can be hooked into HTML/DOM (internal representation of HTML)

```
<script src="https://www.ucla.edu/js/gobruins.js"></script>
<script src="gobruins.js"/>
<script> alert ("Hello World");</script>
```

    - Source code in <script> tags is often obfuscated (meaning has been stripped away, and hard to read)
    - JXS
        - If you're using JavaScript to generate DOM; you want JS in website to dynamically change what you see → can do this in pure JavaScript
        - For common situations when ur trying to create HTML, can use JSX → like a preprocesser for JavaScript
        - Can use a new syntax that makes it easy to write this kind of code

```
const header = <h1 lang={language}> assignment 3 </h1>;
```

    - In pure JS, this would be a syntax error; but JSX allows you to integrate HTML for creating elements

- Browsers (Browser Rendering Pipeline)
    - Browser gets HTML doc → Renders doc onto user's screen
        - Classic way to do this: Browser gets entire doc from web server → Sticks into RAM → Turns it into tree → Then starts rendering. This approach works, but is very slow esp with large websites
        - Modern Browsers start rendering before entire webpage has arrived, or before entire webpage has been processed → Show partial display

- Optimizations in the pipeline:
    - If it looks like element in DOM won't be on screen, browser won't analyze it
    - Elements may have priorities → if element is low priority, browser will get to that element later (JS code won't be immediately executed)
    - Browser can guess layout of page and revise later (e.g. will reformat tables, insert items into display, etc.) → code may be executed in a different order
        - High performance code → can be notified of these events; can have hooks in JS that gets revoked if browser re-renders, etc.

7. **Version Control**

- Backups
    - Operations staff is responsible for dealing with both hardware problems or software bugs or human errors (someone mistakenly trashes data, etc.) or deliberate attacks
    - One approach: periodically, make a copy of everything to a backup device (not a snapshot)
        - CPU → Flash Drive or Hard Drive → Backup Device (slower and cheaper)
        - Flash tends to be faster, more expensive (tends to be with smaller devices); Hard drives are cheaper per byte
        - This is a simple approach, but kind of expensive; modern flash drives typically have 1-10 Terabytes, and modern machines copy data at around 100 megabytes/second
            - 1 TB copied in ~ 8 hrs; here we assume that hardware is going flat out, there may be glitches, etc.
            - While this copying happens, what happens when CPU tries to make changes? You will be backing up data that is mutating as you speak, backup copy will not correspond to any snapshot of any data in the system
            - Will have a backup copy that doesn't match
            - "Down for maintenance" → freeze activity on the system to prevent changes while backups are being made
    - `cd .snapshot` → in home directory; hidden directory on Linux file system → hourly, nightly, weekly snapshots of your home directory
        - Other way: Have a filesystem that occasionally takes snapshots; snapshots are atomic, thereon out, the snapshoted file will be exactly the same as it was at the time of write
        - Take snapshots through **CoW (copy on write)** → don't actually make a copy of the file until someone writes to it
            - Commonly done at the block level not the file level
    - What to backup?
        - Data in files (file contents)
        - Metadata about files (e.g. permissions, timestamps, etc.) → ls -l output
    - What level to backup?
        - What the user sees (data + metadata)
        - Underlying blocks used to implement file system
    - How to reduce costs of backups
        1. Do them less often → backup at longer and longer intervals - more likely to lose important data
        2. Use cheaper (slower) backup device → flash to hard disk
        3. Use third-party service; back up to AWS or across the network → slower but perhaps a good price; typically this is only cheaper for smaller companies/projects
        4. Incremental backups (e.g. backup only files that have changed)
            - In the worst case, you'll have to search through all backups to find a snapshot
            - Typically done in a schedule → every now and then you do a full backup so don't have to save full backups before that, then incremental backups daily

            A. Don't store entire new file, but just the changes
                - Generalization of the concept of (4)
                - `diff -u A B > A-B` → shows most recent changes to a file; stored in a relatively smaller file → can backup these instead of the whole file (will take more CPU time than the standard CoW approach)
                - `patch A < A-B, patch -r B <A-B`

5. Periodically discard old backups (save storage space) → also do this on a schedule (known to users)

- Features of Version Control System
  - What you need from version control system?
    - History of Your Project → records both data and metadata (file metadata, who made the last change, etc.)
      - WHY? Get explanations for why software is the designed the way it is.
        - Will include an extra set of comments (commit messages) that pertain specifically to the change, not the underlying module → belong to the metadata
      - you want connections to other parts of your software dev system; e.g. bug report database
    - Future of your Project → want the version control system to maintain plans for future changes
      - Can have conflicting plans between software developers
  - Useful Features:
    - Keep histories (ideally indefinitely)
    - Modeling changes → How smart is your diff program? Comes up with a compact representation of your changes
      - Can be dumber, and possibly come up with long output, where it could be much shorter
    - Store metadata about the history
      - Information about what you did and when; may be in random order compared to ordering of the file system
    - Atomic changes
      - You want to have single atomic changes that cover the entire scope ofthe project — if you are renaming functions, all files that reference the functions should be changed in one update
    - Hooks
      - Occur when part of the procedure for making changes is automated; you want to tailer
      - E.g. pre-commit hook → added conditions to the commit process, you want to tailer your version control system for your project → can help guide develoeprs into processes that are good for your particular project
    - Format conversion of source
    - Cryptographically signed commits
- Intro to Git
  - Git repository = object database recording history + index (staging area → your plans for the next commit/change); many commands look and change only the index (this is where you actually edit the file contents - current version)
  - `git init <directory_name>` → new empty project
  - `git clone “url or filename”` → will clone an entire repository from some spot on the network or elsewhere on filesystem and give you a copy of the repository; also creates an empty index
    - Copies the repository then extracts working files from the latest version of the repository
    - Local Clone: cloned .git folder is very small → history is read-only; uses a hard link to the history on another system, since you don’t have to modify it
    - Remote Clone: actually have to create new version of the history onto file system
  - Working File <⟶ Index (staging area) <⟶ History
  - `git add “filename”` → updates the index with latest edits from the working file; old state is lost
  - `git commit` → goes from index into your history (adds to the history, does not replace existing history)
    - Commit Message convention:
      - Really short elevator pitch (< 50 char first line)
      - Empty Line
      - Good explanation of why that will make sense to you/others 6 months from now
  - `git diff` — can compare against HEAD (history), index (plans for next commit), or working files
    - `git diff` → compare unstaged changes
    - `git diff --staged` → compare staged changes
    - `git diff <commit1> <commit2>` → can use for branches as well
    - `git diff <file>` → compare changes between curr working dir and file (can use for commit)

- `git rm “filename”` → puts removal of a file into the index

- `git status` → tells you what you’re branch on, untracked files, tracked files (added, not committed)
- `git log` → shows you commit history, in roughly reverse time order
  - options: `--pretty=format`, etc.
  - `git log HEAD^...HEAD` → HEAD^ is the commit immediately before (parent) → HEAD^! shorthand
  - Algebra for naming commit:
    - HEAD stands for the most recent one
    - c^ = immediate parent of C
    - a…b → all commits in this range (a,b]
- `git ls-file`
  - Lists all files under version control in the current commit
- `git grep “pattern”` → searches source files for specific patterns; can inspect contents of working files, history, etc.
- `.gitignore`
  - Ordering of patterns matters
  - Globbing patterns have slight differences with other regular expressions
- `git clean` → removes files that looks like we’re not interested in them
  - git clean -n → don’t actually do it, just list the files that you would clean if you did
  - git clean -f → uses force to clean; be careful!
- `git checkout HEAD***` → change working files to be what they were three commits ago; thrown away working files that you had, and replace them with new ones that match what they were three versions ago
- `git config`
  - Can edit git configuration; can edit for project you are working on, configuration for current directory (stored in .git/config), or configuration for all your projects
  - Not under git control → on purpose; if it were under git control, you could mess up a repo by putting bad info into .git/config file
  - Stuff in gitignore is not delicate enough to actually mess up your repo; .git/config is!
  - ./autogen.sh or ./bootstrap → shell scripts that have standardized git config commands; developers trust each other enough not to mess up each others’ repositories

  - $HOME/.gitconfig → home git configuration file; also has parameters that can be modified → how you use git for every project
- `git blame “filename”` → goes through history of file; for each line in file, what most recent commit affected that line, who made that change, when that change occured, and shorthand version of commit id
  - commit id = 40 hex digit numbers; git will allow you to abbreviate them, give an initial prefix and this is good enough (as long as its unambiguous)
  - way of figuring out who changed what line of code and why

- Distributed Version Control System (DVCS)
  - Centralized version control system → if you want to change source code, you all have to communicate with a central server that is in charge of the history
    - Advantage of centralized approach = easy to explain and understand; disadvantage: performance → if you want to make a change, it is going to take a while - network may be slow, etc.
    - Doesn’t scale very well
  - Git …
    - Have several different repositories on several different computers (peer to peer network); each has full history
    - Attacks performance issue as you just have a copy of the repo on your computer; very fast
    - Problem: versions of the history might disagree
  - `git fetch` → takes what remote repo looks like, and copy it to my repository; if managed correctly, you will always be adding to history, no deletions

        - puts upstream changes into branch origin / master → communicated changes down from the remote to my local clone
        - This is your opinion of the remote repository; not necessarily what is actually in the remote repository
    - `git pull` → does 2 things: 1. git fetch, 2. git merge
        - `git merge` → attempts to take all upstream fetched changes and combine them with your repository to produce new combined repository that works
        - Only with merging and pulling that you run into trouble with collisions
        - Pull request → more of a GitHub notion; would like someone else to do a git pull, accepting my branch as the remote (nothin to do with these other things)
    - `git push` → rarer (usage) than git fetch; publish your changes on your branch to the upstream/remote repository
        - most dangerous command; once you do git push, your changes will be available to all developers/users
        - Some problems: suppose your opinion of the remote repo is out of date (didnt use git fetch recently enough); git push will fail
- Branches
    - Almost every commit in repo will have a parent; commit that it is based on
        - Oldest commit will not have a parent
        - HEAD = latest commit
    - We'd like to be able to add new commits, but allow other developers/ourselves in the future to follow along with the line of development
    - Branch = lightweight, movable name of commit that is intended to follow a line of development
        - if you're on a branch, and make a commit → name of branch will update to be the new commit
        - git branch → will list burrent branches
            - git branch <name_of_branch> <commit_hash> → name of branch at commit hash
        - git branch -D → remove branch
    - `git checkout -b "newbranchname "branchname"` → I want to switch to branchname, but also create a new branch called newbranchname that is on branchname → result will be, you are on newbranchname
    - Why branches?
        - Multiple developers/multiple views of what the future should look like — let other ppl look at it
        - Multiple releases of the same software
        - Hotfix = fix that is applied to all branches, needed by all - apply it selectively to branches; doesn't scale it very well
- Tags
    - Tags let you name commits; branches also let you do this → kind of mean the same things
    - Difference is that branches move when you commit, and tags don't
    - You can pick any commit in the repository, assign it a tag, and that name will not move
    - Useful when dealing with public releases → can know exactly what was in one particular release of software
    - `git tag <tagname>, git tag <tagname> -a` (annotated tag)
- Merging
    - Two lines of development - you want the best versions from both lines of development
    - May have some collisions; How would you resolve them?
        - In a normal repository, every two branches will always have one common ancestor
    - If merge isn't clean, you have to actually think through to make edits to resolve conflicts
    - `diff3` → three way diff to resolve conflicts manually
    - The only branch that moves/merges is the branch that you are on → merge commits and commits work exactly the same with this respect
    - You can merge multiple branches into your branch simultaneously
- Rebasing
    - Competing approach to merging; resulting graph will be a much simpler graph (perfectly linear) → perhaps same or less number of commits

        - History is very simple; makes it appear that you knew exactly what to do from the beginning and it was all perfectly planned out and executed
        - Disadvantage: is not exactly accurate

    `git checkout "mybranch"` – make sure that I'm in the right branch (also changes to branch if created)
    `git rebase "main"` – acts like merge; still have same problem wtih merge conflicts, but resulting graph will be different

    - Rebasing — finds all deltas/changes from shared ancestral node to latest node in "main", and tries to apply them to the last node in "mybranch"
    - `git rebase -i COMMIT` (interactive)
        - Special case of git rebase; will drop you into an editor from COMMIT…HEAD → you can delete any commits that you want
        - Essentially revising history, deleting commits that you don't like → constructs an alternate history for your branch ???
        - Can also hand edit old commits → "edit" instead of "pick"
        - "squash" → can squash one commit into its previous (has both sets of changes) → concatenation of the two commit messages, can modify them as necessary
        - Once you start editing history, you can have the same issue with conflicts
    - Rebasing can make your patch/changes look very nice/polished — not the way you actually wrote it; nice for publishing/easy to understand for other developers
    - With rebase, your job is to end up with a linear history
    - `get-reflog` → will tell you the history of what you did the history, where you were in the repo
    - Once your project is big enough, you have to merge, there is no sense trying to make this beautiful, and you're better off seeing the actual history
    - If project/sub-project is small, makes sense to rebase; you want to make changes aesthetically nice so that it passes committee
    - `git reflog (log version: git log -g)` → use this if you lose your latest commit. HEAD updated everytime you commit, can use to find lost commits
    - `git fsck –full` → find dangling objects, objects not pointed to by other objects
    - `git stash push` → takes all your most recent changes and stuffs it into a temporary storage space so that you can change branches to work on other parts of project → takes snapshot of index so that you can come back to it later
    - `git stash pop` → will bring back the most recent stash; can also go and recover an arbitrary stash later
    - `git bisect` →

        `git bisect start main r12`
        `git bisect run make check` (make check is the test in this case, can use another command)

        - Runs test cases across commit history in binary search, so that you can narrow down which commit caused an issue
        - Finds first bad commit and looks at changes that introduced the bug
        - Cant necessarily know that the result will be the result of the resulting commit - not a magic wand, but there are sub options that the user can set to narrow down the result
        - Bisect does still work with merged paths, but might not work as paths are much more complicated
- Git Internals
    - Porcelein → user interface of git
    - Plumbing → low level commands; more interested in what that plumbing says about the implementation - how that git repo is represented on the lower level system (file system)
    - What's sitting in the file system is the only thing that will persist if plumbing and porcelein crashes → core part
    - `.git` → all of repository's consistant state is stored in this subdirectory
        - branches directory → vestigial part of the git repository

            - there are faster way to store branches in file system than have separate file for each branch; however, bunch of old software was used to having branch files, so they left it there just in case
        - `.git/config`
        - description → vestigial part of git
            - used to have textual representation of what project is for; however, it is more useful to put project descriptions under git control → README file
        - HEAD – tells you where you are currently
        - hooks
            - populated with examples (from git's pov) that you might want to put into the hooks; programs that git will call at specific points during workflow (at checkout, rebase, commit, etc.)
                - Allows you to control to some extent how git behaves
            - precommit → script that will check from all sorts of mistakes that developers might have made during a commit and and prevents it
            - you can bypass hooks during commit if you want → not good style/practice (voluntary); you can change hooks as necessary
            - files under .git are not under git control → to avoid self-referential problems; you can change them any way you like and get rid of them as necessary
        - index – binary data that helps git keep track of where you are and what you plan to do next; if you have a large project and you've done a lot of git adds, your index will be fairly large
        - info – has a directory that has a file exclude → info/exclude (vestigial file) → alternative to your gitignore file; this file is not version controlled, while gitignore is
        - logs – keeps track of logs of where head is and why it got to be where it was
        - objects
            - Commit files contained within subdirectories within .git/objects
            - Subdirectory name is first 2 characters of the commit hash; within that are files that have the remaining 38 characters for the commit ID
            - You can have at most 2^160 commits
        - packed-refs – Representing all of the tags, and all of the branches under git
        - refs
            - has references for all **heads, remotes, and tags**
            - refs/tags is obsolete → git packs tags into packed-refs
- Git Objects
    - Git repository is an object oriented database; one in which everything of interest is considered to be an object - each has attributes, can have other objects as attributes, etc.
    - Each object has a type; has to have some portable way to indentify and locate objects
    - Things that you want the objects to do:
        - Persistance → to reliably survive crashes
        - Stable relationship between key/id for object and contents → deterministic → object contents cannot change
            - Property is important for a version control system; can't change history → don't want people to reach into repository and change past versions of source code
            - Way git comes up with name for object → user specifies complete contents of object (all bytes that go into the commit), git produces name that can be reliably produced → any git installation on the planet will reproduce the same id
            - Name for object is a function of its contents (mathematical function) → hash function
            - One could imagine an alternate git design that was not based on hashing → you give git an object, and it consults an internal directory of names, adds 1 to the last one, etc.
                - Git has to maintain table of all id's its ever given out
                - Overhead gets worse when you have distributed version control system; two different people on two different clones that have duplicate names for same object → disaster

        - With hashing, never be any disagreements over what a name stands for unless you are unlucky
            - If there is some other string that also produces the same hashed output
        - Git uses Sha1 → you give it string of bytes, it hashes and gives output in range 2^{60} - 1
            - Sha1 is no longer secure - it is possible to find a collision
                - its only 0-9, a-f
            - However, uselessness is a deterrent → not an enormous concern
        - Currently, there are plans to migrate to Sha3 (current state of the art)
- `git hash-object -w "filename"`
    - Can store contents of files in git objects; git will not remember metainformation → operating on too low level for most software development
- Object types:
    - Blobs → sequences of bytes
        - Blob contents:

          blob20Ncontents = blob (object type) + size of contents + null byte + contents
        - We take blob contents and we compress them → as result of **compression**, we get what we hope is shorter string than original → doesn't necessarily have to be
        - Typically though, blobs (text) are bigger than the compressed file
    - Trees
        - Designed by Git devs; very strong influence of Linux file system on design of trees → tree is like directory in Linux file system
        - Directory = mapping from file names to files; Tree = mapping from object name to object
        - You can have multiple names pointing to the same object
        - Trees cannot have cycles
            - Name of an object is computed as a hash function of its contents → collision free hash function
            - You can't create a cyclic data structure in git because its collision free; the contents of a child directory cannot (or are very unlikely to) point back to the parent directory
            - Cryptographically secure, collision-resistant hash function is key to the entire structure
        - `git cat-file -t "object-id"` → access type of object
        - `git cat-file -p "tree-id"` → access contents of git tree
        - `git update-index --add --cacheinfo "10664, object-hash, object-name"`
            - 10664 → regular working file "magic number" in git
            - Assigning an object-name mapping at low-level; do not actually have to correspond to any file you've actually created
        - `git write-tree` → creates tree
    - Commits
        - A simpler object than a tree → represents result of a git commit command.
        - Contents:
            - 1. Information abt the commit: timestamps (commit time, authorship time), author name, commiter name, message
            - 2. Pointer to the tree → mapping to some other objects
        - These objects are the ones that everyone sees
        - `git commit-tree "tree-id" "commit message"` → creates a commit object attached to a particular tree
        - Branch → simple mapping from name to commit id; does not have its own tree!!
    - You should not change objects' contents in git. Git assumes that the object's contents match the hashed id; if this is not longer the case, it will not be able to match properly
    - Whenever it appears that git is changing something, it actually creates new objects with changed contents then gives you those id's that are different from the original
        - Even if all the contents of two commits are the same, the commit date will be different causing the resulting id's to differ

            - Very strong protection against this kind of manipulation
- Hashing
    - Developed 70 years ago by IBM Developer
    - You have long char string → convert that into a number within a fixed size (0 … 2^{32} - 1) → you want a unique name for that string that any computer can reliably reproduce

      ```
      unsigned h (char const *s ) {
            unsigned i = 0;
            while ( *s)
                  i = (i << 7) ^ *s++ ^ (i >> 25)
            return i;
      }
      ```

    - You want to have a number that is as random as possible from the input string but still deterministic
    - Smartest ppl on planet are working on hash functions that are A. rare, and B. impractical to arrange even if you try → collision resistant
    - Hash function is collision resistant if given a string, it is impractical to compute a different string with the same hash value
    - Collision resistant hash functions have been proved mathematically to be so → essential to modern cryptography
    - As computers get faster, collision resistant hash functions get attacked; ppl find ways to get collisions even though this was impractical 10 years ago
- Compression
    - Git extensively uses **compression** in repositories (saves space — compressed data takes less space)
        - Another advantage: **saves time?**
            - Takes CPU time to decompress data; takes fewer CPU instructions to read uncompressed bytes, rather than taking them into a separate area and decompressing and reading
            - How then? Saves time by saving space, because you save space, SAVE I/O requests
                - On modern machines → I/O is much slower than CPU time, takes forever to read data from secondary storage
                - Compression saves real time by sacrificing CPU time
        - What about saving *energy*? (novice software developer LOL)
            - On most current machines, you spend energy to save real time
    - zlib comrpession → compression algorithm that git uses to compress objects
    - Based on two ideas to usually generate a smaller representation of the input
        - Huffman coding
        - Dictionary coding

    - Huffman Coding
        - You can come up with expected probabilities for characters you see in the English text, and come up with a shorter way to represent text than the default (8 bits per char)
        - Assumption: know the probability of each symbol where your definition of symbol is up to application → if we define it as a byte, we know the P(byte) for each bytes in the text (each byte value) → come up with an optimal encoding of single bytes into a bitstream
            - You ship the table of probabilities to the recipient before even the message → very cheap to send table, and helps comrpession
        - If you know probabilities, there is an optimal way to assign bit strings to characters that will minimize the total length of the mssage
            - Take all chars that you have, find all probabilities
            - Build a tree → find the leaves with lowest probabilities, have them be child to a node with shared probability (probability of being either of the least likely chars) → continue building up
            - Final tree → the path from the root down to any of the leaves will be a bit stream
        - This method works very well for long strings; for short strings, effort you take in building info abt tree will be greater than effort you save in just storing/saving the original contents

    - Problem w/ Huffman Coding = Assumption that you know the probability of each symbol in advance

- Adaptive Compression w/ Huffman Coding
    - How do you come up with a compression algorithm that works with any kind of text?
    - One option = not use a fixed probability table, used a varying P table depending on the text → Adaptive compression → Adapting to the input
    - When sender starts compressing → starts by saying knows nothing about the distribution; all symbols equally likely
        - Each byte P = 1/256
        - **Initial Huffman tree:** Each leaf is same distance from the root, each leaf will have a bit encoding that is 8 bits long → perfectly balanced
            - Each leaf will be exact same distance from the root, each bit will have a 8-bit long representation → no compression at all
    - First byte is sent → Both sender and recipient re-evaluate their trees assuming that this first byte is more commonly occuring in the text → will get closer to the root of the tree (shorter encoding)
    - Next byte will be slightly compressed compared to the first → recipient, since has own updated copy of tree, will be able to interpret
    - Only 1 way communication: sender → recipient
    - No table is sent from the sender to recipient directly; both have an original uniform table (very easy to construct) → communicate what the table should look like simply by sending data → as long as they update tables in sync, recipient is always able to interpret compressed data
        - Data sent is perhaps not compressed as effectively as it would be if you calculated all of the probabilities ahead of time

- Dictionary Compression
    - Huffman's proof of optimality was right under assumption: your compression algorithm is mapping bytes to bitstreams → if you change compression algorithm, you can come up with more better results
    - If you transmit English text → sender and recipient share a dictionary → map numbers to words; recipient is given the keys, able to consult dictionary to reproduce original uncompressed text
    - If you're words are long, doesn't matter → still compressed down to a small number of bytes → ratio is even better than Huffman coding
        - Best huffman coding can do is compress a byte down to 1 bit
    - **How do you come up with shared dictionary?**

- Adaptive Dictionary Compression
    - Combines ideas from adaptive compression w/ Huffman coding → dictionary built up dynamically by the recipient and sender as the message is sent
    - Dictionaries contain only words that we have seen so far in the input
    - Sender looks at next set of unsent data → sees if there is an instance of the byte-string already sent → sends a message toe the recipient (distance from current position in data to beginning of earlier instance of the word + length of the word) → recipient is able to perform the lookup
    - If words get repeated a lot, you are able to compress very well
    - This approach = much more CPU intensive
    - It is impractical to save all of what has already been sent - esp if you are dealing with large volumes of data → gets around this by saving fixed-size window maintained by both sender and recipient (both agree on size)

- Combining — Dictionary compression gives you a set of symbols; apply Huffman coding to those symbols → applying one comrpession algorithm on top of another
- Measures of compression quality:
    - Shrinkage → size of compressed data vs. original data; usually depends on the kind of data that you are sending

        - RAM (sender and recipient) → compression algorithms have to work on IoT, etc.
        - CPU time

8. C Programming & Debugging

- White House issued press release → told devs to stop using C - facing real problems with infrastructure w/ code that is too low level (not memory safe, dump core or worse), vulnerable to Russian attacks, etc.
    - use memory safe languages - no matter what you write, program cannot crash due to subscript error, or bad pointer, or accessing memory that it shouldn't
        - cannot dereference a nullptr, cannot access freed storage, etc.
    - suggested that we switch to Rust; however, they mentioned that Rust is not a proven language for avionics (aerospace) applications
        - Rust is memory safe, unless you use keyword "unsafe"
    - Context: talking about garbage collection (automated part of system that will reclaim storage no longer in use) → they reccomend Rust is applications where garbage collection is prohibited
        - Garbage collector will cause hiccups in execution; program will go catatonic while its garbage collecting (e.g plane crashes LOL)
- Department of Defense said the same thing 45 years ago - memory unsafe languages are dangerous
    - 1980: DoD launched Ada; memory safe, unless you use keyword "unsafe"
        - People started using Ada, but found it was too much pain to run safe code
        - Need to break rules if you're doing avionics, getting at low-level registers; wrote unsafe all the time → just switched back to C
- **Idea: writing code that is close to the bare machine or OS kernel**

- C vs. C++
    - C++ has classes, C does not; if you want to have classes in C, write them yourself (allocate storage, implement them yourself, etc.) → lot of people don't want to (avoid overhead, etc.)
    - Dont have …
        - Classes
            - Polymorphism (when you call a function, you know exactly what its going to do, don't have to worry about polymorphs for it)
            - encapsulation (at least not much)
            - inheritance or multiple inheritance
        - Structs and static data members, functions
        - Namespaces → control over names is very simple; C has static names vs. external names
            - Static names only available in the source module they are defined in, 1 .c file; External names are available anywhere in the program
        - Overloading → only 1 implementation of a function
        - Exception handling → no try, catch, except; the way you indicate to caller that something bad happened is you return a value from your function
        - Builtin memory allocation → there are no new(), delete() operator
            - If you want to allocate memory, there are small set of standard function that allocate bunch of bytes, you can use them any way that you want
            - malloc(), free() → don't know types of objects they are allocating; all they know is size (byte)
                - void * malloc (size_t nbytes) → you tell it how many bytes you want, returns a pointer of type void * (generic pointer)
                    - Any pointer type can be converted to void *; converse is also true
                    - malloc doesn't care of the type of object you are creating; one of the reason why C is memory unsafe (no type checking - could result in errors)
            - Not a type safe system the same way that it is in C++; can do more things with resulting memory
        - Cin, Cout

            - You have standard functions: printf(), scanf(), … putchar() → lower level than what you get with formatting primitives of C++
- Compilation & Execution Environments
    - `gcc -E foo.c`
        - Means "just run the preprocesser"; don't do anything other than textual expansion of preproccessor macros
        - gcc -E foo.c > foo.i → By convention, preproccer output ends in .i → takes you one step through the compilation process
        - foo.i → C source file; just doesn't use macros → can find an fix bugs just by doing this; see if macros expand into something that you didn't want/expect
            - A lot of the art of debugging is not using the debugger; one of the worst ways to find and fix bugs is to use the debugger
    - `gcc sourceCode.c -o outexe`
        - gcc: This is the GNU Compiler Collection, specifically the C compiler. It's used to compile C source code into executable programs.
        - sourceCode.c: This is the input file, a C source code file that you want to compile. The .c extension indicates it's a C language source file.
        - -o: This is an option flag that specifies the output file name.
        - outexe: This is the name of the output executable file that will be created after successful compilation.
    - `gcc -S foo.i > foo.asm`
        - Takes the preprocessed output and generates Assembly language — textual representation of machine code —, suitable for the machine that you are targeting
        - By default gcc assumes that you are targeting machine you are currently running on; could be running gcc on an x86/64, and targeting a different processor
            - `gcc -m32 foo.i` → targets x86 (32 bit architecture, rather than 64 bit architecture)
        - Foo.s → just a text file that tells reader what the machine instructions program wants to generate
    - `gcc -c foo.asm`
        - Runs the assembly, generates .o file (object file) — has machine instructions except in binary format (machine code)
        - His example: Won't run either, because call instructions have been issued to place where we don't know what it is yet — used a function that is external
    - `gcc foo.o libc.a -o foo` (libc.a → contains all your C libraries, fills in the blanks)
        - Generates an executable file → Linking; You see all instructions from before, but fills in the blanks → machine instructions would actually work
        - CPU cannot execute instructions sitting in a file, can only execute functions sitting in RAM
    - `./foo`
        - Runs the program; puts contents of executable, puts it into RAM, and sets instruction pointer (rip) to go there, and FINALLY instructions run
- When you're debugging, any of these steps can go wrong.
    - Common problem: External functions required were not in the C library you provided/wanted → Linking step will fail; Message won't come during compilation stage, preprocessing stage, will come from linking phase
        - Need to know which phase is generating which error message
    - For convenience: `gcc foo.c -o foo`
        - Does all of the above steps; doesn't even show the temporary files → as long as everything is working, much more convenient
        - When things stop working, can be helpful to break things into components
- Which step are bugs most found in?
    - `-c stage` (generating .o file) → most complex, can do the most analysis of your program
        - If you use fancier flags, this can be very good at spotting bugs
        - Static analysis → compiler is looking at your program before it runs and finding bugs
        - Dynamic checking → add separate flags to compiler; won't find bugs, but will put in extra checking code that is assembled and linked, will slow down program

        - Benefit of slow down: report bugs faster, program will crash more often (crash reliably rather than crashing erratically)
  - Bug Reports → someone yells at you (lol)

- Heap Allocation vs Stack Allocation
  - Heap-allocated variables are:
    - Dynamically created at runtime using functions like malloc(), calloc(), or realloc()
    - Manually managed, requiring explicit deallocation using free() to prevent memory leaks
    - Accessible globally within the program
    - Not limited by size constraints, allowing for larger data structures
    - Slower to access compared to stack variables
    - Prone to memory fragmentation over time
    - Resizable using functions like realloc()
  - Stack-allocated variables are:
    - Automatically created and destroyed within their scope
    - Very fast to access due to efficient CPU management
    - Limited to local variables within functions or blocks
    - Restricted in size, with limits depending on the operating system
    - Unable to be resized once created
    - Efficiently managed by the CPU, preventing memory fragmentation
    - Automatically deallocated when they go out of scope, ensuring deterministic memory management
- How to Debug?
  1. **Best strategy: Don't debug!** Degugging is an inefficient way to make your program more reliable (find and fix bugs).
     - In badly run projects, debugging can take more than half time
     - (a) Need to have better dev practices so you don't get sucked into debugging; Prevent bugs from happening in the first place
     - (b) When bugs do occur, design system so that they are **easy to find**
     - Static and dynamic checking are techniques to attack (a) and (b)
  2. Test cases
     - Good test cases save you a lot of development time in most cases
     - TDD: Test Driven Development (school of thought in software development) — test cases are so important that you should write them first before you write the code, before you know what code you will write (don't have the spec yet)
       - Let's you see the code you plan to write from its users' POV; very big deal in software development
     - Once you have test cases, need to maintain them; as you add feature/fixes, test cases should evolve in a similar way (e.g. add tests for new features, test whether fixed bugs are still present)
  3. Use a better platform (Rust, Java, Python, etc.; not C or C++)
     - Java is memory safe, but is not integer overflow safe; C# is probably similar
       - If you have a subscript calculation that is computed by an integer arithmetic that overflowed, you get completely the wrong subscript → if out of range of array, it'll hit an error, if not unpredictable behavior
  4. Defensive Programming
     - Gets its name from concept of defensive driving — when driving, assume that everyone else is texting or drunk or both; won't get to destination quite as fast, but will reliably get there
     - You assume that other parts of the code are busted
     - Defensive techniques:
       A. Runtime checking of your own
       B. Assertions
          - `# include <assert.h>  .... assert` (any expression); if expression is false, program will dump core immediately — automated way of doing A

- Expression should be side-effect free (e.g. printing); if it does, program will run differently with debugging vs. without debugging (very bad)
- `gcc -DNDEBUG`; compiles with debugging disabled, turns assertion off

C. Exception handling
- In most languages this is built in; C does have it → #include <setjmp.h>; in C not considered good style

D. Traces & logs
- `$ strace` → built in command in linux; can run any program under strace, will tell you all system calls (low level calls to OS) that the program does as it runs
    - Getting a log/trace of every action of a certain flavor (lot of options)
- May want to do this even when your program works; can help you debug code 2-3 months later – particularly useful for security-related purposes

E. Checkpoint + restart
- Every now and then, you call a function checkpoint() that you write – takes entire state of your program (everything in RAM) and saves it (either prints it out or saves it in binary form)
    - Application dependent; have to find every variable in your program and print out its value into binary file
    - Have to decide which parts of your program are part of the state vs. can be recomputed (can be forgotten)
- `restart(f)` – takes the state of your program out of the file, and puts it into your program so that your program can start running
- If your program crashes, don't need to start from the beginning; can just resume from the most recent checkpoint, and give it slightly different flag/input data, etc.
    - Useful for long running programs

F. Barricades
- You have a big, messy program that talks to outside world (written in C) – being attacked constantly; attackers trying to send in bad data that will confuse program
- Barricade –
    - Focus on core (important data/functionality) & clean it up (relatively small amount of code)
        - Within the cleaned up area, assume you have reliable data structures – everything is organized, doesn't have bad data, etc.
    - Between core and messy parts of program, you have a **BARRICADE** → an API that lets data in, and exports data out; barricade checks the data, doesn't trust it
        - Makes sure that every bit that comes into the core is clean
- With time, goal is to migrate the barricade outwards; get more and more of our code to be clean and reliable and less messy
- Similar to a "software firewall"; firewall is more of a hardware technique – not perfect but can help cut down on number of bugs

G. Interpreters
- Assumption: not just have bad data, but have bad code; either written poorly or b/c someone is attacking the system
    - Standard example: Browser (e.g. Chrome)
- Program has a code barricade around it; inserted code is run under the control of the interpreter, rather than run on the bare machine
- At any point, the interpreter can stop code from running or send wrong results to the code to prevent it from attacking the core system
    - Ex. One common attack vector: asking what time it is, to expose underlying details of the machine; interpreter lies abt the time (adds noise to the resulting time)
- Takes a lot of work to write; heavyweight

H. Virtual Machines

            - Instead of interpeting source code, you interpret machine code; a program that pretends to be a chip, but isn't really
            - You can scale up from interpeter solutions; can take entire suites of applications and run them within a virtual machine, so they can't infect the rest of your system
    - Don't guess at random →
        - When debugging, tempting to to assume that a bug is in one place, and fix that with the hope of correcting the bug
        - Guessing works only if the program is very small; doesn't scale to large programs.
        - Often a bug is not confined to a single line, but is a combination of various parts of your program
    - You want a more systematic approach:
        - **1. Stabilize the failure** → Many bugs are evanescent; program may work at times and break at other times, it may be input dependent
            - Need to find a way to get around sources of randomization; make sure that you can reproduce the bug
            - Bug report may come from a user (may not know how to file bug report effectively → may be ambiguous); you have to reproduce the user's bug
        - 2. Locate the failure's cause or source
            - Line number in the error message (where the failure is observed), is not necessarily the actual location of the bug; other parts of the program didn't do its job
    - Terminology:
        - Failure → if program behavior is wrong and a user notices it; user-visible misbehavior
        - Fault → latent bug in program such that if that part of the program is executed in the WRONG ENVIRONMENT, will get a failure; will cause failure in some circumstances
        - Error → Mental mistake on the part of the software developer; you write the wrong code
            - Error may not necessarily cause a fault (incorrect/incomplete comments); fault may not cause an error (part of the code is never executed, or never hits the requisite circumstances)
        - Bug → no standard; Eggert tends to think of bugs as faults
- GDB
    - Intro to GDB
        - Why? It's a low-level debugger; we should learn low-level debugging as engineering students. GDB is also the oldest debugger still in widespread use
            - Has lots of features, since it's been around a long time
        - Process – puts itself in control of your process; you want 2 different processes → progam being debugged should be unable to make GDB crash – insulates GDB from your program
            - We don't want the converse to be true; GDB should be controlling your program, program should not be insulated from it (1-way insulation)
            - GDB can:
                - Stop your process; it's still there, still has an instruction pointer, but is not executing
                - Read your process's memory; any variable, machine code of any function, figure out which parat of addresses are valid, etc.
                - Read program's registers; things that are sitting in the CPU
                    - One of these is the instruction pointer (next instruction your program is going to run); can set to be current value - 1, or any other spot in your code to execute
                - Write all of the above. Process is frozen, and GDB can change values of items as necessary
                - Continue execution of your process
        - It's possible on some systems to have a different environment
            - May have debugger module and debugged module → part of 1 program; also works, but less reliable → if module you are debugging has serious bugs that causes it to interfere with ex. print buffers, it won't work properly

- If you use GDB's full power, it can change your program to something else while its running
- Get Started
  - `gcc -g` → In addition to everything else, adds extra info to the .o files and executables; these don't help program at all, but help debugger – debugging information
    - Doesn't change the generated code/machine instrucitons
    - What you see in memory will be the same; executable file will be bigger than you'd expect (will take up more space)
    - Debugging info – for each function, what is the first and last machine address? where functions are/global variables are? what are their types?
    - If you don't use -g, will still be possible to debug, but is much more of a hassle
  - `gcc -g3` → will output more debugging information
  - `-O` → If you enable optimization, will hinder debugging; compiler will throw out useful things like local variables
    - `-O0` → lowest level of optimization; really slow code, but is easier to debug
      - Problem: when you change the machine code that you are generating (that's what -o option does), this might change program's behavior; program behaves 1 way with optimization, and another way without (one with bugs, one without)
      - If you temporarily disable optimization for debugging, you are essentially debugging a different program
    - `-Og` → optimize for debuggability in addition to speed
- GDB Commands
  - `gdb <program>` →
    - GDB knows you want to run the program, but hasn't run it yet; GDB process has started, but your process has not
    - Have to determine what environment you want the program to run under; configure debugging environment
      - Configuration is one of the biggest sources of bugs/problems
  - `(gdb) set cwd /a/b` →
    - when the program starts, it won't be running under the original directory, but under a new dir; can be helpful if test cases are in the other directory
  - `(gdb) set env PATH /usr/bin/my/bin`
    - Environment variable setting will apply to the debugged program (special environment, different from parent environment), not to GDB
  - `(gdb) set disable-randomization off/on`
    - Default is on → randomizaton of memory space that the program runs in → if the program has a faulty pointer or some such, can help with security
    - However, can make it more challenging to stabilize behavior because runs in differences spaces
  - `(gdb) run “arguments”`
    - Starts up a process that you specified prior, with a number of arguments, under gdb control
  - `(gdb) attach 53196`
    - the last number is a process ID; it has to be a process id that is already running in your system → GDB will take your process and takes over that already running process
    - Process is frozen now
    - The process id must be your process (has to have your user id); cannot attach to any process on the system
  - `(gdb) detach`
    - Lets the process continue as if GDB had never touched it
  - `(gdb) bt` → backtrace — how you got to where you are right now
  - `(gdb) p` → can put any expression after p and gdb will print out value of that expression
    - When printing variables, prints varaibles that are relevant in the current context
    - gdb knows variable data types, knows how to do arithmetic, etc.

- Can also print out values of user defined functions → not limited to just what's built into the language; you can call your own functions → even if these functions print out their own material
  - Arranges for the subsidiary process to actually call the function; outputs whatever side effects the function produces
- `(gdb) p/x n` → print in hexadecimal format
- `(gdb) p x=y` → assigns y to x then prints the output; changes the state of the program

- `(gdb) up` → changes context; no longer looking at the local variables of the current function, looking at the caller's local variables
  - `(gdb) down` → goes in the opposite direction

- Extending GDB
  - Can add commands to GDB to make it do extra stuff: print out data structures of your own design, access builtins, etc.

```
define pl
      print *(long *) $arg1
end
```

`(gdb) pl x` → treats x as a pointer to a long, and print out the long that it points at

  - When you start GDB, it configures itself by reading file .gdbinit in the current directory
  - There is also a python interface for gdb, and you can write python scripts to modify gdb; can also program it with Lisp
  - Debugging tools need to be configurable and programmable
- Remote Debugging
  - Gdb process running on one machine can access processes running on a different machines; can be over physical wires or over the internet
  - These two machines may have different architectures/machine instructions → gdb will have to know the target architecture, so all instructions executed on the remote machine match
  - (gdb) target …
- Breakpoints

```
(gdb) b core.c 97 // breakpoint
(gdb) run // run w args
(gdb) cont
```

  - Assuming that you've compiled so that gdb knows where the machine code for line 97
  - gdb looks at machine code and finds first struction of line 97 of core.c → can arrange such that whenever program gets to line 97, it stops
  - Only changing the instructions, not changing the data
  - Will show that the program has stoped, gdb regains control → can examine the state of the program at the breakpoint
  - Then the user can make the program continue as normal → unless you are in a loop, it will bring back the breakpoint
  - f the program is compiled using high level of optimization, program compiled mutliple ways by the optimizer → multiple spots in the machine code would correspond to the breakpoint

  - `(gdb) info break` → gets info for all breakpoints currently active
  - `(gdb) info regs` → gets info for commonly used machine registers

  - `(gdb) step` → single stepping through the source code; keep running until the source code line changes and then stop; temporary breakpoint in one spot then remove it

- `(gdb) stepi` → just execute a single machine instruction → typically several stepi's = a single step instruction
- `(gdb) disas` → disassemble; show machine instructions
- `(gdb) next` → Keep running until the line number changes in the current function; What is the difference between next and step??
- `(gdb) fin` → Keep running until the current function returns; tells you value current function returns

- Reverse Execution
  - `(gdb) rc` → reverse continue
    - Runs your program backwards; undo's assignments/changes made by your program until you hit a breakpoint
  - Have to give GDB a particular flag if you intend to use reverse continue; a specific mode → will make GDB single step the program for every instruction
  - Every instruction that has a side effect → stores old contents of program memory/registers in RAM (internal log of all values that are destroyed by assignments) → stored in GDB's own memory, not the program's memory
  - Downside: this will slow down the program significantly
  - `(gdb) checkpoint` → will save the entire state of your program into its memory; will provide you a checkpoint number
    - `(gdb) restart "checkpointnumber"` → can run the program from the checkpoint defined previously
  - `(gdb) watch "expression"` → no function calls/side effects in the expression; can create a watch point → stop the program when the expression value changes
    - Watching the data to see when it changes
    - Can watch up to 4 variables at the same time without any change/decrease in speed

- Performance Optimization
  - Most compilers have optimization options:
    - `gcc -O1, O2, etc.` → gcc gets slower, but the hope is that your program gets faster/smaller, etc. Program also gets harder to debug; machine code is also bigger (more space → may blow instruction cache)
  - `void error () __attribute__((cold))`
    - Cold → advice to compiler on how to use instruction cache more effectively; indicates that function is rarely called, and calls to cold code partitioned in instruction coche different from hot code (frequently called code)
  - `_Noreturn` → performance advice; don't have to use a call instruction, use jump instruction (more efficient) → use less stack space, etc.
  - Problem with hot/cold
    - More work for developers
    - Sometimes developers will make mistakes - give wrong advice to the compiler
  - Profiling
    - You tell compiler to generate machine code that will count the number of times each instruction is executed
      - `gcc --coverage` → generages extra code to count # of times each instruction is executed
      - At end of program, can print out a profile, statistics graph
    - Find hotspots for program → feed it back into gcc to identify hot and cold information
    - This will be a profile of just your own program (perhaps on your own test cases) → may be different int the real world → relies on your profile being accurate
    - Will slow down your program and make it behave differently
  - Whole program optimization
    - `gcc -flto` (link time optimization)

        - When you do ordinary compiles, you compile individual chunks of program → whole program optimization, optimizes all of the source code linked together at once instead of just relying on individual module optimization
        - Can take a long time, and is occasionally impractical
        - Since these optimization is so extensive, may run into compiler bugs
- Dynamic (Runtime) Analysis
    - `gcc -fstack-protector` → causes gcc to geenrate extra code for functions it determines …
        - I do not understand what stack protectors are and what they do??
        - Error: writing past the end of the array and crashing stack
        - Return-oriented programming (ROP) → attempts to take over return addresses, have them point to other parts of program you didn't intend
            - If they can control enough return addresses, they can build a Turing machine (that can do whatever they want)
    - `gcc -mshstk` → generates for a machine that supports shadow stack protection
        - Stack protector slows down execution, you need to add extra instructions
        - Let's make hardware support to make function calls more reliable
        - Ordinary stack → newer AMD chips have shadow stack (contains return addresses, copy of what's in main stack)
        - Key difference between stacks → shadow stack is not writable except by the call instruction
        - Call instructions will push return address onto both stacks, and the return instruction will pop addresses from both stacks and confirm that they match → Security!
        - Shadow stack is a relatively new feature → older Intel and AMD chips don't support it
        - If return address is not matched in the shadow stack → program crashes

```
# include <stackdint.h> //C23 - not available in older sytems
      if (ckdadd (x+5, y*3, &z)
error
```

    - Let's you do arithmetic, but checks for integer overflow
    - `-fsanitize=address` → generate more machine code to catch most memory violations
        - Not 100% safe, but good enough for most applications
        - `-fsanitize=undefine` → every instance of undefined behavior (signed integer overflow, calling unreachable(), etc.) causes the program to crash reliably instead of random-ish behavior
        - `-fsanitize=leak` → catches memory leaks
        - `-fsanitize=thread` → attemps to catch race conditions; 2 threads, 1's writing to a piece of memory, 1's reading → causes a race
            - Adds extra code to try to detect the race; can slow down your program significantly, but can catch race conditions
            - Either have it quit, or have it continue → decent performance
    - Downside of these components is that you have to recompile; generate different executables
    - `valgrind`
        - Attemps to take your executable, instead of compiling a different program
        - Instead of letting your program run at full speed, it single steps through your program; when it encounters an instruction that might cause problems, checks to see if there were issues
        - Interprets the program, and checks for certain classes of runtime errors → most likely slower, but you don't need the source code (can run other ppl's executables w/out source code)
        - Doesn't catch as many errors as you would from the gcc or clang flags, but you don't need to recompile your program
- Static (Compile-time) Checking
    - You can do it yourself. – `static.assert (INT_MAX < LONG_MAX)`
    - Works for all runs of the program; constraint is that it only works for constant expressions → equations that the compiler can know before the program begins to run → can be known statically
    - `gcc -W_____` - Many options that allow you to access benefits of static checking without changing source code (finding compile time errors)

        - `gcc -Wall` → turn on all static checking that gcc devs think is good for everybody
            - `gcc -Wcomment` → warns if comments aren't handled properly
            - `gcc -Wparentheses` → warns if there are implied parentheses that should be outlined to prevent ambiguity (for arithmetic expressions or logical operators for example)
            - `gcc -Waddress` → if addresses are used in context where they are almost certainly not intended
            - `gcc -Wstrict-aliasing` (can disable this: `gcc -Wall -Wno-strict-aliasing`)
        - `gcc -Wextra`
            - `gcc -Wtype-limits` → if checking is done beyond a variable type's limits, will give warnings
    - `gcc -fanalyzer`
        - Looks through all paths of your program → statically; looking for common errors
        - Like -Wmaybe-initialized, but its interprocedural → maybe-initialized only looks at 1 function
        - fanalyzer looks through all functions and sees if there is a path through any one of those function calls that will leave a variable uninitialized

9. Software Security

- Your responsibility, the customer will expect you to handle this without prompting
- We need: 1. a security model (what you're defending), and 2. a threat model (who's attacking and how)
- What do we mean by security?
    - Confidentiality/Privacy → information inside the system shouldn't leak out into places it shouldn't go
    - Integrity → someone is going in an tampering with information inside system; shouldn't be able to
    - Availability → whether the system is up at all; provide a sservice even if the attacker is trying to bring the system down
- What do we need to identify in security model
    - Assets → what is important to protect
    - Vulnerabilities
- Threat modeling + classifications:
    - Insiders
    - Social engineering
    - Network attacks (e.g. phishing — isn't this more of a social engineering attack? —, denial of service DoS, buffer overruns, etc.)
- Attacks
    - Phishing - you exploit a bug in client program (browser, mail client, etc.), you craft a url/image to exploit the bug, then take over the client's computer
    - DoS → attackers set of up bunch of clients that continually set up bogus requests to your server → swamp the server and they can't do any legitimate work → pay me money or I'll keep doing it
    - Cross Site Scripting (XSS) → JavaScript from one site constructs requests inside the browser, tells the browser I'd like to perform this request on the web from another server, gets responses back
        - One website attacking another website through your browser
    - Prototype Pollution → one website attacks another one, by attacking prototypes of JavaScript programming language
    - Device Attacks → attacker supplies hardware to victim; victim trusts hardware to behave as it should, but it behaves differently

- Network Attacks
    1. Broken access control →
        - Give you access control via a unique, hard-to-guess url for user; any attacker can get the url and access the website as you
        - Can be done via cookies, JWT, etc. Very common mistake → I think we made it in our project
    2. Crypto Failures
        - HTTP not HTTPS
        - Using weak cryptophic algorithms, that have been superceded by new technology/techniques for breaking them

        - Not validating certificates → if on browser side you don't validate certificates, can be a disaster
    2. Injection
        - Trust user data to be ordinary user data (English text, etc.); can cause your program to be the wrong thing
        - SQL Injection
    3. Insecure Design
    4. Security Misconfiguration
        - Might leave maintenance ports open; often not as well secured
        - Leaving in default accounts + passwords
    5. Outdated and vulnerable components
    6. Identification and authentication Failures
        - Allowing easy passwords; is it easy to get a list of usernames? allow password guessing?
    7. Software and data integrity failures
        - Data/programs that have been tampered with; should be doing integrity checks - on the developer
        - Supply chain verification - make sure software/modules you import are not intentionally defective
    8. Security logging & monitering failures
    9. Server-side Request Forgery SSRF
        - Often combined with injection; trick web server to send request to another server that it wouldn't otherwise send

- Cross Site Scripting (XSS)
    - You have a client → talks to server → server sends back to client JS code
    - When client runs the code, it will send off a request to some other server → that request and response is not initiated by the end-user
    - A lot of the time, this is totally inoccuous; there are situations where it can be dangerous
    - Two servers not maintained by the same group; run by different companies → one is trying to attack/modify information maintained by the other
    - Same Origin Policy (SOP) → a browser is only going to send off requests to the origin server; doens't allow any XSS for anything complicated or interesting
        - Same origin: has to have the same protocol and same port number
    - CORS Cross Origin Resource Sharing → resource for undoing the SOP for trusted servers to collaborate; normally browsers insist on same origin, notifies server of the origin of request (in https header) → victim server will know who wants to use, and can verify yes/no
    - Trying to set up who's authorized to use what server/XSS

- Prototype Pollution
    - Bug in javascript program, which enables attacker to call function with arguments
      ```
      f ("__proto__", "cmd", "rm -rf") -> if you accept user inputs into the JS function
      ```
    - At the very top level of JS, there is root prototype – you clone it when you create any other object; changing cmd attribute allows you to run commands and alter behavior of JS code
- Testing Security
    - Different from testing functionality; ordinarily, if you look at parts of your program, you assume that bugs are random-ish (ppl are trying to write code well, but make errors)
    - W/ security, bad guys are focusing on the bugs; owness on testing is significant
    - Static analysis is a bigger deal here; can prove that certain classes of bugs cannot possibly occur → if attackers have access to ur source code, they'll be using the same tools that you are
    - Testing need to look for breaking abstraction boundaries
        - Reads password into buffer, checks it (if ok), keeps going → need to clear the buffer → clear out registers
            - Problems: Buffer overflow (C/C++), not clearing the buffer
            - Information in the buffer may be sitting in the contents of your register (loaded to do comparing)
        - Password laid out onto two memory pages

        - Two areas of memory that each separately are either in RAM or can be paged out to swap device, etc.
        - Machine pretends its in RAM; if you access a page and swapped out, machine will swap it back in and give you access to the page
    - Spectre/Meltdown attacks
        - Attack your caches; can happen faster than password guessing attack → can let javascript inspect contents of your browser's memory and seeing how fast they respond to invalid memory accesses
        - Downloads malicious JS that contains subscript errors, times how fast subscript errors are trapped and reported to that code → used to find contents of memory in some other completely unrelated part of browsers
        - Defense: not accurately reporting what time it is
- Risk Assessment
    - You can't defend against every attack. You can assess risk of various set of attacks; devote resources in right areas
    - Common Problem Areas:
        - Lack of Training for Developers
        - Neglecting or Not Negotiating/Assessing Risks → When setting up requirements; need to talk security issues with users from Day 1
        - No Threat Model → don't have a good idea which architecture would be good to defend against those kinds of threats
        - No design awareness of security
        - Coding security vulnerabilities (prototype pollution, SQL injection, DoS, repudiation, buffer overflows, etc.)
        - No static checking, no dynamic checking, no fuzz testing, no penetration testing
            - Fuzz testing — program will attempt to generate random data and test program with that random data, will interpret your program and determine branches taken
            - Penetration analysis - you hire someone to break into your application
        - No/insecure default configuration - program isn't secure out of the box; when the user logs out, doesn't really log out (trust OS to clean out RAM more than JS interpreter), wrong ports enabled
        - Maintenance → fixing bugs after deployment; forget to repeat the above
- Trusted Computing Base
    - Helpful to come up with a core part of the system → most of application is not trusted; other parts will not be able to write/read files unless TCB verifies this
    - Putting all security eggs in one basket; there are opposing schools of thought that say to distribute important info → harder to do, easier for ppl starting out

Git Internals - Git Objects:
→ key-value data storage

<u>git hash-object</u>: takes data, stores in .git/objects, and gives unique key referring to data object

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

1. **Just git hash-object would take the content and return the unique key that would be used to store in Git database**
2. **-w writes the object to the database and doesn't simply return the key**
3. **--stdin tells has-object to get the content to be processed from stdin**
4. **output is 40-character checksum hash / SHA-1 hash**

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

1. first 2 characters of “d6” are subdirectory
2. next 38 characters is filename

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

1. git cat-file is used to examine content in object database
2. -p instructs to figure out type of content and display correctly

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

1. redirection to test.txt
2. save contents of test.txt into database

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

1. same shit as above, but for new content

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

1. object database has both versions of the new file

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
 |||| or the second version:
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

1. Basically, you can retrieve the first or second version if you hash and save contents into the database

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

1. git cat-file -t tells us you the object type
2. In this case, it's a blob!

Tree objects:

- One tree object has one/more entries, each of which is a SHA-1 hash of a blob or subtree

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859     README
100644 blob 8f94139338f9404f26296befa88755fc2598c289     Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0     lib
```

1. **master^{tree} tree object: specifies the tree object pointed to by the last commit on the master branch**
2. **lib subdirectory: pointer to another tree, not a blob**

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b     simplegit.rb
```

1. **retrieve whatever hash you put, in this case, it was a tree that contains a blob called simplegit.rb**

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

1.

C:

Core Language Philosophy:

1. functions and data separate instead of object-oriented
2. closer to hardware experience
3. no auto cleanup mechanisms

1. No classes or objects
    a. no encapsulation of data and methods
        i. encapsulation: dundling data together with methods that operate on that data into a single unit (usually a class in OOP), allowing for you to hide the internal details of an object and exposing functionality through public methods
    b. no function overloading
        i. every function needs a unique name
    c. no references
        i. only pointers
        ii. parameters passing is always values or pointers
    d. no templates
        i. generic programming uses void pointers, or macros
    e. no standard template library
        i. so no vector, list, map
    f. must implement data structures manually or use external libraries
    g. no exception handling
        i. error handling uses return values
        ii. must check function returns explicitly
    h. no bool type
        i. traditionally used 0 for false and non zero for true

```
#include <stdio.h> // input/output operations
#include <stdlib.h> // memory management, random numbers
# include <string.h> // string operations
```

```
#include <math.h> // math functions
# include <ctype.h> // character type checks
# include <time.h> // time/date utilities
```

printf syntax:

```
printf(“Age: %d, Height: %.1f, Grade: %c\n”, age, height, grade); // each include text and format specifier for each respective variable
```

%d	Signed integer	42
%f	Floating-point number	3.14
%c	Single character	A
%s	String (character array)	Hello
%p	Pointer address	0x7ffeefbcbad0
%X	Hexadecimal (uppercase)	FF
%o	Octal	377
%u	Unsigned integer	42
%e	Scientific notation (lowercase e)	3.14e+01
%g	Shortest representation of %f or %e	3.14

ex:

```
printf("Integer: %d\n", 42); // Basic integer
printf("Float: %.2f\n", 3.14159); // Float with 2 decimal places
printf("String: %s\n", "Hello"); // String
printf("Character: %c\n", 'A'); // Single character
printf("Pointer: %p\n", (void*)ptr); // Pointer address
printf("Hex: 0x%X\n", 255); // Hexadecimal
printf("Octal: %o\n", 64); // Octal
```

```
int num;
char str[100];
float fnum;
```

```
scanf("%[^\n]s", str); // ^ → is not operator, \n → new line, % → placeholder that indicates format specifier
```
→ so together, its reading all of str up until it encounters a new line

```
fgets(str, sizeof(str), stdin); // Safer string input
```
→ str: pointer to the character array where the input will be stored
→ sizeof(str): specifies the max number of characters to read, including the null
fgets reads characters from the input stream and stores them in the str buffer

- keeps reading until newline characters is encountered
- sizeof(str) - 1 characters have been reached
- end of file is reached

Memory management:

```
void* malloc(size_t size); // malloc reserves a certain amount of space with no organization
void* calloc(size_t num, size_t size); // same as malloc, but this time space is allocated for memory and also initialized to zero
void* realloc(void* ptr, size_t size); // resize the amount of memory allocated for a given space
void free(void* ptr);  // freeing memory
```

- remember: pointer = address of integer

```
int* p1 = (int*)malloc(sizeof(int)) // malloc reserves a certain amount of space, and we typecast the void*, a generic pointer to convert the pointer to an integer (int*)
if (p1 == NULL) return 1;

int* p2 = (int*)malloc(n*sizeof(int));
if (p2 == NULL) return 1;

struct Person* p3 = (struct Person*)malloc(sizeof(struct Person));
if (p3 == NULL) return 1;
```

For matrices, you can either have contiguous memory accesses or use an array of pointers, meaning that each pointer will point to another pointer, with each pointer representing a row of the matrix

```
int* matrix1 = (int*)malloc(rows * cols * sizeof(int)); // Access: matrix1[i * cols + j

for (int i = 0; i < rows; i++) {
matrix2[i] = (int*)malloc(cols * sizeof(int));
        if (matrix2[i] == NULL) {
                // Handle error - free previous allocations
                for (int j = 0; j < i; j++) {
                        free(matrix2[j]);
                }
                free(matrix2);
                return 1;
        }
}
```

— C programming from the beginning —

<stdio.h> was #include ‘d // Standard I/O library, also printf

<stdlib.h> to use malloc()

No iostreams or stream operators (no cout or cin)
I/O, use printf() fo routput and scanf for input

printf() ➝ format string and contains conversion specifiers to interpret and display remaining arguments
➝ contains flags
➝ field width, precision, format

Conversion specifiers begin with a %, may include flags which control field width, precision, and format
Print % with %%
int x = 3;
printf("this is an int: %d\n", x); // %d tels print to format the appropriate parameters as an integer, and insert at the spot in the string

d	converts an int
f	converts a float
c	converts a char
s	converts a string
p	converts a pointer

flags:
l ➝ instead of an int/float, conver a long int or double
ll ➝ convert a long long int or long double
0 ➝ zero pad the conversion to fill the field width
- ➝ left justify the field
‘’ ➝ leave a blank space where an omitted + would go
+ ➝ display a + for positive numbers
non-zero int ➝ min field width
. (then a non-zero integer after the .) ➝ number of digits of precision

printf("pi = '%+10.4f'\n", pi); —> /* pi = '   +3.1416' */

- % = start of format specifier
- + = sign for positive numbers
- 10 = total min width of the output (spaces, sign, and digits) (this part: +3.1416) includes spaces
- .4 = number will display 4 digits after the decimal point
- f = value formatted as a floating-point number
- pi = variable the hold the value you want to print

printf("Hello from %s, which begins at %p!\n", __PRETTY_FUNCTION__, main); /* Hello from main, which begins at 0x4004f4! */

- This last one uses the __PRETTY_FUNCTION__ macro, which is a string representation of the function name. And the ‘main’ in there is printing out the hexadecimal address of where the main() function begins in the x86 code section.

Variable argument lists:
example: open()

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

both of these work, but the actual signature of open() is int open(const char *pathname, int flags, ...);
… ➝ means that the function takes a variable number of parameters
int scanf(const shar *format, …) converts input instead of output
scanf() converts input and needs a place to store it (pass-by-address, which is really passing a pointer by value) but c++ passes by reference.

scanf() matches non-whitespace characters in the format string. if there are multiple white spaces between characters, treats it as a single separator. For most format specifiers, scanf() stops the string after it encounters whitespace character in the output. In addition, characters in the input that match the format string (ex: /, :) are ignored.

ex:
int day, month, year;
scanf("%d/%d/%d", &day, &month, &year);
Input: "12/04/2024"

%d ➝ extract 12 and stores in day
'/' ➝ ignored since it matches the format string
The next %d extracts 04 and stores it in month.
The next / is ignored.
The final %d extracts 2024 and stores it in year.

ex:
```
int age;
char grade;
char school[3];

printf("AGE: ");
scanf("%d", &age);    /* Converts input to an integer and stores it in age */

printf("GRADE: ");
scanf("%c", &grade);  /* Converts input to a letter grade (probably 'A') and stores it in grade */

printf("SCHOOL: ");
scanf("%s", school);  /* Converts input to a string and stores it in school */
⇒ this will overflow buffer, so you need to use fgets() then sscanf() instead of scanf()
fgets(school, sizeof(school), stdin);
sscanf(school, %s, school);
```

Note: if you don't know how large your buffer (temporary storage area in a computer's memory that holds data while it's being moved from one place to another), then use fgets() then sscanf()

scanf() reads input from the standard input stream (stdin) and processes only the characters needed to match its format specifier.

- When it finishes processing, any leftover characters (like the newline character \n from pressing Enter) remain in the input buffer

fgets() reads the entire line from the input buffer, including the leftover newline character. If a \n is the only thing left in the buffer, fgets() will read just that and return immediately.

- If you use scanf() first (e.g., to read a number) and then call fgets(), the leftover \n from scanf() will cause fgets() to return immediately with an empty or incomplete line.

Dynamic Memory Management:

malloc() (c++ version of new) → allocation from the heap, returns a pointer to the newly allocated space on success
→ Null on failure, make sure to check return value before attempting to access the returned storage
→ storage allocated by malloc not initialized in any way
→ Initialization subroutines on data structures you allocate in C

void* malloc(size_t size) → must tell how much storage you require (sizeof important/useful)

void free(void* ptr) → deallocate storage with malloc, callo, realloc; similar to delete
→ don't call free() on any address not returned by malloc(), on null pointer, or twice on same address

Derived Types:

C doesn't have classes, it has structs

```
struct name { // if name omitted, struct becomes anonymous
 // but if name used inside struct, incomplete type or forward references used to build recursive structures
        type1 member1;
        type2 member2;
        …
        typen membern;
};
// outside of struct, instances can be declared.
```

```
typedef struct list_item {
   struct list_item* prev; // Must use 'struct list_item*' here since typedef doesn't exist yet
   struct list_item* next;
   void* datum;
} list_item_t; // 'list_item_t' is not yet known here
```

typedef → allows us to refer to the struct as list_item_t

- thus, to make variable declaration of the struct type, you can do:
    - list_item_t the_item

However, inside the struct, must be written as struct list_item* … since typedef isn't defined yet at that point, it just works for the list_item_t

Union:

1. type for which the compiler only allocated space sufficient for the largest member, not for all of the members
2. At any moment in union, only one member is valid
3. syntax same as a struct
4. Anonymous union can provide useful syntactic sugar for structure member

Function pointers:
Use case: function tables (arrays of function pointers), implementation of code where you might noy know what function will be called (qsort, bsearch)
int (*pprintf)(const char* format, ...) // function pointer
function name (name of the pointer, pprintf)
pointer that points to printf() // not useful really
Instead, set address of printf to pprint f → pprintf = printf;
This then works: pprintf("Hellow World!\n);

qsort():
Use case: quick sort, sorts arrays of data of arbitrary type
Subroutine has no knowledge of how to compare arbitrary elements being sorted, qsort() takes a pointer to a comparison function
void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*));
The comparison function takes pointers to two elements of the array starting at base and returns negative, zero, or positive if the first is smaller than, equal to, or larger than the second, respectively.

#include <unistd.h> // For the write() system call
#include <string.h> // For string manipulation functions like strlen()

ssize_t write(int fd, const void *buf, size_t count);

STDIN_FILENO (0): Standard input.
STDOUT_FILENO (1): Standard output.
STDERR_FILENO (2): Standard error.

fd: File descriptor where the data is to be written (e.g., STDOUT_FILENO).
buf: Pointer to the buffer containing the data to write.
count: Number of bytes to write from the buffer.
Return Value:

On success, returns the number of bytes written.
On error, returns -1 and sets errno appropriately.

Underfull reads in C: read operation reads fewer bytes than expected. Happens due to reaching the end of a file or an interrupt.

ssize_t: `read`, `write`, `getline`
size_t: `sizeof`, memory allocation functions like `malloc`

error handling for read() → returns -1
error handling for malloc() → returns NULL
error handling for open() → returns -1
error handling for write() → returns -1

OWASP Top 10 Web Application Security Risks:

1. Injection: SQAL< LDAP< NOSQEL< OS
    a. enter username and password, inject code that causes return TRUE
2. Broken Authentication
    a. Unique username and password that are different from your username and passwords from different websites
3. Sensitive Data Exposure:
    a. Personal information transfer
4. XML External Entities (XXE):
    a. External
5. Broken Access Control:
6. Security Misconfiguration
7. Cross-site Scripting XSS:
    a. Send attack to machine
8. Insecure Deserialization:
    a. Fetch properties of object, need to serialization to convert into byte code, server side does JSON deserialization back to JSON
    b. You could give btye code that gets converted into JSON that is actually code malicious code that gets executed
9. Using Compoenents with known vulnerabilities
    a. you get errors, what server and version you are using
10. Insufficent Logging and Monitoring:
    a. Log files, make sure if attacker is not hacking

ValGrind:

Runtime instrumentation of your binary useful for profiling and debugging memory.
Very useful for catching memory leaks (for example, forgetting to free a pointer to a buffer dynamically allocated with malloc).

You can run an executable with Valgrind using the following command.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose <program>
```

Goal is that it will say that you have 0 bytes leaked (it will notify you if you have lost memory due to a leak).

Address Sanitizers:

Useful libraries you can use for debugging by instrumenting your binary at compile time.
To use these, you have to recompile your binary and link these libraries as shared objects.
google/sanitizers: There are many that exist but here are the main ones.
AddressSanitizer: captures a wide range of improper accesses to addresses in memory (shown on the right).
UBSan: captures patterns involving undefined behavior where the compiler can act on your code with unknown results.
On the SEASNet servers, its quite hard to compile the sanitizer libraries with GCC so we recommend using a different compiler, clang. You can use the following command.

```
clang -g3 -Wall -Wextra -mtune-native -mrdrnd -fsanitize=address -fsanitize=undefined *.c -o randall
```

After compiling your program to include these specific libraries, you can run it. If there is some behavior captured by one of the sanitizers, your program will stop and will show an output similar to the right.