CS 97 Final Review

Week 8

- HTML + JSX
  - HTML tags
    - **M**arkup **L**anguage
    - Describing what content a webpage should have
      - **more appropriate text**

        ```
        <b>more appropriate text</b>
        ```

      - Sometimes tags don't have a **closing one**

        ```
        <img src="crouton.png"/>
        <br/>
        ```

    - HTML elements have **attributes/properties** and **children**
      - For example, the `src="crouton.png"` is an attribute of the `img` tag.
      - The **child** of an HTML tag is the stuff inside of it.
        - In the `<b>` tag above, its child is "more appropriate text".
        - Children can be **text** or it can be **other HTML tags**
  - DOM
    - Document Object Model
    - A data structure that encapsulates the HTML structure of your website.
    - Starting from an HTML document, we can create a tree (!) that represents the "order" of HTML elements. For example:

      ```
      <body>
        <h1>Title!</h1>
        <div>
          <div id="cat">bread recipe</div>
          <img src="bread.png"/>
        </div>
      </body>
      ```

```
          body
         /    \
        h1    div
             /   \
           div   img
```

- **One of the most important distinctions to make is between HTML code and the DOM**.
- The **DOM** is generated by the **HTML code**.
- A browser uses the **DOM** to "figure out" what to render and display to the user.
- `document.getElementByID("cat").innerHTML = "cat"`

- JavaScript
  - JavaScript can manipulate the DOM to change the webpage.
  - Google Chrome > Right click and open developer tools > Console tab
    - You can access a webpage's `document` object and do stuff with it
    - `console.log(document)`
  - JSX
    - React uses **JSX**
    - It is just JavaScript + XML (the tags that you're allowed to return in React components)

```
render() {
  return (<div>
    <p>Hooray</p>
  </div>);
}
```

    - JSX is "transpiled" to normal JavaScript: the above code will actually be "converted" to regular JavaScript by a program called `babel`:

```
render() {
  return([
    React.createComponent("div"),
    React.createComponent(....)
  ])
}
```

  - **Compiling** is a compilation from source code to an executable binary (machine code)
  - **Transpiling** is a translation from source code to more source code
- **DTDs**

- HTML used to rely on SGML
- These elements were standardized by Document Type Definitions (DTDs)
- It specifies what elements and entities are allowed for each element, it specifies what attributes it can have and what content it can have DTD specifies a *grammar* for an HTML document one way to think about it is that DTD is “like” an API contract between web server and a browser if they agree, then the user sees useful text
- People stopped using DTDs because control was essentially passed to browser's to determine how to render HTML/etc.

Week 9

- node.js
  - Event-based programming
    - It's a **paradigm**/model/blueprint/way of writing a program:
    - Core concepts:
      - **Event loop**
      - **Event handlers**

```
// Event loop
while (true) {
  event = wait_for_event();
  event.handle();
}
```

      - Wait for an event to happen
      - When it happens, handle it!
    - What is an event?
      - In different contexts, it can be very different, but generally an event is anything that can happen spontaneously
        - Keyboard input (user touched a key)
        - Network data that we requested has come in
        - The user clicked a button
        - The driver pressed the brakes.
    - Core idea: **Asynchronous handling of events as they come in!**
    - Why event-based programming???

```
// Event loop
while (true) {
  event = wait_for_event() // CPU can sleep here;
  event.handle(); // wake up
}
```

        - What happens at `wait_for_event`?
            - Ideally, while we are waiting for events to process, our CPU/computational resources are sleeping - in practice, they are doing other work so that we are being very efficient.
            - In C, there is a system call named `poll`, which waits on an "input source" until it has data that is ready to be read by the program. While the program is polling, the CPU is allowed to do other useful work.
        - What alternative??

```
while (true) {
  //Check if the button was pressed
  // If not pressed, stay in loop
  // If pressed, break out of loop
}
```

- **Callbacks**
    - Callbacks are just functions that you pass to other functions

```
function i_take_a_callback(cb) {
  do_some_things();
  cb();
}
```

    - Why asynchronous programming?
        - When we have a lot of work that needs to get done at the same time; e.g. a server that is handling 100 connections at the same time - we don't want to handle client 1 and then client 2 and then client 3, etc... we want to give everyone the same amount of time/resources **concurrently**!!!
- Processes and Threads
    - A **thread** is a sequence of instructions to be executed in order.
    - A program can have multiple threads
        - Each thread shares the same memory
        - But each thread has it's own set of instructions (that may be the same) and executed independently of one another.
    - The point of threads is concurrency (i.e. executing multiple instructions at the exact same time)
        - Multiple CPUs on your computer
    - How do threads and processes differ?
        - **Main differences**
            - Threads share memory, processes do not.

```
// Event loop
while (true) {
  event = wait_for_event() // CPU can sleep here;
  pass_to_thread(event.handle()); // wake up
}
```

        - Handle each **event** on its own thread
- C programming + the C compilation process
    - "C is a simple but also complicated language" - Tim Gu
    - Preprocessing - People
        - Get rid of **preprocessor directives** - lines that start with "#"
        - E.g. `#include` means "please find the library to be included and copy and paste the code from there into here"
    - Compiling - Can
        - Transforms the preprocessed source code into **assembly code** (architecture dependent)
            - From `source.c` to `source.S`
            - `mov %rax, %rbx`
    - Assembling - Always
        - Transforms the assembly code into an **object file**
            - From `source.S` to `source.o`
            - The object file, unlike the assembly file, is not human-readable
    - Linking - Love
        - Combine all of the generated `.o` files into a single **executable**
            - `first.c` `second.c`
            - `first.c` imports functions that are implemented in `second.c`
            - Linking functions that were declared in other files (i.e. figuring out where they are)
    - Loading
        - Done when you actually **run** the executable.
            - Load dynamic libraries (don't need to know details about what this is/how it works)

Week 10

- Compilers
    - A compiler such as GCC performs all of the steps we listed ^^, it may invoke other programs to help it do this

        - For example, there is a program called `ld` whose job is solely to perform linking.
        - `cpp` is a C preprocessor
    - We talked about GCC
        - What compilers are good for, besides compiling your program into machine code:
            - Security improvement
            - Performance improvement
            - Static checking
            - Dynamic checking (arrange for this)
            - Portability checking
    - Why compilers?
        - There are many dumb ways to write code

```
for (int i = 0; i < 1000000; i++) { }
cout << "actual work" << endl;
```

        - Never fear, the compiler is here to save the day and make your code better!
    - **Security Improvement**
        - `-fstack-protector`
            - Every program has a bit of memory called the stack
                - Local variables and other fun stuff are allocated on the stack
                - Side note: stuff you declare with `new` (that will be `delete`d later) in C++ is allocated not on the **stack**, but on the **heap**.
            - When you run a program, its **instructions** are actually also loaded into memory. And this memory (that contains the instructions) often "lives" very close to the memory allocated for the stack
            - So if you **really** wanted to be evil, there are ways that you can you write your program so that a malicious user could, in theory, intend for data to be written on the stack, but actually make it so that it overwrites some of the actual instructions/return addresses that your program relies on
            - The option `-fstack-protector` will cause GCC to generate extra code that it inserts into your program to protect your program from these malicious attacks
    - **Performance Improvement**
        - Example with useless for loop up there^
            - Writing your code so that it's more efficient

```
int a = 5;
... never use a
```

            - Compilers can optimize away useless code

```
int a = 1 + 2 + 3 + 4;
```

        - Compiler will actually do this ahead of time, so that it's not done at run-time.
    - **Static checking**
        - Static checking is ensuring that, for example, if you have a function that takes in integers, wherever you call it, you pass it an integer.
        - Statically typed languages are languages that require you to declare what **type** variables are, like integers, strings, bools, floats, and then ensure, for example, that a function that takes in an integer is only ever passed an integer.

```
void blah(string s) {
  cout << s << endl;
}

blah(100); // compiler yells at you because of static checking
```

```
javascript:

function f(cb) {
  // what is the type of cb???? idk
  cb();
}

python:

def a(b):
  // what is the type of b???? idk
```

        - Static checking isn't just about ensuring that variables are the right types, it can also ensure things like making sure that all allocated memory is correctly freed (think `new` and `delete`)
    - **Dynamic checking**
        - `gcc -fsanitize=leak`
            - catches memory leaks
        - `gcc -fsanitize=thread`
            - multithreaded race conditions
    - **Portability checking**
- Debugging and profiling
    - valgrind
        - Memory-checking tool

- GDB
  - GNU Debugger
  - It's a tool that allows you to execute a program step by step and pause along the way, and you can inspect what values variables have, etc. etc.
  - gdb -tui for a ~ fun graphical interface ~
  - **Breakpoints**
    - A breakpoint is a place in your code that you would like the debugger to essentially "pause" your program at. For example:

```
int doComplicatedStuff() {
  int a = 5;
  for (int i = 0; i < 10; i++) {
    a += 10;
  }
  int b = 10; <<<< say I set a breakpoint here
  int c = a - b;
  return c;
}
```

    - In gdb, you can specify breakpoints by line number, by function name, memory address, etc. etc.
    - When you run your program (in gdb), it will execute normally until it hits the breakpoint, at which point your program will "suspend" or "pause", and then you can issue more gdb commands to inspect variable values, step through code instruction by instruction, etc.
  - **Stepping through code**
  - **Inspecting the current values that variables have**

Networking

- Client–server computing
  - A client requests a webpage, the server sends it back.
    - The clients are in **control**, they are the ones "calling the shots"
    - The server's job is "reactive"; its sole purpose is to respond to client requests
  - P2P
    - There is no central server; instead, resources are spread out amongst many different **peers** (computers) all connected in one network.
    - E.g. you request a File A, Alice has the first 1/3 of file A Bob has the second 1/3 and Charlie has the last 1/3. They all need to send you their data
- Performance issues

    - Throughput (and out-of-order execution)
        - How much data (i.e. volume) can be sent at once?
    - Latency (and caches)
        - How much time does it take for a single bit to get from point A to point B?
- The World Wide Web
    - HTTP/1.1, /2, and /3
    - The Internet fundamentally solves the problem: how do we get information from one computer to another?
        - There are many **protocols** involved to get this done. HTTP is one of them.
            - A **protocol** is just a set of rules or standards to follow/agree upon so that everyone can get their jobs done
        - HTTP is an **application-level** protocol
        - HTTP has Requests and Responses
        - Different versions of HTTP
            - /1.1
                - Most common but also **worst**/slowest
                - HOL blocking
            - /2
                - Allows for multiple connections.
                - Say I want to request 10 images from a server
                    - In HTTP 1.1 I have to wait for reach response to come back before I can send the next request
- Transport level protocols
    - UDP
        - Unreliable
        - I guess I'll try, idk if it'll get there can't make any promises lmao
        - UDP is faster and more performant
    - TCP
        - Reliable
        - I guarantee you, if you send data over TCP, **at some point**, the data will be sent successfully
        - Requires much more overhead
        - HTTP runs over TCP
            - TCP involves a "three-way handshake"
                - Basically it guarantees that both client and server are ready to send and receive messages reliably

Shell + Regex

- What is the shell?
  - A shell is usually a wrapper around "all the good stuff"
  - In the computer's case, "all the good stuff" is the **operating system** and its resources - CPU time, access, modification, and creation of files, process creation and execution.
  - The shell is a way for us to access the operating system's services through a command-line interface (CLI).
- Basic shell syntax
  - `>` redirect output of a program into a file
  - `|` please pass the output from one program to another
  - `<` please provide this file as input to a program
- What you should be really familiar with:
  - POSIX commands
  - Manipulating input and output of programs
  - Writing to and modifying files
  - File permissions

Regex

- A regular expression allows you to match lines of text; by match, we mean find substrings in text that adhere to a certain pattern.
- For example, the regex "ab*" matches all strings that start with an 'a' and then have 0 or more 'b's after.
- You can pass regexes to shell commands like `grep`, `sed`, etc. to ask them to often find lines that contain text that matches these regexes.
- Extended regular expressions vs. Basic regular expressions
  - `grep` vs `grep -E`

Git

- Version control system
  - Refer to lecture notes for what services good VCSes should provide
- Most important git commands to know
  - `git add`
  - `git commit`
  - `git push`
  - `git pull`
  - `git clone`
  - `git checkout`

    - `git reset`
    - `git branch`
    - `git merge`
    - `git rebase`
- Concepts
    - Staging area
        - `git add`
        - Stages files to be committed
        - Notifies git that you want to start tracking changes to a file
    - Actually making commits
        - `git commit`
        - Create a "snapshot" of the repo
    - Sharing your work via the Internet
        - `git push`
        - `git pull`
        - `git clone`
    - Branching, merging
        - `git checkout`
        - `git branch`
        - `git merge`
    - Rewriting history
        - `git rebase`
        - `git reset`
    - **Merging vs rebasing**
        - Merging only goes "forward in time"
        - A rebase goes "backwards in time"
        - But they're both trying (generally) to solve the same problem - given two branches, i.e. two different versions of a codebase, how do we merge them to one single version that respects the changes made in each
        - When you run `git merge`
            - Git tries to merge both versions of the codebase and once it successfully does that, it **creates a new commit** which represents those merged changes.
        - When you run `git rebase`
            - Git first **unwinds** commits that may have been made before placing your changes on top of it
        - In practice, you always prefer **git merge** to **git rebase** because you never **lose history** when merging. (losing history refers to getting rid of commits and pretending that they never happened). However, in certain cases where you want to get "rid" of bad commits, you have to use a rebase, i.e. git merge cannot be used to delete/unwind

commits

- Git internals
    - As mentioned earlier, whenever you make a commit you "ask" Git to create a snapshot of the current repo. This snapshot consists of a single commit object that points to a Tree, which points to Blobs or Other Trees
        - Commit
            - The commit object typically contains the author's name and time of the commit - it contains a lot of metadata
        - Tree
            - The tree contains info about what blobs are relevant to the commit in question
        - Blob
            - Raw file data
        - All of these objects are compressed (gzip) and have their own unique SHA-1 hash which is used to identify them.
        - Assignment 2b as well
- Miscellaneous