This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

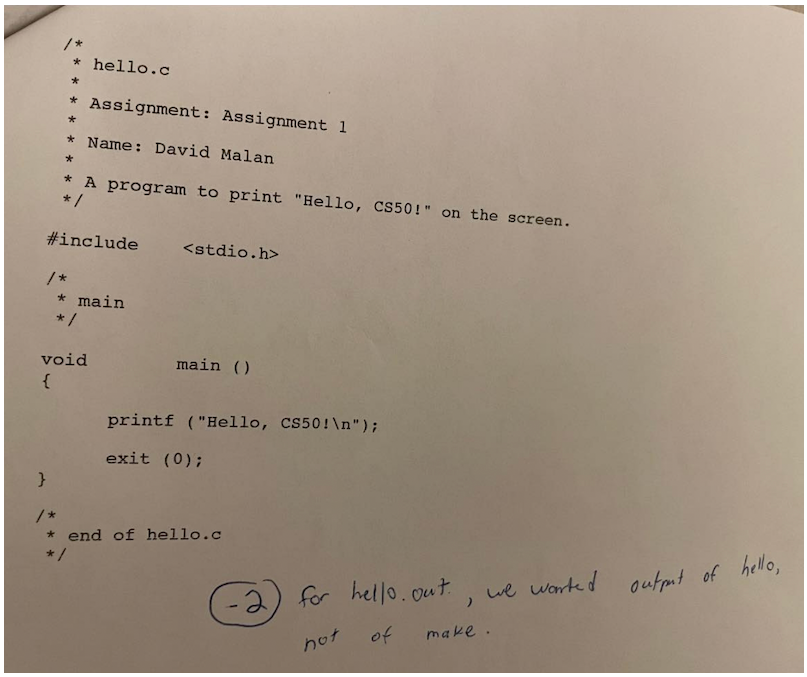
Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 0

Welcome

- This year, we're back in Sanders Theatre, and David took CS50 himself as a sophomore years ago, but only because the professor at the time allowed him to take the course pass/fail.
- It turns out that computer science was less about programming than about problem solving. And though there may be frustration from feeling stuck or making mistakes, there will also be a great sense of gratification and pride from getting something to work or completing some task.
- In fact, David lost two points on his first assignment for not following all of the instructions correctly:



```
/*
 * hello.c
 *
 * Assignment: Assignment 1
 *
 * Name: David Malan
 *
 * A program to print "Hello, CS50!" on the screen.
 */

#include    <stdio.h>

/*
 * main
 */

void        main ()
{

      printf ("Hello, CS50!\n");

      exit (0);
}

/*
 * end of hello.c
 */
```

(-2) for hello.out, we wanted output of hello, not of make.

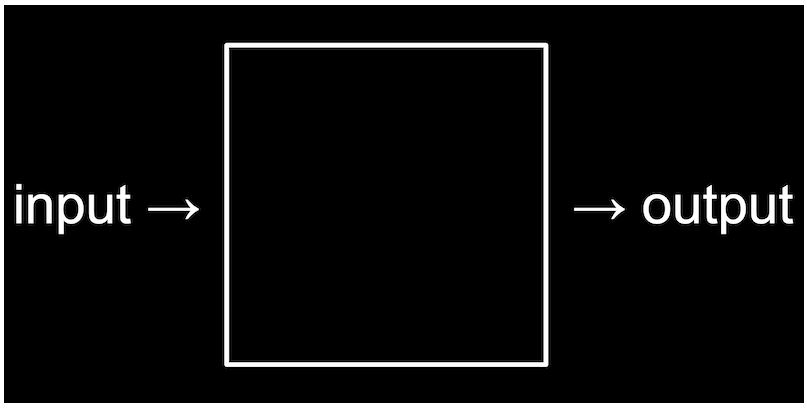
  - And while this code (written in a programming language as opposed to a language like English) looks cryptic at first, it may only take weeks or months before we can understand main programming concepts and even teach ourselves new languages.
- Importantly,

  > what ultimately matters in this course is not so much where you end up relative to your classmates but where you end up relative to yourself when you began

- In fact, two-thirds of CS50 students have never taken a computer science course before.

What is computer science?

- Computer science is fundamentally problem solving, but we'll need to be precise and methodical.
- We can think of **problem solving** as the process of taking some input (a problem we want to solve) and generate some output (the solution to our problem).



- To begin doing that, we'll need a way to represent inputs and outputs, so we can store and work with information in a standardized way.

Representing numbers

- To count the number of people in a room, we might start by using our fingers, one at a time. This system is called **unary**, where each digit represents a single value of one.
- To count to higher numbers, we might use ten digits, 0 through 9, with a system called **decimal**.
- Computers use a simpler system called **binary**, with just two digits, 0 and 1.
- For example, in binary this would be 0:

```
0 0 0
```

- And this would be 1:

```
0 0 1
```

    - (We don't need the leading zeroes, but we'll include them to see the patterns more easily.)
- Since there is no digit for 2, we'll need to change another digit to represent the next number:

```
0 1 0
```

- Then we'll "add 1" to represent 3:

```
0 1 1
```

- And continue the pattern for 4 ...:

```
1 0 0
```

- ... 5 ...:

```
1 0 1
```

- ... 6 ...:

```
1 1 0
```

- ... and 7:

```
1 1 1
```

- Each *binary digit* is also called a **bit**.
- Since computers run on electricity, which can be turned on or off, we can simply represent a bit by turning some switch on or off to represent a 0 or 1.
- Inside modern computers, there are billions of tiny switches called **transistors** that can be turned on and off to represent different values.
- And the pattern to count in binary with multiple bits is the same as the pattern in decimal with multiple digits.
- For example, we know the following number in decimal represents one hundred and twenty-three.

```
1 2 3
```

  - The `3` is in the ones place, the `2` is in the tens place, and the `1` is in the hundreds place.
  - So `123` is `100×1 + 10×2 + 1×3 = 100 + 20 + 3 = 123`.
  - Each place for a digit represents a power of ten, since there are ten possible digits for each place. The rightmost place is for 10^0, the middle one 10^1, and the leftmost place 10^2:

```
10² 10¹ 10⁰
1   2   3
```

- In binary, with just two digits, we have powers of two for each place value:

```
2² 2¹ 2⁰
#  #  #
```

  - This is equivalent to:

```
4  2  1
#  #  #
```

- With all the light bulbs or switches off, we would still have a value of 0:

```
4 2 1
0 0 0
```

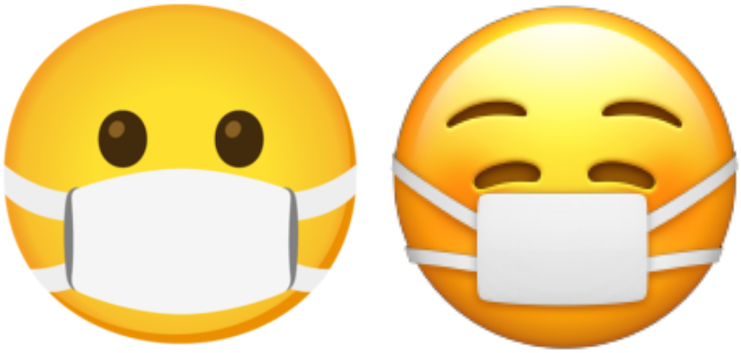
- Now if we change the binary value to, say, `0 1 1`, the decimal value would be 3, since we add the 2 and the 1:

```
4 2 1
0 1 1
```

- To count higher than 7, we would need another bit to the left to represent the number 8.
- Most computers use 8 bits at a time, like `00000011` for the number 3.

Text

- To represent letters, all we need to do is decide how numbers map to letters. Some humans, many years ago, collectively decided on a standard mapping of numbers to letters. The letter "A", for example, is the number 65, and "B" is 66, and so on. In binary, the letter "A" is the pattern `01000001`. By using context, like the file format, different programs can interpret and display the same bits as numbers or text.
- The standard mapping, **ASCII** (https://en.wikipedia.org/wiki/ASCII), also includes lowercase letters and punctuation.
- When we receive a text message, we might be getting patterns of bits that have the decimal values `72`, `73`, and `33`. Those bits would map to the letters `HI!`. And the sequences of bits we receive would look like `01001000`, `01001001`, and `00100001`, with 8 bits for each character.
- With eight bits, or one byte, we can have 2^8, or 256 different values (including zero). (The highest *value* we can count up to would be 255.)
- And we might already be familiar with using bytes as a unit of measurement for data, as in megabytes or gigabytes, for millions or billions of bytes.
- Other characters, such as letters with accent marks and symbols in other languages, are part of a standard called **Unicode** (https://en.wikipedia.org/wiki/Unicode), which uses more bits than ASCII to accommodate all these characters.
- When we receive an emoji (https://en.wikipedia.org/wiki/Emoji), our computer is actually just receiving a number in binary that it then maps to the image of the emoji based on the Unicode standard.
    - For example, the "face with medical mask" emoji is just the four bytes `11110000 10011111 10011000 10110111`:



- And it turns out that different companies that create software for their devices will have slightly different images that represent each emoji, since only the descriptions have been standardized.

Images, video, sounds

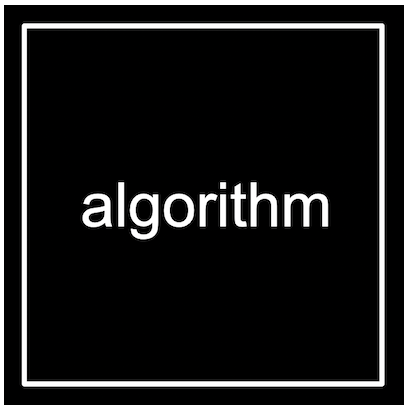
- With bits, we can map numbers to colors as well. There are many different systems to represent colors, but a common one is **RGB**, which represents colors by indicating the amount of red, green, and blue within each color.
- For example, our pattern of bits earlier, `72`, `73`, and `33` might indicate the amount of red, green, and blue in a color. (And our programs would know those bits map to a color if we opened an image file, as opposed to receiving them in a text message.)
  - Each number might be 8 bits, with 256 possible values, so with three bytes, or 24 bits, we can represent millions of colors. Our three bytes from above would represent a dark shade of yellow:

- The dots, or squares, on our screens are called **pixels**, and images are made up of many thousands or millions of those pixels as well. So by using three bytes to represent the color for each pixel, we can create images. We can see pixels in an emoji if we zoom in, for example:

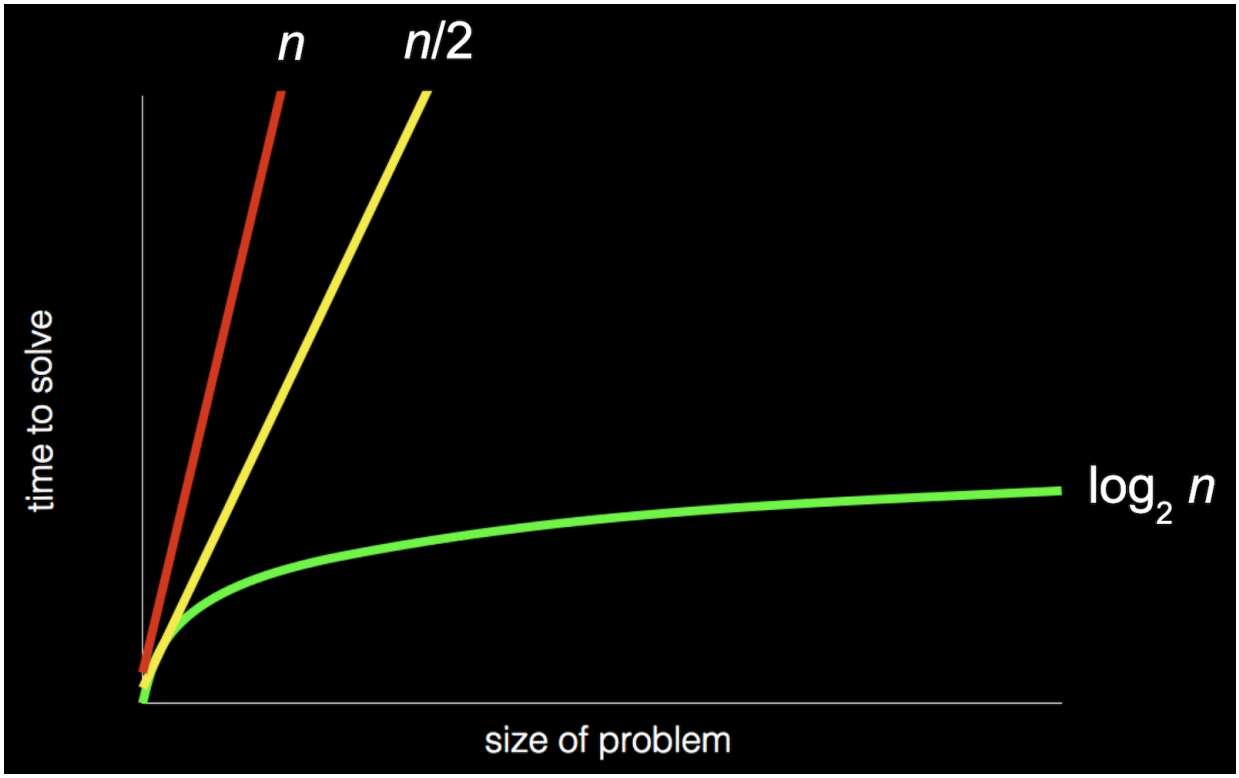
- Videos are sequences of many images, changing multiple times a second to give us the appearance of motion, as a flipbook (https://www.youtube.com/watch?v=sz78_07Xg-U) might.
- Music can be represented with bits, too. MIDI (https://en.wikipedia.org/wiki/MIDI) is one such format which represents music with numbers for each of the notes and their duration and volume.
- So all of these ideas are just zeroes and ones, interpreted and used by software we've written to interpret them in the ways that we want.
    - There are other formats, some of which use compression (mathematical ways to represent some data with fewer bits), or some which might be containers that store multiple types of data together.
    - And since there are many companies and groups developing software, we have lots of different file formats in existence, each with their own ways of representing data. But there are also organizations that work on some consensus, like the one (https://en.wikipedia.org/wiki/Unicode_Consortium) responsible for maintaining the Unicode standard.

Algorithms

- Now that we can represent inputs and outputs, we can work on problem solving. The black box that transforms inputs to outputs contains **algorithms**, step-by-step instructions for solving problems:



- We might have an application on our phones that store our contacts, with their names and phone numbers sorted alphabetically. The old-school equivalent might be a phone book, a printed copy of names and phone numbers.
- We might open the book and start from the first page, looking for a name one page at a time. This algorithm would be correct, since we will eventually find the name if it's in the book.
- We might flip through the book two pages at a time, but this algorithm will not be correct since we might skip the page with our name on it.
- Another algorithm would be opening the phone book to the middle, decide whether our name will be in the left half or right half of the book (because the book is alphabetized), and reduce the size of our problem by half. We can repeat this until we find our name, dividing the problem in half each time.
- We can visualize the efficiency of each of those algorithms with a chart:



  - Our first algorithm, searching one page at a time, can be represented by the red

line: our time to solve increases linearly as the size of the problem increases. *n* is a number representing the size of the problem, so with *n* pages in our phone books, we have to take up to *n* steps to find a name.

- The second algorithm, searching two pages at a time, can be represented by the yellow line: our slope is less steep, but still linear. Now, we only need (roughly) *n* / 2 steps, since we flip two pages at a time.
- Our final algorithm, dividing the phone book in half each time, can be represented by the green line, with a fundamentally different relationship between the size of the problem and the time to solve it. If the phone book doubled in size from 1000 to 2000 pages, we would only need one more step to find our name.

Pseudocode

- We can write **pseudocode**, which is a representation of our algorithm in precise English (or some other human language):

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

  - With these steps, we check the middle page, decide what to do, and repeat. If the person isn't on the page, and there's no more pages in the book left, then we stop. And that final case is particularly important to remember. When programs or code don't include that final case, they might appear to freeze or stop responding, or continue to repeat the same work over and over without making any progress.
- Some of these lines start with actions or verbs that solve a smaller problem. We'll start calling these *functions*:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
```

```
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

- We also have branches that lead to different paths, like forks in the road, which we'll call *conditionals*:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

- And the questions that decide where we go are called *Boolean expressions*, which eventually result in answers of yes or no, or true or false:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

- Lastly, we have words that create cycles, where we can repeat parts of our program,

called *loops*:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

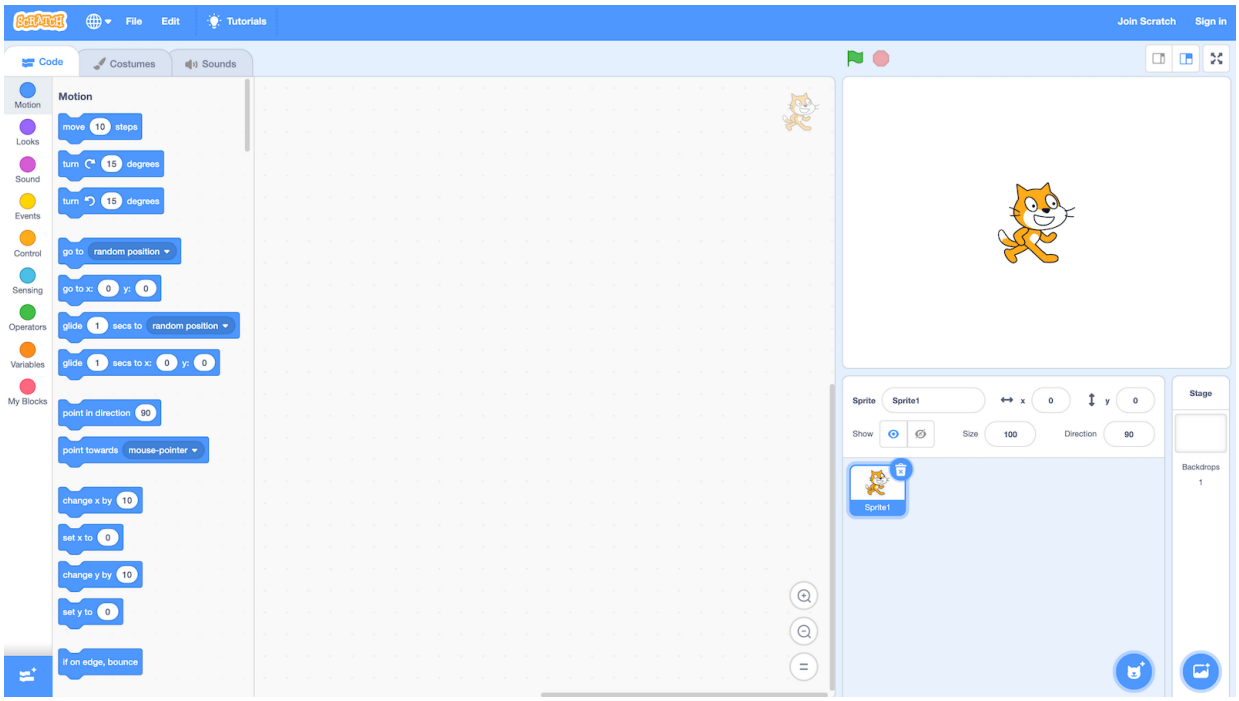
- We'll soon encounter other ideas, too:
  - functions
    - arguments, return values
  - conditionals
  - Boolean expressions
  - loops
  - variables
  - …
- And David's first program just printed "hello, world" to the screen:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

  - But this program, written in a language called C, has lots of other syntax that keeps us from focusing on these core ideas.

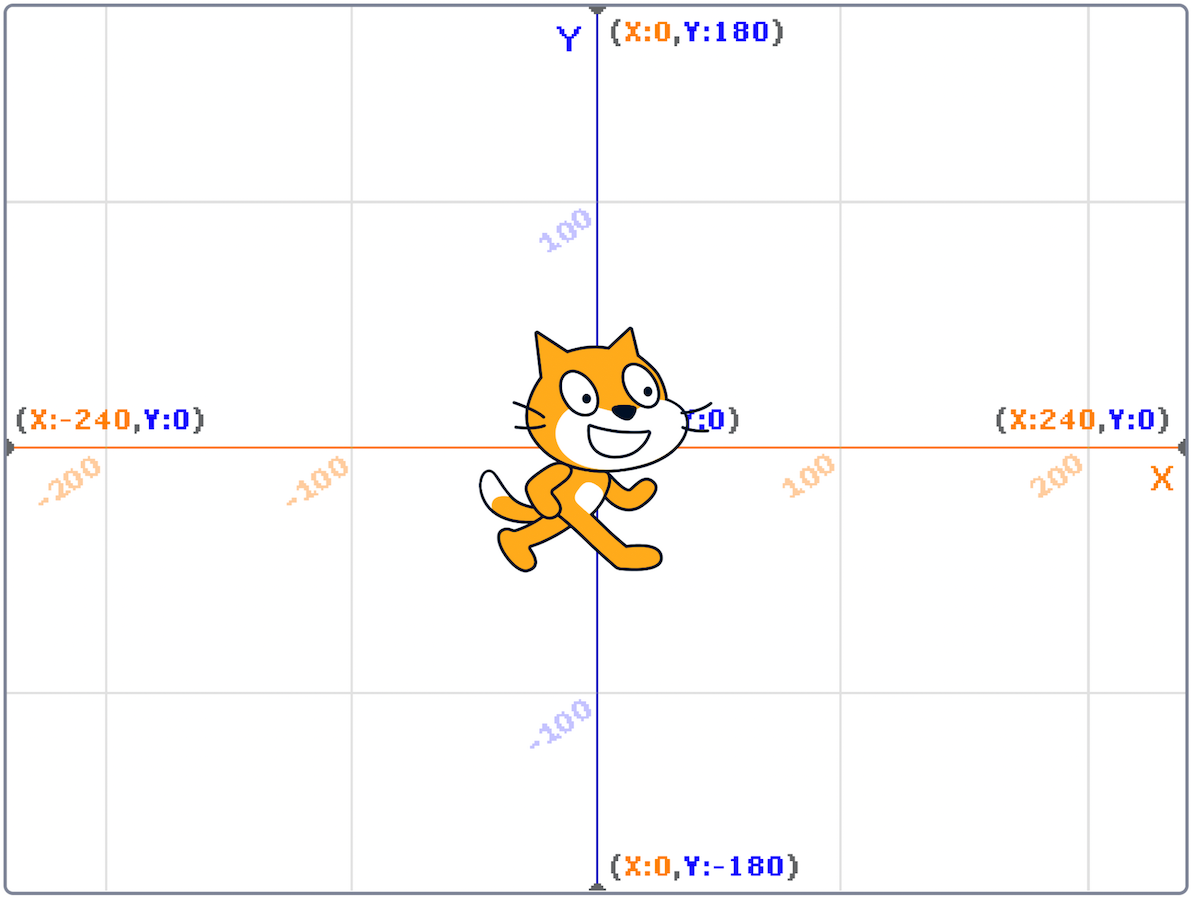
Scratch basics

- We'll start programming with a graphical programming language called Scratch (https://scratch.mit.edu/), where we'll drag and drop blocks that contain instructions.
- The programming environment for Scratch is a little more friendly:



- On the left, we have puzzle pieces that represent functions or variables, or other concepts, that we can drag and drop into the big area in the center.
- On the bottom right, we can add more characters, or sprites, for our program to use.
- On the top right, we have a stage, or the world that will be shown by our program.

- The world in Scratch has a coordinate-based system for positioning things on the screen:



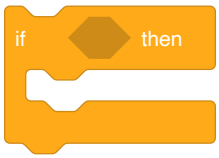
    - The center of the screen is a coordinate of 0 for x and 0 for y, and the top of screen would be 0 for x and 180 for y. The bottom of the screen would still be 0 for x, and -180 for y. The left of the screen would be -240 for x and 0 for y, and the right of the screen would be 240 for x and 0 for y.
- Scratch also categorizes its pieces, each of which might be a function, conditional, or more:
    - The "Motion" category has functions like the "move" block that will do something:

    - In the "Events" category, we can see blocks that will activate when something happens, like when the green flag on top of the stage is clicked:

    - "Control" has conditionals, each of which will only do something if the Boolean expression inside is true:

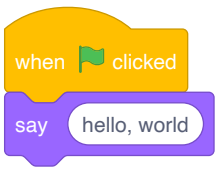
    - "Sensing" includes those Boolean expressions, or questions like whether the sprite is touching the mouse pointer:

    - "Operators" contains blocks that let us do math or pick random numbers, or combine multiple Boolean expressions:

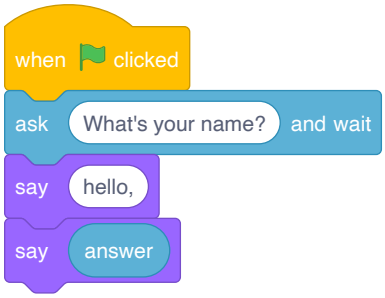
      

    - "Variables" will let us store values like words or numbers, and save them with names like `x`, `y`, or other full words to describe them.
    - We can even combine multiple blocks ourselves into a new puzzle piece, or function, with "My Blocks".
- We can drag a few blocks to make our cat say "hello, world":

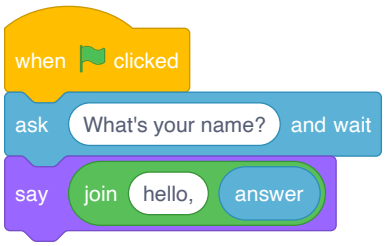
  

    - The purple block, "say", is a function that takes some sort of *input*, the text in the white oval, and makes our cat say it on the stage as its output.

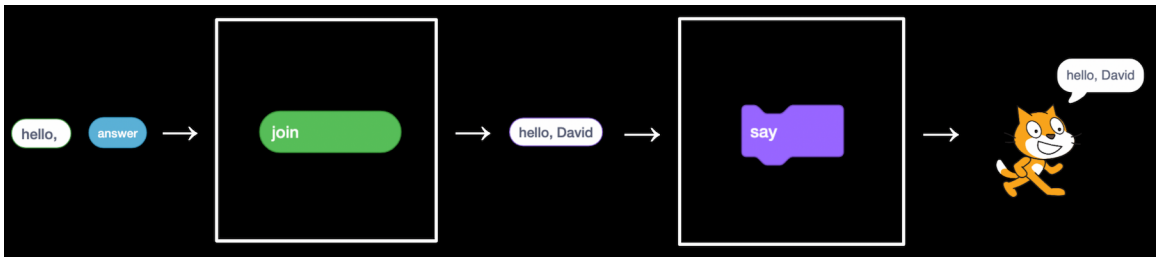
- We can also drag in the “ask and wait” block, with a question like “What’s your name?”, and combine it with a “say” block for the answer:



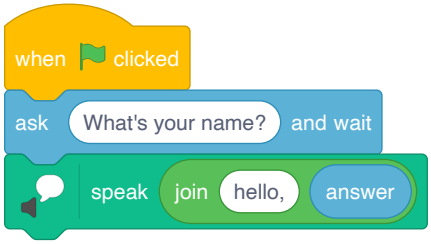
  - The “answer” block is a variable, or value, that stores what the program’s user types in, and we can place it in a “say” block by draggind and dropping as well.
  - The “ask and wait” block takes in a question as its input (or *argument*), and stores its *return value* into the “answer” block as output.
- But we didn’t wait after we said “Hello” with the first block, so we didn’t see the first message of “hello” before it was covered by our name. We can use the “join” block to combine two phrases so our cat can say “hello, David”:



  - Note that the “join” block takes not just one, but two arguments, or inputs, and its *output*, or the combined phrase, is used immediately as the *input* to another function, the “say” block:



- At the bottom left of the screen, we see an icon for extensions, and one of them is called Text to Speech. After we add it, we can use the “speak” block to hear our cat speak:

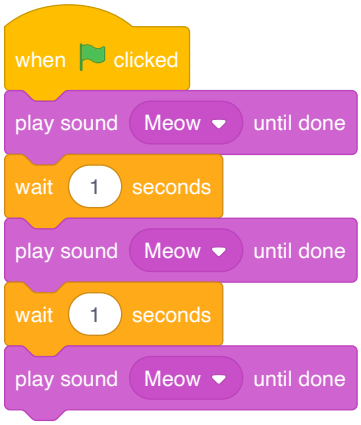


  - The Text to Speech extension, thanks to the cloud, or computer servers on the

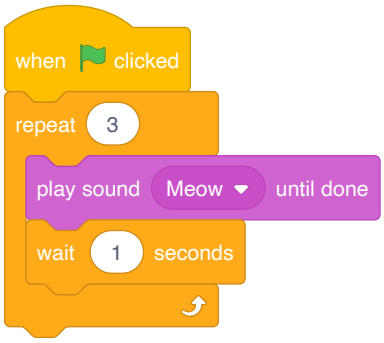
internet, is converting our text to audio.

Abstraction

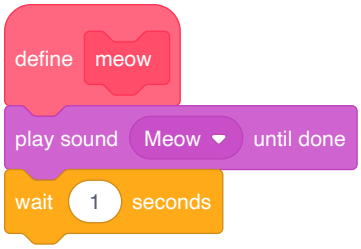
- We can try to make the cat say meow:

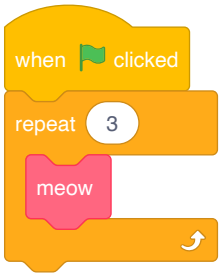


  - We can have it say meow three times, but now we're repeating blocks over and over.
- Let's use a loop, or a "repeat" block:

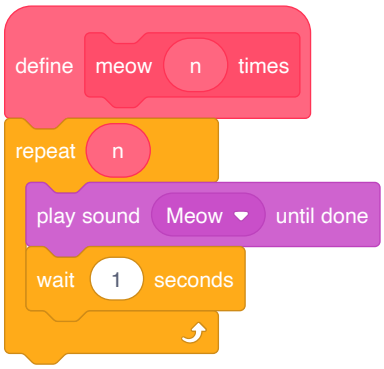


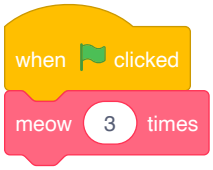
  - Now our program achieves the same results, but with fewer blocks. We can consider it to have a better design: if there's something we wanted to change, we would only need to change it in one place instead of three.
- We can use the idea of **abstraction**, or combining several ideas (or puzzle pieces) into one, so we can use and think about them more easily. We'll go into the "My Blocks" category, and click "Make a Block", and call it "meow":





    - Now, our main set of blocks can just use the custom "meow" block, and we'll be able to read this code later and understand what it does more easily.
    - We could even drag the set of blocks with "define meow" to the bottom of the screen so it's not visible, and this will still work, even if we don't know the **implementation details**, or exactly how our custom block works.
- We can change the "meow" block to take an input, so it can repeat any number of times:

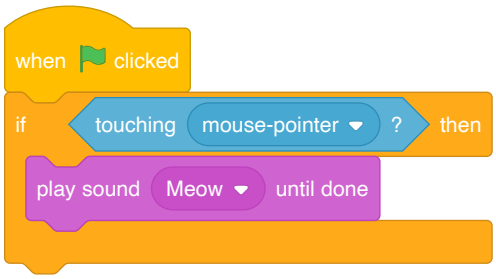




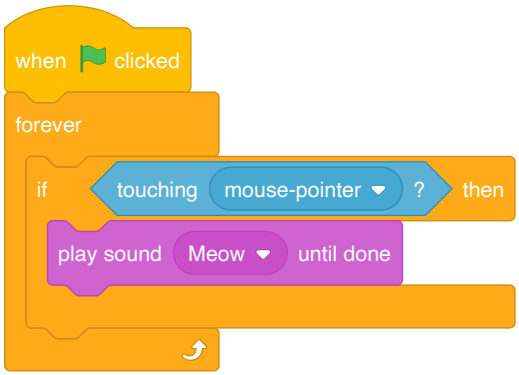
    - Now, our "meow" block achieves the same effect, but we can easily reuse it or change the number of times our cat says meow.
- A good strategy when programming is breaking down a larger problem into smaller subproblems, and solving those first.

Conditionals and more

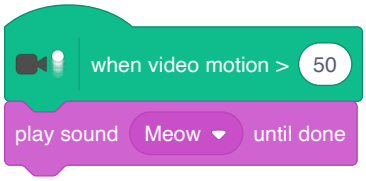
- We'll try to have our cat make a sound if we "pet" it with our mouse:



  - But this doesn't seem to work. That's because the cat is checking whether the mouse pointer is touching it right as the green flag is clicked, and nothing happens since we're clicking the flag.
- We can have our cat check over and over with the "forever" block:



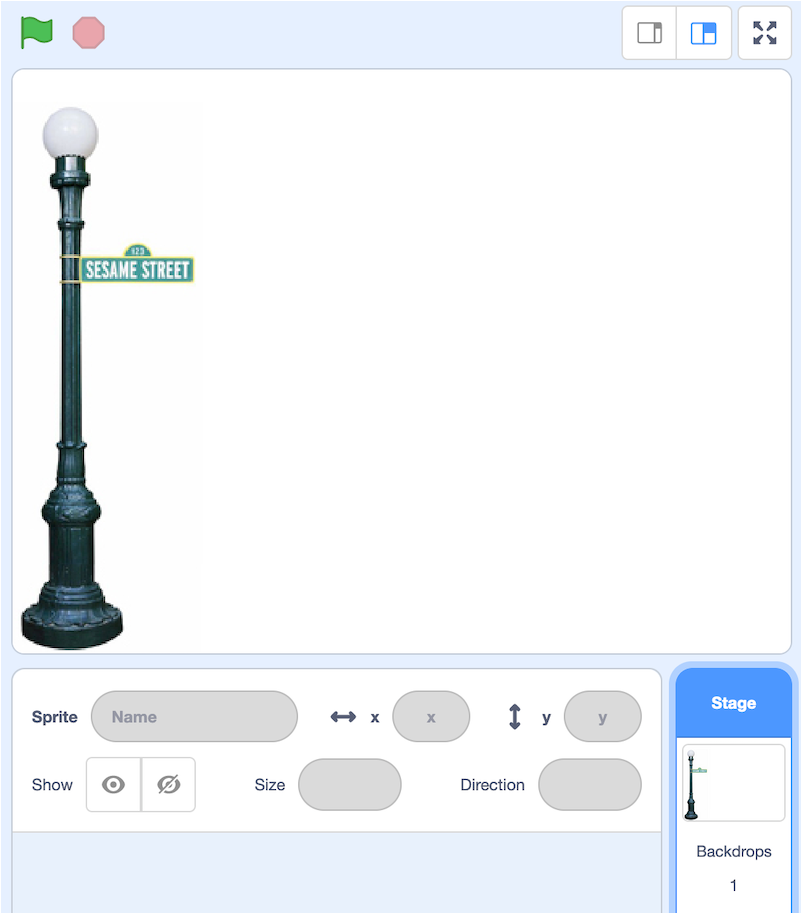
- We can add another extension, "Video Sensing":



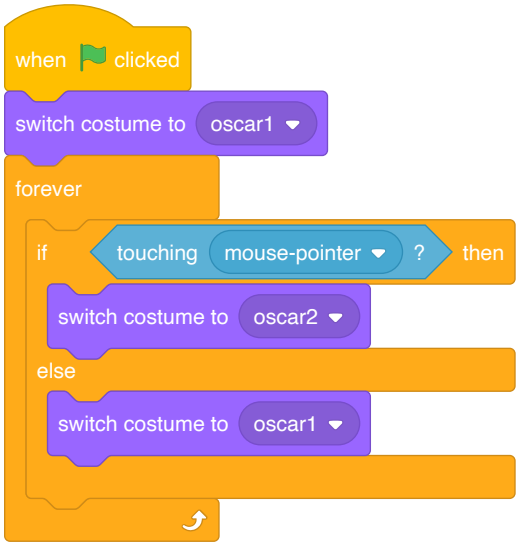
  - Now, if we move in view of the camera slowly, our cat won't make a sound, but if we move quickly, it will.

Demos

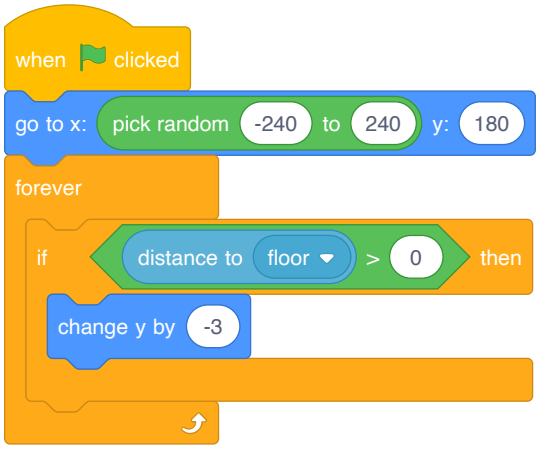
- With a volunteer from the audience, we demonstrate a whack-a-mole (https://scratch.mit.edu/projects/565362715) game.
- We also take a look at Oscartime (https://scratch.mit.edu/projects/277537196), another game where the player drags trash into a trashcan for points.
- We'll take a look at how we might have built this program. First, we can add an image of the lamp post (https://scratch.mit.edu/projects/565133620) as a backdrop:



- Then, we'll add images of trash cans (https://scratch.mit.edu/projects/565100517), with one open and one closed:

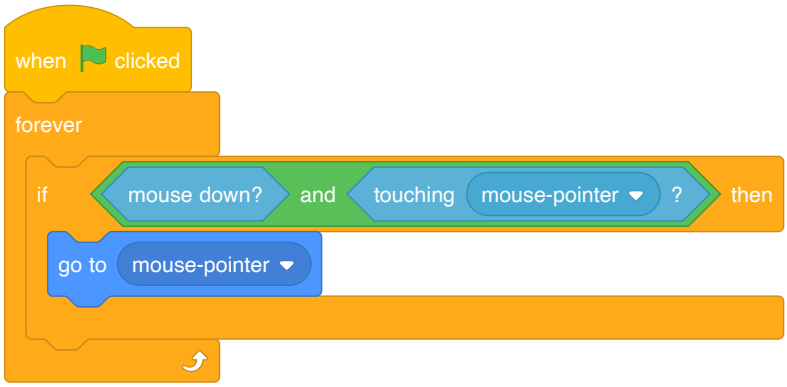


  - We name these costumes "oscar1" and "oscar2", and whenever the mouse is touching it, the trash can will appear to be open.
- Then, we'll work on a piece of falling trash (https://scratch.mit.edu/projects/565117390):



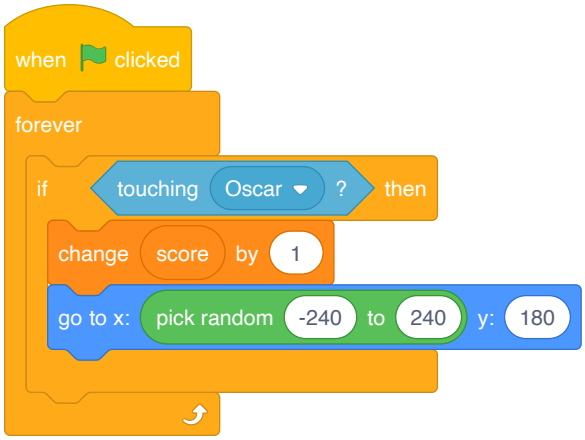
- We move the trash sprite to a random horizontal position, and have it move downwards over and over while the distance to the floor (another sprite that's a black line) is more than 0.

- We'll allow dragging trash (https://scratch.mit.edu/projects/565119737) with these blocks:



- If the mouse is down and touching the trash, then our trash will move to the mouse's location.

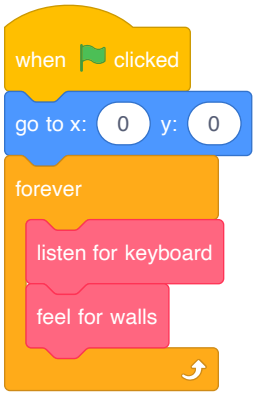
- Finally, we'll use variables (https://scratch.mit.edu/projects/565472267) to keep track of our score:

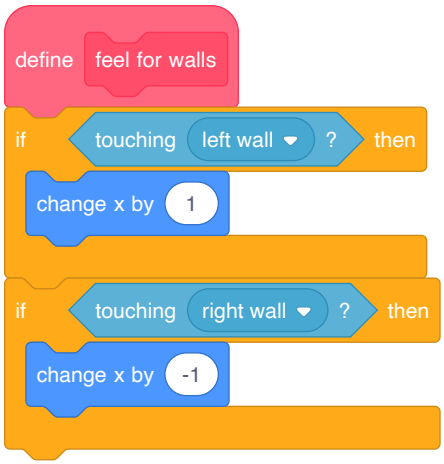


- Now, we have another sprite called "Oscar", and if the trash is touching it, then it will add 1 to the "score" variable, and move back to the top at a random horizontal

position so we can continue the game.

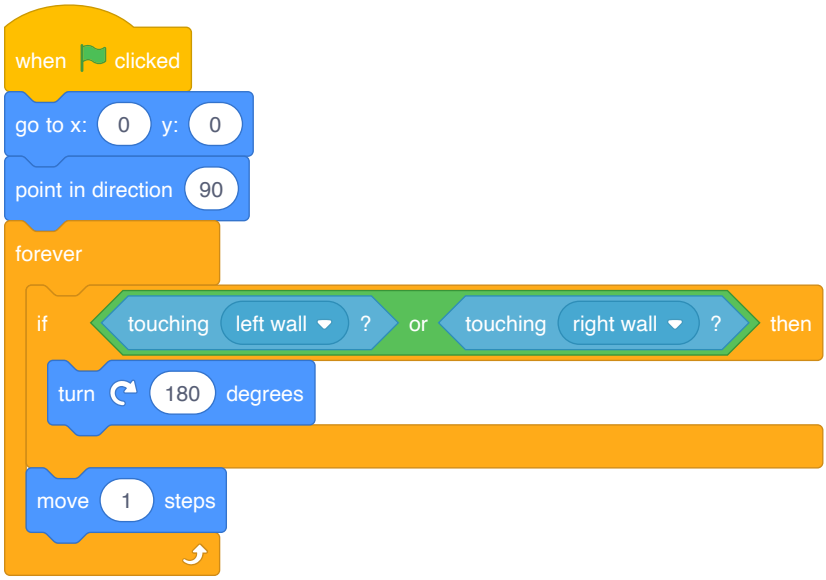
- Now, we'll take a look at moving (https://scratch.mit.edu/projects/565121265). Here, we have a few different scripts, one checking for whether keys are being pressed, and one for whether our sprite is touching a wall:



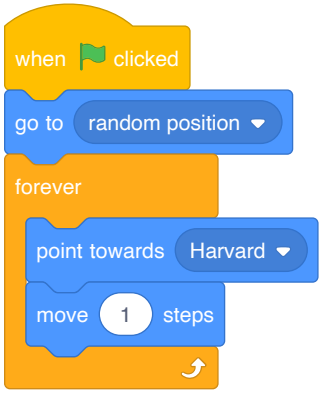


- Our main script, when the green flag is clicked, will move our sprite to the center of the stage at 0, 0 and then “listen for keyboard” and “feel for walls” forever.

    - The custom “listen for keyboard” script has blocks that will change our sprite’s x- or y-coordinate on the stage for each of the arrow keys, moving it around.
    - The “feel for walls” script will check whether the sprite is now touching a wall, and move it back if it is.
- We can make another sprite bounce (https://scratch.mit.edu/projects/565127193) back and forth, like it’s getting in our way:



    - First, we’ll move the sprite to the middle and have it point 90 degrees to the right.
    - Then, we’ll constantly check if it’s touching a wall and turn 180 degrees (reversing direction) if so, and move 1 step every time.
- We can have one sprite follow (https://scratch.mit.edu/projects/565479840) another:



    - Our sprite will start at a random position, and move towards our “Harvard” sprite one step at a time.
    - We can change the script to move two steps at a time, so it will always catch up.
- We’ll finish by trying out the full Ivy’s Hardest Game (https://scratch.mit.edu/projects/565742837) with a volunteer.
- See you next time!

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 1

- C
- IDEs, compilers, interfaces
- Functions, arguments, return values, variables
- main, header files, commands
- Types, format codes, operators
- Variables, syntactic sugar
- Calculations
- Conditionals, Boolean expressions
- Loops, functions
- Mario
- Imprecision, overflow

C

- Today we'll learn a new language, **C** (https://en.wikipedia.org/wiki/C_(programming_language)): a programming language that has all the features of Scratch and more, but perhaps a little less friendly since it's purely in text.
- By the end of the term, our goal is not to have learned a specific programming language, but *how* to program.
- The syntax, or rules around structure, punctuation, and symbols in code, will become familiar to us, even if we might not understand what everything does right away.
- With Scratch, we explored some ideas of programming, like:
    - functions
        - arguments, return values
    - conditionals
    - Boolean expressions
    - loops
    - variables
    - …
- Today, we'll translate some of those ideas to C, a computer language with new syntax and more precision, though fewer words to learn than a human language might include.
- As a first-year student, we might not have known all the information about our new campus right away, but instead learned what we needed to on a day-by-day basis. Here too, we'll start with the most important details, and "wave our hands" at some of the other details we don't need quite yet.
- When we evaluate the quality of our code, we might consider the following aspects:
    - **correctness**, or whether our code solves our problem correctly
    - **design**, or how well-written our code is, based on how efficient and readable it is
    - **style**, or how well-formatted our code is visually
- Our first program in C that simply prints "hello, world" to the screen looks like this:

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

IDEs, compilers, interfaces

- In order to turn this code into a program that our computer can actually run, we need to

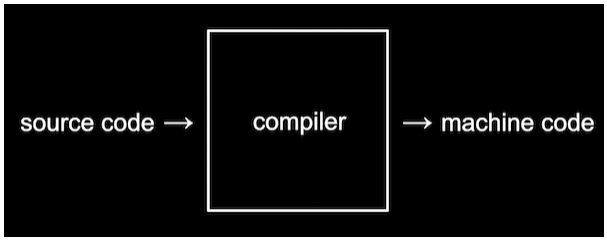
first translate it to binary, or zeroes and ones.

- Tools called IDEs, **integrated development environments** (https://en.wikipedia.org/wiki/Integrated_development_environment), will include features for us to write, translate, and run our code.
- One popular IDE, Visual Studio Code (https://en.wikipedia.org/wiki/Visual_Studio_Code), contains a text editor, or area where we can write our code in plain text and save it to a file:

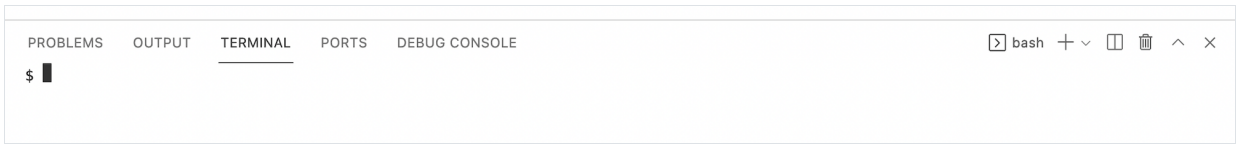
hello.c

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- Now our **source code**, or code that we can read and write, is saved to a file called `hello.c`. Next, we need to convert it to **machine code**, or zeroes and ones that represent instructions that tell our computer to perform low-level operations.
- A **compiler** (https://en.wikipedia.org/wiki/Compiler) is a program that can convert one language to another, such as source code to machine code:



- Visual Studio Code, also referred to as VS Code, is typically a program that we can download to our own Mac or PC. But since we all have different systems, it's easier to get started with a cloud-based version of VS Code that we can access with just a browser.
  - In Problem Set 1, we'll learn how to access our own instance of VS Code.
- In the bottom half of the VS Code interface, we see a **terminal** (https://en.wikipedia.org/wiki/Terminal_emulator), a window into which we can type and run text commands:

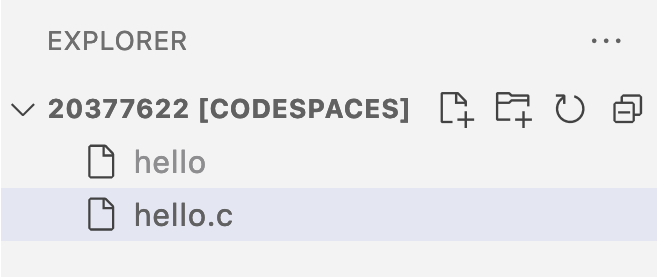


  - This terminal will be connected to our own virtual server, with its own operating system, set of files, and other installed programs that we access through the browser.
- The terminal provides a **command-line interface**, or CLI, and it allows us to access the virtual server's operating system, Linux (https://en.wikipedia.org/wiki/Linux).

- We'll run a command to compile our program, `make hello`. Nothing appears to happen, but we'll now have another file that's just called `hello`, which we can run with `./hello`:

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  DEBUG CONSOLE

$ make hello
$ ./hello
hello, world
$
```

  - `./hello` tells our computer to find a file in our current folder (`.`), called `hello`, and run it. And we indeed see the output that we expected.
- We'll open the sidebar and see that there are two files in our virtual server, one called `hello.c` (which we have open in our editor), and one called `hello`:



  - The `make hello` command created the `hello` file containing machine code.
  - The sidebar is a graphical user interface, or GUI, with which we can interact visually as we typically do.
- To delete a file, for example, we can right-click it in the sidebar and select the "Delete Permanently" option, but we can also use the terminal with the `rm` command:

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

$ make hello
$ ./hello
hello, world
$ rm hello
rm: remove regular file 'hello'? y
$
```

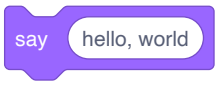
  - We run `rm hello` to remove the file called `hello`, and respond `y` for "yes" to confirm when prompted.
- We can also run the `ls` command to *list* files in our current folder. We'll compile our file again and run `ls` to see that a file called `hello` was created:

```
$ make hello
$ ls
hello*  hello.c
$
```

    - `hello` is in green with an asterisk, `*`, to indicate that it's executable, or that we can run it.
- Now, if we change our source code to read a different message, and run our program with `./hello`, we won't see the changes we made. We need to compile our code again, in order to create a new version of `hello` with machine code that we can run and see our changes in.
    - `make` is actually a program that finds and uses a compiler to create programs from our source code, and automatically names our program based on the name of the source code's file.

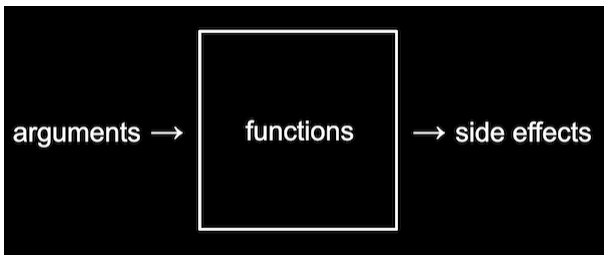
Functions, arguments, return values, variables

- Last time, we learned about functions, or actions, and arguments, or inputs to those functions that change what they do.
- The "say" block, for example, is closest to `printf` in C:

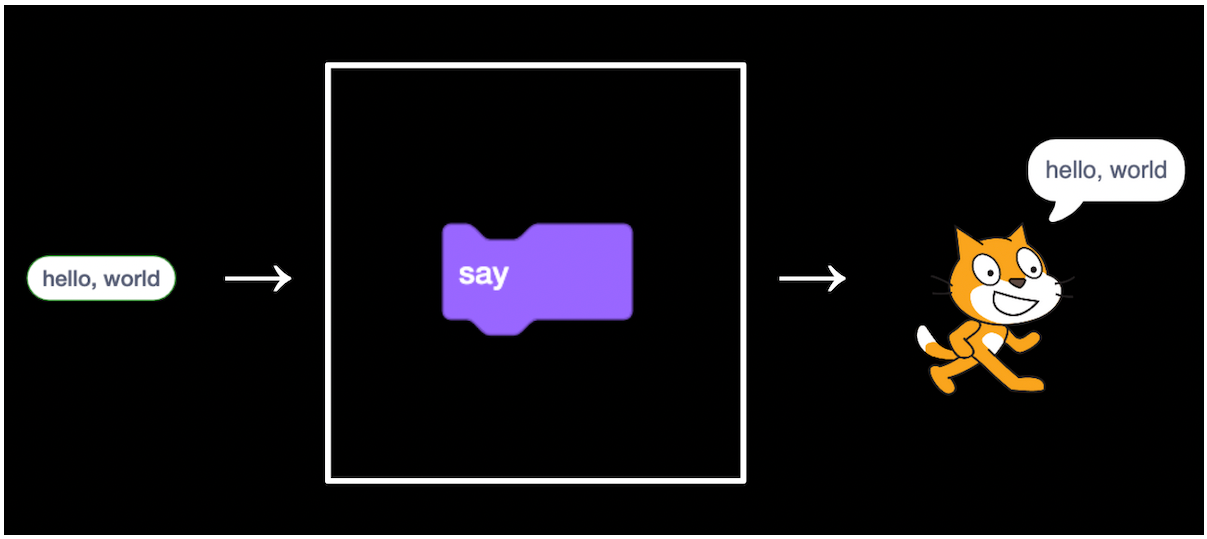



```
printf("hello, world");
```

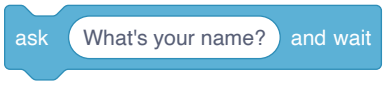
    - The `f` in `printf` refers to a "formatted" string, which we'll see again soon. And a **string** is a number of characters or words that we want to treat as text. In C, we need to surround strings with double quotes, `""`.
    - The parentheses, `()`, allow us to give an argument, or input, to our `printf` function.
    - Finally, we need a semicolon, `;`, to indicate the end of our statement or line of code.
- One type of output for a function is a **side effect**, or change that we can observe (like printing to the screen or playing a sound):



- In Scratch, the "say" block had a side effect:



- In contrast to side effects, we also saw blocks, or functions, with **return values** that we can use in our program. That return value might then be saved into a **variable**.
- In Scratch, the "ask" block, for example, stored an answer into the "answer" block:




```
string answer = get_string("What's your name? ");
```

  - In C, we have a function called `get_string()`, into which we pass the argument `"What's your name? "` as the prompt.
  - Then, we save the return value into a variable with `answer =`. Here, we're not asking whether the two sides are equal, but rather using `=` as the **assignment operator** to set the *left* side to the value on the *right*.
  - Finally, we need to indicate in C that `answer` is a variable with the **type** of `string`. Another type, `int`, is short for integer, or whole number. We'll see other types soon, but this is how our program will interpret different bytes.
    - If we try to set a value with a different type to a variable, the compiler will give us an error.
  - And just like learning a new human language, it might take weeks or months before we start automatically noticing these small details, like the semicolon. For some programming languages, the convention is to use all lowercase letters for variable and function names, but for others the conventional style might be different.
- We'll experiment again with our original program, this time removing the `\n` from the string we pass into `printf`:

```
#include <stdio.h>
```

```c
int main(void)
{
    printf("hello, world");
}
```

- And now, when we compile and run our program, we won't have the new line at the end of our message:

```
$ make hello
$ ./hello
hello, world$
```

- Let's try adding a new line within our string:

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world
    ");
}
```

- Our compiler will gives us back many errors:

```
$ make hello
hello.c:5:12: error: missing terminating '"' character [-Werror,-Winvali
    printf("hello, world
           ^
hello.c:5:12: error: expected expression
hello.c:6:5: error: missing terminating '"' character [-Werror,-Winvalid
    ");
    ^
hello.c:7:2: error: expected '}'
}
^
hello.c:4:1: note: to match this '{'
{
^
4 errors generated.
make: *** [<builtin>: hello] Error 1
```

    - Since many of these tools like compilers were originally written years ago, their error messages are concise and not as user-friendly as we'd like, but in this case it looks like we need to close our string with a `"` on the same line.

- When we use `\n` to create a new line, we're using an **escape sequence**, or a way to indicate a different expression within our string. In C (https://en.wikipedia.org/wiki/Escape_sequences_in_C), escape sequences start with a backslash, `\`.
- Now, let's try writing a program to get a string from the user:

```
#include <stdio.h>

int main(void)
{
    string answer = get_string("What's your name? ");
    printf("hello, answer\n");
}
```

  - We'll add a space instead of a new line after "What's your name?" so the user can type in their name on the same line.
- When we compile this with `make hello`, we get a lot of errors. We'll scroll up and focus on just the first error:

```
$ make hello
hello.c:5:5: error: use of undeclared identifier 'string'; did you mean
    string answer = get_string("What's your name? ");
    ^~~~~~
    stdin
/usr/include/stdio.h:137:14: note: 'stdin' declared here
extern FILE *stdin;             /* Standard input stream.  */
            ^
```

  - `hello.c:5:5` indicates that the error was found on line 5, character 5. It looks like `string` isn't defined.
- It turns out that, in order to use certain features or functions that don't come with C, we need to load libraries. A **library** is a common set of code, like extensions in Scratch, that we can reuse, and `stdio.h` refers to a library for standard input and output functions. With the line `#include <stdio.h>`, we're loading this library that contains `printf`, so that we can print to the screen.
- We'll need to also include `cs50.h`, a library written by CS50's staff, with helpful functions and definitions like `string` and `get_string`.
- We'll update our code to load the library ...

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
```

```
    string answer = get_string("What's your name? ");
    printf("hello, answer\n");
}
```

- ... and now our compiler works. But when we run our program, we see `hello, answer` printed literally:

```
$ make hello
$ ./hello
What's your name? David
hello, answer
$
```

- It turns out, we need to use a bit more syntax:

```
printf("hello, %s\n", answer);
```

  - With `%s`, we're adding a placeholder for `printf` to *format* our string. Then, outside our string, we pass in the variable as another argument with `answer`, separating it from the first argument with a comma, `,`.

- Text editors for programming languages will helpfully highlight, or color-code, different types of ideas in our code:

hello.c

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     string answer = get_string("What's your name? ");
7     printf("hello, %s\n", answer);
8 }
9
```

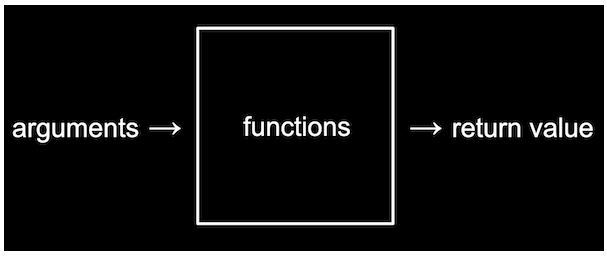
  - Now, it's easier for us to see the different components of our code and notice when we make a mistake.
  - Notice that on line 6, too, when our cursor is next to a parenthesis, the matching one is highlighted as well.
  - The four dots on lines 6 and 7 also help us see the number of spaces for indentation, helping us line up our code.

- We could also use the return value from `get_string` directly as an argument, as we might have done in Scratch with nested blocks:

```
#include <cs50.h>
#include <stdio.h>
```

```c
int main(void)
{
    printf("hello, %s\n", get_string("What's your name? "));
}
```

    - But we might consider this to be harder to read, and we aren't able to reuse the return value later.
- Both `get_string` in C and the "ask" block in Scratch are functions that have a return value as output:



- `printf("hello, %s\n", answer);` is also similar to these Scratch blocks:

    - We're placing a variable into our string, and displaying it right away.

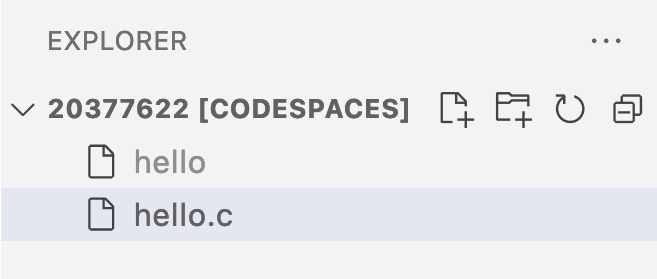
main, header files, commands

- In C, `main` achieves a similar effect as the Scratch block "when green flag clicked":


```c
int main(void)
{

}
```

    - The curly braces, `{` and `}`, surround the code that will run when our program is run as well.
- **Header files**, like `stdio.h`, tells our compiler which libraries to load into our program. `stdio.h` is like a menu of functions and features like `printf` that we can use in our code, though header files themselves don't include the actual implementation.
- In Linux, there are a number of commands we might use:
    - `cd`, for changing our current directory (folder)
    - `cp`, for copying files and directories

    - `ls`, for listing files in a directory
    - `mkdir`, for making a directory
    - `mv`, for moving (renaming) files and directories
    - `rm`, for removing (deleting) files
    - `rmdir`, for removing (deleting) directories
    - …
- In our cloud-based IDE, we're able to create new files and folders with the GUI in the sidebar:



- We can also use the terminal with:

```
$ mkdir pset1
$ mkdir pset2
$ ls
hello*  hello.c  pset1/  pset2/
$
```

    - We'll run `mkdir` twice, giving it the names of two folders we want to create. Then, we can run `ls` to see that our current directory has those folders.
- Now, we can run `cd` to change our current directory:

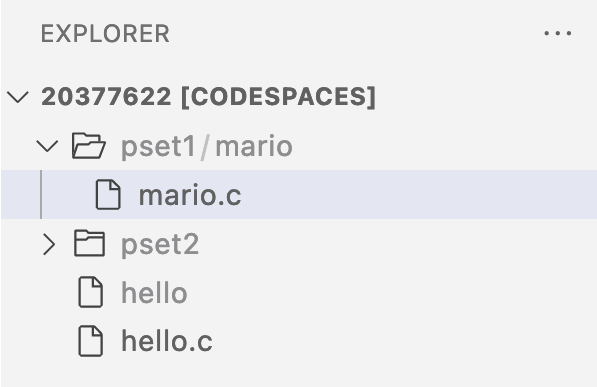
```
$ cd pset1/
pset1/ $ ls
pset1/ $
```

    - Notice that our prompt, `$`, changes to `pset1/ $` to remind us where we are. And `ls` shows that our current directory, now `pset1`, is empty.
- We can make yet another directory and change into it:

```
pset1/ $ mkdir mario
pset1/ $ ls
mario/
pset1/ $ cd mario/
pset1/mario/ $
```

- We'll run a command specific to VS Code, `code mario.c`, to create a new file called

`mario.c`. We see that it opens in the editor, and we can see our new folders and file in the sidebar as well:



- To change our current directory to the parent directory, we can run `cd ..`. We can go up two levels at once with `cd ../..` as well:

```
pset1/mario/ $ cd ..
pset1/ $ cd mario/
pset1/mario/ $ cd ../..
$
```

- `cd` on its own will also bring us back to our default directory:

```
pset1/mario/ $  cd
$
```

- And `.` refers to the current directory, which allows us to run a program `hello` in our current directory with `./hello`.

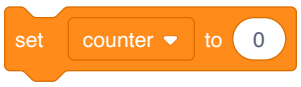
Types, format codes, operators

- There are many data **types** we can use for our variables, which indicate to our program what type of data they represent:
  - `bool`, a Boolean expression of either `true` or `false`
  - `char`, a single character like `a` or `2`
  - `double`, a floating-point value with more digits than a `float`
  - `float`, a floating-point value, or real number with a decimal value
  - `int`, integers up to a certain size, or number of bits
  - `long`, integers with more bits, so they can count higher than an `int`
  - `string`, a string of characters
  - ...
- And the CS50 Library has corresponding functions to get input of various types:

    - `get_char`
    - `get_double`
    - `get_float`
    - `get_int`
    - `get_long`
    - `get_string`
    - ...
- For `printf`, too, there are different placeholders for each type, called **format codes**:
    - `%c` for chars
    - `%f` for floats or doubles
    - `%i` for ints
    - `%li` for long integers
    - `%s` for strings
- There are several mathematical **operators** we can use, too:
    - `+` for addition
    - `-` for subtraction
    - `*` for multiplication
    - `/` for division
    - `%` for remainder

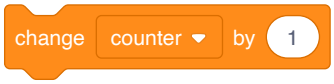
Variables, syntactic sugar

- We might create a variable called `counter` and set its value to `0` in Scratch and C with the following:




```
int counter = 0;
```

- And we can increase the value with:




```
counter = counter + 1;
```

    - In C, we're taking the original value of `counter`, adding 1, and then assigning it into the left side, or updating the value of `counter`.

    - We don't need to specify the type of `counter` again, since it's been created already.
- C also supports **syntactic sugar**, or shorthand expressions for the same functionality. We could equivalently say `counter += 1;` to add one to `counter` before storing it again. We could also just write `counter++;`, or even `counter--;` to subtract one.

Calculations

- Let's create a new file in our instance of VS Code with the command `code calculator.c` in our terminal. Then, we'll add in the following code to the editor that's opened for us, and save the file:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int x = get_int("x: ");
    int y = get_int("y: ");
    printf("%i\n", x + y);
}
```

    - We'll prompt the user for two variables, `x` and `y`, and print out the sum, `x + y`, with a placeholder for integers, `%i`.
    - These shorter variable names are fine in this case, since we're just using them as numbers without any other meaning.
- We can compile and run our program with:

```
$ make calculator
$ ./calculator
x: 1
y: 1
2
```

- We can change our program to use a third variable, `z`:

```
int z = x + y;
printf("%i\n", z);
```

    - This version gives us a reusable variable, but we might not intend on using the sum again in our program, so it might not necessarily be better.
- We can improve the style of our program with **comments**, notes to ourselves that the compiler ignores. Comments start with two slashes, `//`:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Prompt user for x
    int x = get_int("x: ");

    // Prompt user for y
    int y = get_int("y: ");

    // Perform addition
    printf("%i\n", x + y);
}
```

    - Since our program is fairly simple, these comments don't *add* too much, but as our programs get more complicated, we'll find these comments useful for reminding ourselves what and how our code is doing.
- In the terminal window, we can also start typing commands like `make ca`, and then press the `tab` key for the terminal to automatically complete our command. The up and down arrows also allow us to see previous commmands and run them without typing them again.
- We'll compile our program to make sure we haven't accidentally changed anything, since our comments should be ignored, and test it out:

```
$ make calculator
$ ./calculator
x: 1000000000
y: 1000000000
2000000000
$ ./calculator
x: 2000000000
y: 2000000000
-294967296
```

    - It turns out that data types each use a fixed number of bits to store their values. An `int` in our virtual environment uses 32 bits, which can only contain about four billion (2^{32}) different values. But since integers can be positive or negative, the highest positive value for an `int` can only be about two billion, with a lowest negative value of about negative two billion.
- We can change our program to store and display the result as a `long`, with more bits:

```c
#include <cs50.h>
```

```c
#include <stdio.h>

int main(void)
{
    // Prompt user for x
    long x = get_long("x: ");

    // Prompt user for y
    long y = get_long("y: ");

    // Perform addition
    printf("%li\n", x + y);
}
```

- But we could still have a value that's too large, which is a general problem we'll discuss again later.

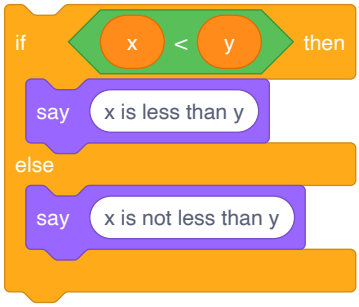
Conditionals, Boolean expressions

- In Scratch, we had conditional, or "if", blocks, like:

- In C, we similarly have:

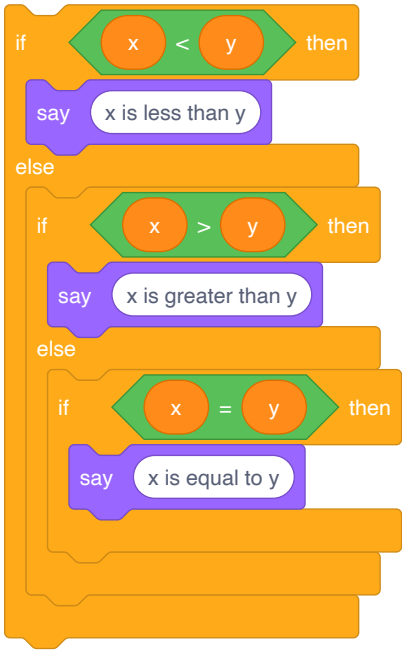
```c
if (x < y)
{
    printf("x is less than y");
}
```

  - Notice that in C, we use `{` and `}` (as well as indentation) to indicate how lines of code should be nested.
  - And even though `if` is followed by parentheses, it is not a function. We also don't use semicolons after the conditionals.
- We can have "if" and "else" conditions:




```c
if (x < y)
{
    printf("x is less than y\n");
}
else
{
    printf("x is not less than y\n");
}
```

- And in C, we can use “else if”:




```c
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else if (x == y)
{
    printf("x is equal to y\n");
```

```
}
```

- Notice that, to compare two values in C, we use two equals signs, `==`.
- And, logically, we don't need the `if (x == y)` in the final condition, since that's the only case remaining. Instead of asking three different questions, we can just ask two, and if both of the first cases are false, we can just say `else`:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

- Let's write another program. We'll start by running `code points.c` in our terminal window, and in the text editor, add:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int points = get_int("How many points did you lose? ");

    if (points < 2)
    {
        printf("You lost fewer points than me.\n");
    }
    else if (points > 2)
    {
        printf("You lost more points than me.\n");
    }
    else if (points == 2)
    {
        printf("You lost the same number of points as me.\n");
    }
}
```

- We'll run `make points`, and try it a few times:

```
$ make points
$ ./points
How many points did you lose? 1
You lost fewer points than me.
$ ./points
How many points did you lose? 0
You lost fewer points than me.
$ ./points
How many points did you lose? 3
You lost more points than me.
```

- But in our program, we've included the same **magic number**, or value that comes from somewhere unknown, in two places. Instead of comparing the number of points against `2` in both cases manually, we can create a **constant**, a variable that we aren't able to change:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    const int MINE = 2;
    int points = get_int("How many points did you lose? ");

    if (points < MINE)
    {
        printf("You lost fewer points than me.\n");
    }
    else if (points > MINE)
    {
        printf("You lost more points than me.\n");
    }
    else
    {
        printf("You lost the same number of points as me.\n");
    }
}
```

  - The `const` keyword tells our compiler to ensure that the value of this variable isn't changed, and by convention the name of the variable should be in all uppercase, `MINE` (to represent the number of my points).
  - By convention, too,

- We'll write another program called `parity.c` (https://cdn.cs50.net/2021/fall/lectures/1/src1/parity.c?highlight):

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int n = get_int("n: ");

    if (n % 2 == 0)
    {
        printf("even\n");
    }
    else
    {
        printf("odd\n");
    }
}
```

  - The `%` operator gives us the remainder of `n` after we divide it by `2`. If it is `0`, then `n` is an even number. Otherwise, it's an odd number.

- And we can make and test our program in the terminal:

```
$ make parity
$ ./parity
n: 2
even
$ ./parity
n: 4
even
$ ./parity
n: 3
odd
```

- We'll look at another program, `agree.c` (https://cdn.cs50.net/2021/fall/lectures/1/src1/agree.c?highlight):

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Prompt user to agree
    char c = get_char("Do you agree? ");

    // Check whether agreed
    if (c == 'Y' || c == 'y')
```

```
    {
        printf("Agreed.\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("Not agreed.\n");
    }
}
```

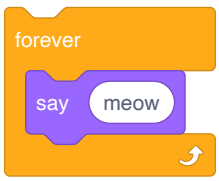
- First, we can get a single character, `char`, with `get_char()`. Then, we'll check whether the response is `Y` or `y`, *or* `N` or `n`. In C, we can ask two questions with "or", represented by two vertical bars, `||`, to check if at least one of them has an answer of true. (If we wanted to check that both questions have an answer of true, we would use "and", represented by ampersands, `&&`.)
- In C, a `char` is surrounded by single quotes, `'`, instead of double quotes for strings. (And strings with just a single character will still have double quotes, since they are a different data type.)

Loops, functions

- We'll write a program to print "meow" three times, as we did in Scratch:

```
#include <stdio.h>

int main(void)
{
    printf("meow\n");
    printf("meow\n");
    printf("meow\n");
}
```

  - But we could improve the design of our code with a loop.
- The "forever" block in Scratch can be recreated with a while loop in C:




```
while (true)
{
    printf("meow\n");
}
```

    - A `while` loop repeats over and over as long as the expression inside is true, and since `true` will always be true, this loop will repeat forever.
- We can also recreate the "repeat" block with a variable and a while loop:


```
int counter = 0;
while (counter < 3)
{
    printf("meow\n");
    counter = counter + 1;
}
```

    - We'll create a variable, `counter`, and set it to `0` at first. This will represent the number of times our loop has run.
    - Then, we'll have our while loop repeat as long as `counter` is less than `3`.
    - Each time our loop repeats, we'll print "meow" to the screen, and then increase the value of `counter` by one.
- We can simplify our loop slightly:

```
int i = 0;
while (i < 3)
{
    printf("meow\n");
    i++;
}
```

    - Since we're using the variable `counter` just as a mechanism for counting, we can use `i` as a conventional variable name.
    - We start `i` at `0` by convention as well, so by the time `i` reaches `3`, our loop will have repeated 3 times.
- It turns out that this is a common pattern, so in C we can use a for loop:

```
for (int i = 0; i < 3; i++)
{
    printf("meow\n");
}
```

    - The logic in the first line is the same as what we just saw in a while loop. First, a variable `i` is created and initialized to `0` with `int i = 0`. (Each of these pieces

are separated by a semicolon, just because of how the language was originally designed.) Then, the condition that is checked for every cycle of the loop is `i < 3`. Finally, *after* executing the code inside the loop, the last piece, `i++`, will be executed.

    - One minor difference with a for loop, compared to a while loop, is that the variable created within a for loop will only be accessible within the loop. In contrast, the variable `i` we created outside the while loop will still be accessible after the while loop finishes.

- We'll use a loop to "meow" three times in our program:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("meow\n");
    }
}
```

    - We can compile and run our program:

```
$ make meow
$ ./meow
meow
meow
meow
$
```

- Now we can start creating our own functions, like custom blocks in Scratch:

```
#include <stdio.h>

void meow(void)
{
    printf("meow\n");
}

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
```

```
}
```

  - We define our function with `void meow(void)`. The first `void` means that there isn't a return value for our function. The `void` within the parentheses also indicates that the function doesn't take any arguments, or inputs.
  - The lines of code in the curly braces that follow will be the code that runs every time our function is called.
- We can move our function to the bottom of our file, since we don't need to know how it's implemented right away:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}

void meow(void)
{
    printf("meow\n");
}
```

- But now, when we try to compile our program, we see some errors:

```
$ make meow
meow.c:7:11: error: implicit declaration of function 'meow' is invalid i
          meow();
          ^
meow.c:11:8: error: conflicting types for 'meow'
  void meow(void)
       ^
meow.c:7:11: note: previous implicit declaration is here
          meow();
          ^
2 errors generated.
make: *** [<builtin>: meow] Error 1
```

  - We'll start with the first one, and it turns out that our "implicit declaration", or use of the function without defining it first, is not allowed.
- The compiler reads our code from top to bottom, so it doesn't know what the `meow` function is. We can solve this by **declaring** our function with a **prototype**, which just tells

the compiler that we'll define our function later with the return type and argument type specified:

```
#include <stdio.h>

void meow(void);

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}

void meow(void)
{
    printf("meow\n");
}
```

- `void meow(void);` is our function's prototype. Notice that we don't actually write the implementation of the function until later in our code.

- We can add an argument to our `meow` function:

```
#include <stdio.h>

void meow(int n);

int main(void)
{
    meow(3);
}

void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

- With `void meow(int n)`, we're changing our function to take in some input, `n`, which will be an integer.
- Then, in our for loop, we can check `i < n` so that we repeat the loop the right

number of times.
    - Finally, in our `main` function, we can just call `meow`, giving it an input for the number of times we want to print "meow".
- Header files, ending in `.h`, include prototypes like `void meow(int n);`. Then, library files will include the actual implementation of each of those functions.
- We'll explore how our `main` function takes inputs and returns a value with `int main(void)` another day.

Mario

- Let's try to print out some blocks to the screen, like those from the video game Super Mario Bros (https://en.wikipedia.org/wiki/Super_Mario_Bros.). We'll start with printing four question marks, simulating blocks:

```
#include <stdio.h>

int main(void)
{
    printf("????\n");
}
```

- With a for loop, we can print any number of question marks with better design:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 4; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

    - After our for loop, we can print a new line. Then we can compile and run our program:

```
$ make mario
$ ./mario
????
$
```

- Let's get a positive integer from the user, and print out that number of question marks, by

using a **do while** loop:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int n;
    do
    {
        n = get_int("Width: ");
    }
    while (n < 1);

    for (int i = 0; i < n; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

- A do while loop does something first, and *then* checks whether the condition is true. If the condition is still true, then it repeats itself. Here, we're declaring an integer `n` without specifying a value. Then, we ask the user, with `get_int`, what the value of `n` should be. Finally, we repeat and ask the user for another input only if `n < 1`, since we want to print at least one question mark.
- We'll also change our for loop to use `n` as the number of times we print the question marks.
- We can compile and run our program:

```
$ make mario
$ ./mario
Width: 4
????
$ ./mario
Width: 40
????????????????????????????????????????
$
```

- And we can print a two-dimensional set of blocks with nested loops, or loops one inside the other:

```
#include <cs50.h>
#include <stdio.h>
```

```c
int main(void)
{
    int n;
    do
    {
        n = get_int("Size: ");
    }
    while (n < 1);

    // For each row
    for (int i = 0; i < n; i++)
    {
        // For each column
        for (int j = 0; j < n; j++)
        {
            // Print a brick
            printf("#");
        }

        // Move to next row
        printf("\n");
    }
}
```

- We have two nested loops, where the outer loop uses `i` to do some set of things `n` times. The inner loop uses `j` (another conventional variable for counting), a different variable, to do something `n` times for *each* of those times. In other words, the outer loop prints 3 rows, ending each of them with a new line, and the inner loop prints 3 bricks, or `#` characters, for each line:

  ```
  $ make mario
  $ ./mario
  Size: 3
  ###
  ###
  ###
  $
  ```

- We can stop a loop early as well. Instead of the do while loop from earlier, we can use a while loop:

```c
while (true)
{
    n = get_int("Size: ");
```

```
    if (n > 1)
    {
        break;
    }
}
```

- With `break`, we can break out of the while loop, which would otherwise repeat forever.

Imprecision, overflow

- Let's take a look at calculating values again, this time with floats and division:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Prompt user for x
    float x = get_float("x: ");

    // Prompt user for y
    float y = get_float("y: ");

    // Divide x by y
    float z = x / y;

    printf("%f\n", z);
}
```

- We can compile and test our program:

```
$ make calculator
$ ./calculator
x: 2
y: 3
0.666667
$ ./calculator
x: 1
y: 10
0.100000
$
```

- It turns out, with format codes like `%.2f`, we can specify the number of decimal places

displayed. Let's change our program to print fifty decimal places with `printf("%.50f\n", z);`. We'll compile and run our program again:

```
$ make calculator
$ ./calculator
x: 2
y: 3
0.66666668653488159179687500000000000000000000000000
$ ./calculator
x: 1
y: 10
0.10000000149011611938476562500000000000000000000000
```

    - Now, the values don't seem to be what we expect.
- It turns out that this is called **floating-point imprecision**, the inability for computers to represent all possible real numbers with a finite number of bits, like 32 bits for a `float`. So, our computer has to store the closest value it can, leading to imprecision.
    - In other languages, there are other ways to represent decimal values with more and more bits, though there is still a fundamental limit to the degree of accuracy.
- Similar, last week, when we had three bits and needed to count higher than seven (or `111`), we added another bit to represent eight with `1000`. But if we only had three bits available, the "next" number would be `000`, since we wouldn't have a place for the extra `1`. This problem is called **integer overflow**, where an integer can only be so large given a finite number of bits.
- The Y2K problem arose because many programs stored the calendar year with just two digits, like `98` for 1998, and `99` for 1999. But when the year 2000 approached, the programs had to store only `00`, leading to confusion between the years 1900 and 2000.
- In 2038, we'll also run out of bits to track time, since many years ago some humans decided to use 32 bits as the standard number of bits to count the number of seconds since January 1st, 1970. But since a 32-bit integer can only count up to about two billion, in 2038 we'll also reach that limit.
    - The 32 bits of an integer representing 2147483647 look like:

      ```
      01111111111111111111111111111111
      ```

    - When we increase that by 1, the bits will actually look like:

      ```
      10000000000000000000000000000000
      ```

    - But the first bit in an integer represents whether or not it's a negative value, so the decimal value will actually be -2147483648, the lowest possible *negative* value of an `int`. So computers might actually think it's sometime in 1901.

- Fortunately, we have more hardware these days, so we can start allocating more and more bits to store higher and higher values.
- We'll see one last example:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    float amount = get_float("Dollar Amount: ");
    int pennies = amount * 100;
    printf("Pennies: %i\n", pennies);
}
```

  - We'll compile and run our program:

```
$ make pennies
$ ./pennies
Dollar Amount: .99
Pennies: 99
$ ./pennies
Dollar Amount: 1.23
Pennies: 123
$ ./pennies
Dollar Amount: 4.20
Pennies: 419
```

- It turns out that there's imprecision in storing the `float` we get from the user (`4.20` might be stored as `4.199999...`), and so when we multiply it and display it as an integer, we see `419`.
- We can try to solve this by rounding:

```
#include <cs50.h>
#include <math.h>
#include <stdio.h>

int main(void)
{
    float amount = get_float("Dollar Amount: ");
    int pennies = round(amount * 100);
    printf("Pennies: %i\n", pennies);
}
```

  - `math.h` is another library that allows us to `round` numbers.

- Unfortunately, these bugs and mistakes happen all the time. For example, in the past some airplane's software needed to be restarted every 248 days, since one of its counters for time was overflowing as well.
- See you next time!

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 2

Compiling

- Last time, we learned to write our first program in C, printing "hello, world" to the screen:

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- We compiled it with `make hello` first, turning our source code into machine code before we could run the compiled program with `./hello`.
- `make` is actually just a program that calls `clang`, a compiler named for the "C language".
- We can compile our source code, `hello.c`, directly by running the command `clang hello.c` in the terminal:

```
$ clang hello.c
$
```

- Nothing seems to happen, which means there were no errors. And if we run list, with `ls`, we see an `a.out` file in our directory. The filename is just the default for `clang`'s output, and we can run it:

```
$ ./a.out
hello, world
```

- We can add a **command-line argument**, or an input to a program on the command-line as extra words after the program's name. We can run `clang -o hello hello.c`, where `clang` is the name of the program, and `-o hello` and `hello.c` are additional arguments. We're telling `clang` to use `hello` as the output filename, and use `hello.c` as the source code. Now, we can see `hello` being created as output.
- Let's try to compile the second version of last week's program, where we ask for a name:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

```
$ clang hello.c
/usr/bin/ld: /tmp/hello-733e0f.o: in function `main':
hello.c:(.text+0x19): undefined reference to `get_string'
clang: error: linker command failed with exit code 1 (use -v to see invo
```

- It looks like `clang` can't find `get_string`, but we did include `cs50.h`. It turns out that we also tell our compiler to actually link in the `cs50` machine code with the functions described by `cs50.h`. We'll have to add another argument with:

```
$ clang -o hello hello.c -lcs50
```

- With `make`, these commands are automatically generated for us.
- Compiling source code into machine code is actually made up of four smaller steps:
  - preprocessing
  - compiling
  - assembling
  - linking
- **Preprocessing** involves replacing lines that start with a `#`, like `#include`. For example, `#include <cs50.h>` will tell `clang` to look for that header file, since it contains content, like prototypes of functions, that we want to include in our program. Then, `clang` will essentially copy and paste the contents of those header files into our program.
- For example ...

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

- ... will be preprocessed into:

```
...
string get_string(string prompt);
...
int printf(string format, ...);
...

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

  - `string get_string(string prompt);` is a prototype of a function from `cs50.h`

that we want to use. The function is called `get_string`, and it takes in a `string` as an argument, called `prompt`, and returns a value of the type `string`.

- `int printf(string format, ...);` is a prototype from `stdio.h`, taking in a number of arguments, including a `string` for `format`.

- **Compiling** takes our source code, in C, and converts it to another language called **assembly language**, which looks like this:

```
...
main:                           # @main
    .cfi_startproc
# BB#0:
    pushq    %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    xorl    %eax, %eax
    movl    %eax, %edi
    movabsq    $.L.str, %rsi
    movb    $0, %al
    callq    get_string
    movabsq    $.L.str.1, %rdi
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rsi
    movb    $0, %al
    callq    printf
    ...
```

  - Notice that we see some recognizable strings, like `main`, `get_string`, and `printf`. But the other lines are instructions for basic operations on memory and values, closer to the binary instructions that a computer's processor can directly understand.

- The next step is to take the code in assembly and translate it to binary by **assembling** it. The instructions in binary are machine code, which a computer's CPU can run directly.
- The last step is **linking**, where previously compiled versions of libraries that we included earlier, like `cs50.c`, are actually combined with the compiled binary of our program. In other words, linking is the process of combining all the machine code for `hello.c`, `cs50.c`, and `stdio.c` into our one binary file, `a.out` or `hello`.

- These steps have been abstracted away, or simplified, by `make`, so we can think of the entire process as compiling.

Debugging

- **Bugs** are mistakes in programs that cause them to behave differently than intended. And **debugging** is the process of finding and fixing those bugs.
- A physical bug was actually discovered many years ago inside a very large computer, the Harvard Mark II (https://en.wikipedia.org/wiki/Harvard_Mark_II):

- We'll write a new program to print three "bricks". We'll call it `buggy.c`:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 3; i++)
    {
        printf("#\n");
    }
}
```

    - This program compiles and runs, but we have a logical error since we see four bricks:

```
$ make buggy
$ ./buggy
#
#
#
#
```

- We might see the problem already, but if we didn't, we could add another `printf` temporarily:

```
#include <stdio.h>

int main(void)
```

```
{
    for (int i = 0; i <= 3; i++)
    {
        printf("i is %i\n", i);
        printf("#\n");
    }
}
```

- Now, we can see that `i` started at 0 and continued until it was 3, but we should have our `for` loop stop once it's at 3, with `i < 3` instead of `i <= 3`:

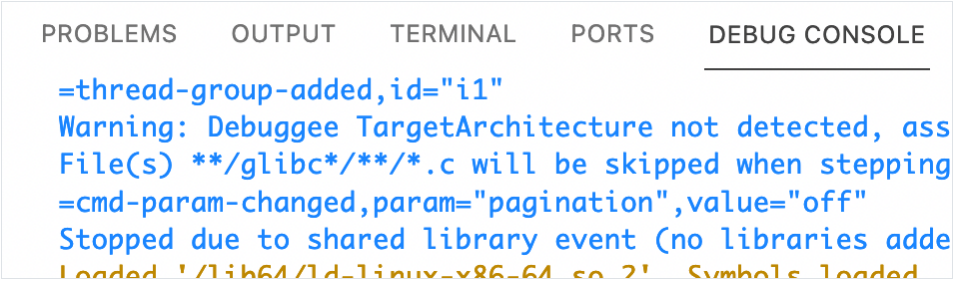
```
$ make buggy
$ ./buggy
i is 0
#
i is 1
#
i is 2
#
i is 3
#
```

- In our instance of VS Code, we have another command, **`debug50`**, to help us debug programs. This is a tool that runs the **debugger** built into VS Code, a program that will walk through our own programs step-by-step and let us look at variables and other information while our program is running.
- First, we'll click next to line 5 in our program to make a red dot appear:

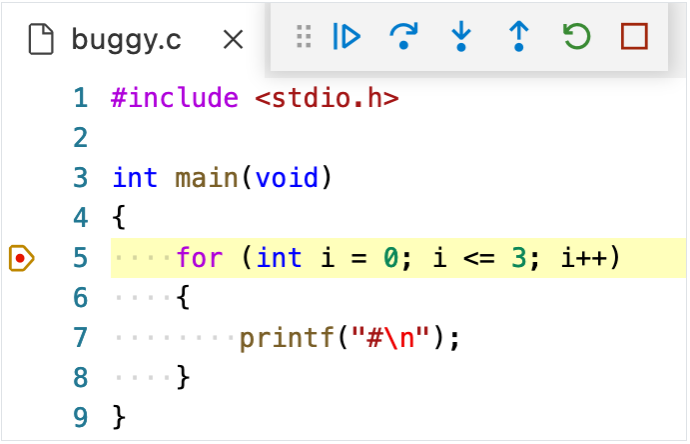
buggy.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for (int i = 0; i <= 3; i++)
6     {
7         printf("#\n");
8     }
9 }
```

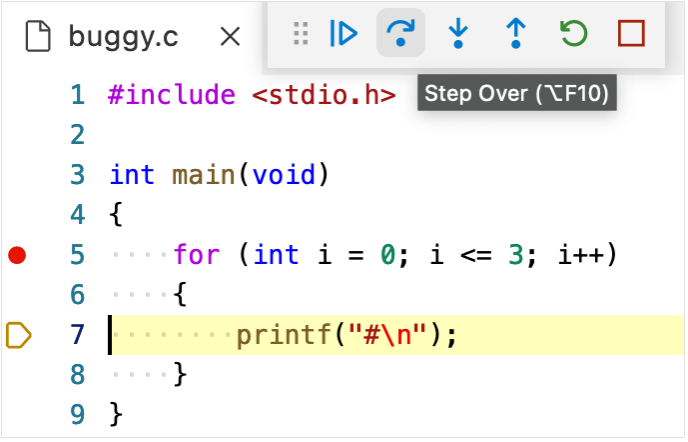
- Then, we'll run the command `debug50 ./buggy`, which might open a tab called Debug Console. We should go back to the tab labeled Terminal, so we can see our program's output still:



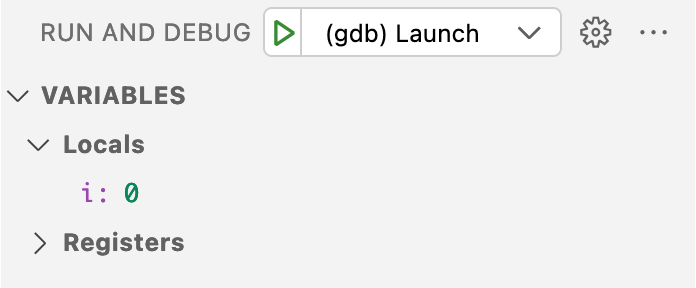
- The red dot we added was a **breakpoint**, a place where we want our debugger to pause our program's execution. Line 5 is now highlighted in yellow since it hasn't executed yet:



- The buttons that have appeared will allow us to control our program. We can hover over each of them to see their labels. The first button that looks like a play button will "continue" our program until the next breakpoint. The second button, which looks like a curved arrow, will "step over", or run the next line. We'll click that one:



- Notice that the panel on the left labeled Run and Debug has a section called Variables, where we can see their values:



    - `i` has a value of `0`.
- We'll click the Step Over button again, and now we see a single `#` printed to the screen. The value of `i` hasn't changed, since we're back to line 5, but that hasn't run yet.
- Now, we can click Step Over again, and we'll see the value of `i` change to `1` as we run line 5 and move to line 7.
- We can repeat this at our pace, and see our program's output and variables.
- To stop this process, we'll click the rightmost button that looks like a square, and we'll be brought back to our terminal. And we can click the red dot next to line 5 to remove the breakpoint as well.
- A third debugging technique is using a **rubber duck** (https://en.wikipedia.org/wiki/Rubber_duck_debugging), where we explain what we're trying to do in our code to a rubber duck (or other inanimate object). By talking through our own code out loud step-by-step, we might realize our mistake.
- Let's look at another buggy program:

```
#include <cs50.h>
#include <stdio.h>

int get_negative_int(void);

int main(void)
{
    int i = get_negative_int();
    printf("%i\n", i);
}

int get_negative_int(void)
{
    int n;
    do
    {
        n = get_int("Negative Integer: ");
    }
    while (n < 0);
    return n;
}
```

    - We've implemented another function, `get_negative_int`, to get a negative integer from the user. We'll use a do while loop to ask the user for an integer, storing it in `n`, and returning it.
    - We'll include the prototype at the top of our program for our compiler.

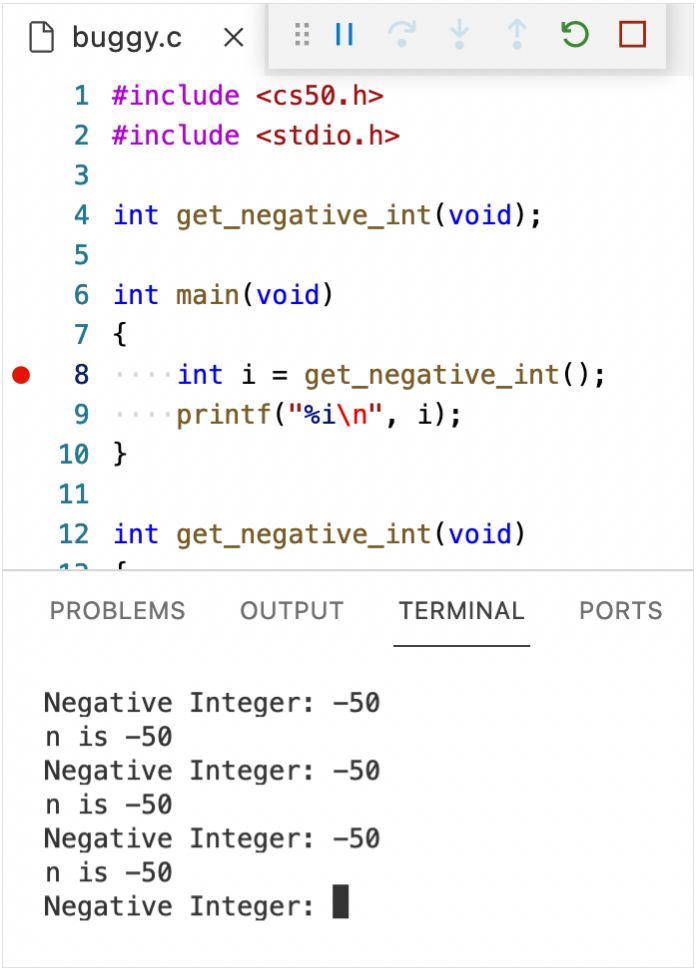
- But when we run our program, it keeps asking us for a negative integer:

```
$ make buggy
$ ./buggy
Negative Integer: -50
Negative Integer: -5
Negative Integer: 0
0
```

- We could add a line to print the value:

```
...
int get_negative_int(void)
{
    int n;
    do
    {
        n = get_int("Negative Integer: ");
        printf("n is %i\n", n);
    }
    while (n < 0);
    return n;
}
...
```

    - But that doesn't help us a lot.
- We'll set a breakpoint on line 8, `int i = get_negative_int();`, since it's the first interesting line of code. We'll run `debug50 ./buggy`, and click the "Step Over" button to run that line:

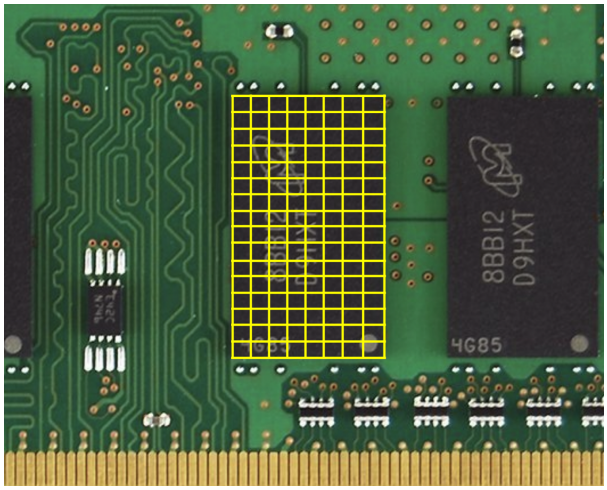


    - But now we are stuck in that line, and we can't really see what it's doing.
- We'll stop our debugger, and start it again. This time, we'll click the third button that looks like an arrow pointing down, "Step Into", which will let us go *into* the function called on that line. Now, the line `n = get_int("Negative Integer: ");` within the `get_negative_int` function is highlighted, and hopefully we can find the mistake in our code.

Memory

- In C, we have different types of variables we can use for storing data. Each variable is stored with a fixed number of bytes, and for most computer systems each type has the following size:
    - `bool`, 1 byte
        - A Boolean value can technically be represented with just a single bit, but for simplicity our computers use an entire byte.
    - `char`, 1 byte
        - Recall that with ASCII, we have a maximum of 256 different possible characters, since there are 8 bits in a byte.

    - `double`, 8 bytes
        - Twice as many bytes as a `float`.
    - `float`, 4 bytes
    - `int`, 4 bytes
        - Recall that a 32-bit integer can represent about 4 billion different values.
    - `long`, 8 bytes
        - Twice as many bytes as an `int`.
    - `string`, ? bytes
        - A `string` takes up a variable amount of space, since it could be short or long.
- Inside our computers, we have chips called RAM, random-access **memory**, that stores zeroes and ones. We can think of bytes stored in RAM as though they were in a grid, one after the other:



    - In reality, there are millions or billions of bytes per chip.
- A `char` which takes up one byte will take up one of those squares in RAM. An `int`, with 4 bytes, will take up four of those squares.
- Now, we can take for granted that bytes are stored in memory, knowing that we can work with them and build abstractions on top of them.

Arrays

- Let's say we wanted to take the average of three variables:

```
#include <stdio.h>

int main(void)
{
    int score1 = 72;
```

```
    int score2 = 73;
    int score3 = 33;

    printf("Average: %f\n", (score1 + score2 + score3) / 3);
}
```

- When we compile this program, we get:

```
$ make scores
scores.c:9:29: error: format specifies type 'double' but the argument ha
    printf("Average: %f\n", (score1 + score2 + score3) / 3);
                    ~~     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    %d
1 error generated.
make: *** [<builtin>: scores] Error 1
```

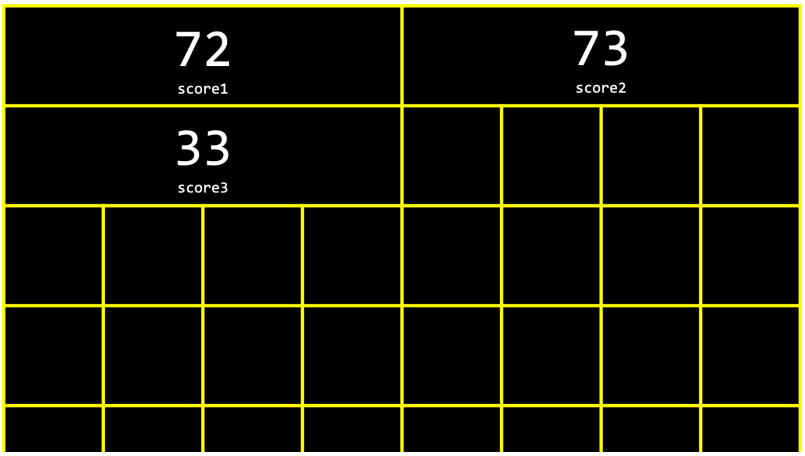
  - It turns out that, dividing three integers by another integer will result in an integer, with the remainder getting thrown away.

- We'll divide by not `3`, but `3.0` so the result is treated as a float:

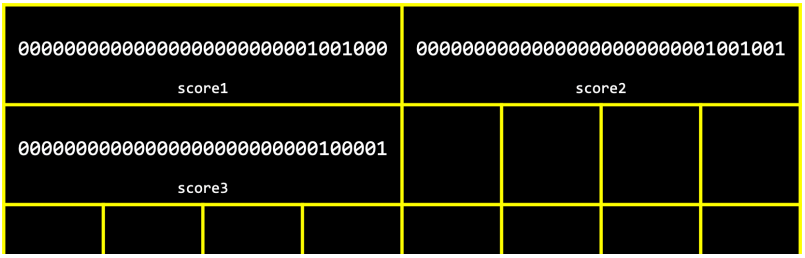
```
...
printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
...
```

```
$ make scores
$ ./scores
Average: 59.333333
```

- The design of our program isn't ideal, since we have just three variables, and we'd have to define more and more variables.
- While our program is running, the three `int` variables are stored in memory:



- Each `int` takes up four boxes, representing four bytes, and each byte in turn is made up of eight bits, 0s and 1s:



- It turns out we can refer to multiple variables with one name with another type called an **array**. With an array, we can store values of the same type, back-to-back, or contiguously.
- For example, in our program above, we can use `int scores[3];` to declare an array of three integers instead. The square brackets, `[3]`, indicates how many values we want to store, and `int` indicates the type of each value. We also name our array `scores` to indicate that it stores multiple scores.
- And we can assign and use variables in an array with `scores[0] = 72`. With the brackets, we're indexing into, or going to, the "0th" position in the array. Arrays are zero-indexed, meaning that the first value has index 0, and the second value has index 1, and so on.
- Let's update our program to use an array:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int scores[3];

    scores[0] = 72;
    scores[1] = 73;
    scores[2] = 33;

    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

  - Now, we're storing three `int`s in an array, and then accessing them again later to add them together.
- We'll ask the user for each score:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int scores[3];
```

```
    scores[0] = get_int("Score: ");
    scores[1] = get_int("Score: ");
    scores[2] = get_int("Score: ");

    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

- The design of our program could be improved, since we see three lines that are very similar, giving off a "code smell" (https://en.wikipedia.org/wiki/Code_smell) that indicates we could improve it somehow. Since we can set and access items in an array based on their position, and that position can *also* be the value of some variable, we can actually use a for loop:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int scores[3];

    for (int i = 0; i < 3; i++)
    {
      scores[i] = get_int("Score: ");
    }

    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

  - Now, instead of setting each value, we use a `for` loop and use `i` as the index of each item in the array.

- And we repeated the value 3, representing the length of our array, in two different places. So we can ask the user and use a variable, `n`, for the number of scores:

```
...
  int n = get_int("How many scores? ");

  int scores[n];

  for (int i = 0; i < n; i++)
  {
    scores[i] = get_int("Score: ");
  }
...
```

Characters

- We might have three characters we want to print:

```
#include <stdio.h>

int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

    printf("%c%c%c\n", c1, c2, c3);
}
```

```
$ make hi
$ ./hi
HI!
```

  - We can use `%c` to print out each character.
- Let's see what happens if we change our program to print `c` as an integer:

```
#include <stdio.h>

int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

    printf("%i %i %i\n", c1, c2, c3);
}
```

```
$ make hi
$ ./hi
72 73 33
```

  - It turns out that `printf` can print `char`s as integers, since each character is really stored as an ASCII value with zeroes and ones.
- We can explicitly convert `char`s to `int`s as well with:

```
printf("%i %i %i\n", (int) c1, (int) c2, (int) c3);
```

- When we convert a `float` to an `int`, however, we'll lose some information, like the values after the decimal point.

Strings

- We can see the same output as above by using a `string` variable:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

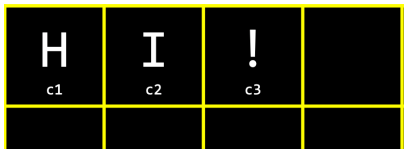
```
$ make hi
$ ./hi
HI!
```

- It turns out that **strings** are actually just arrays of characters, and defined not in C but by the CS50 library.
- Since our string is called `s`, which is just an array, each character can be accessed with `s[0]`, `s[1]`, and so on:

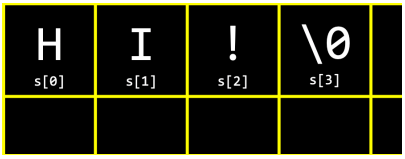
```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%i %i %i\n", s[0], s[1], s[2]);
}
```

```
$ make hi
$ ./hi
72 73 33
```

- In memory, our three `char` variables might have been stored like this:



- Each character in our string is stored in a byte of memory as well:



- In C, strings end with a special character, `'\0'`, or a byte with all eight bits set to 0, so our programs have a way of knowing where the string ends. This character is called the **null character**, or NUL. So, we actually need four bytes to store our string with three characters.
- Other data types we've seen so far don't need anything else at the end, since they are all set to a fixed size. Other languages or libraries might have custom types that represent numbers with a variable number of bytes as well, so there's more precision, but that might ultimately be implemented with recording how many bytes each number is.
- We can even print the last byte in our string to see that its value is indeed `0`:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);
}
```

```
$ make hi
$ ./hi
72 73 33 0
```

- Let's try taking a look at multiple strings:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    string t = "BYE!";

    printf("%s\n", s);
    printf("%s\n", t);
}
```

```
$ make hi
```

```
$ ./hi
HI!
BYE!
```

  - `s` takes up 4 bytes, and `t` takes up 5.

- We can write a program to determine the length of a string:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string name = get_string("Name: ");

    int i = 0;
    while (name[i] != '\0')
    {
        i++;
    }
    printf("%i\n", i);
}
```

```
$ make length
$ ./length
Name: HI!
3
$ ./length
Name: BYE!
4
$ ./length
Name: David
5
```

  - We'll get a `string` from the user, and use the variable `i` to get each character in the string. If the character isn't `\0`, then we know it's a character in the string, so we increment `i`. If the character is `\0`, then we've reached the end of the string and can stop the loop. Finally, we'll print the value of `i`.

- We can create a function to do this:

```
#include <cs50.h>
#include <stdio.h>

int string_length(string s);
```

```
int main(void)
{
    string name = get_string("Name: ");
    int length = string_length(name);
    printf("%i\n", length);
}

int string_length(string s)
{
    int i = 0;
    while (s[i] != '\0')
    {
        i++;
    }
    return i;
}
```

    - Our function, `string_length`, will take in a `string` as an argument, and return an `int`.
- We can use a function that comes with C's `string` library, `strlen`, to get the length of the string:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string name = get_string("Name: ");
    int length = strlen(name);
    printf("%i\n", length);
}
```

- We can use CS50's **manual pages** (https://manual.cs50.io/) to find and learn about libraries and functions. Standard libraries come with older documentation, called manual pages, that aren't always beginner-friendly. CS50's version includes simpler documentation for common functions.
- For example, we can search for "string" and see that `strlen` has a page (https://manual.cs50.io/3/strlen) that includes a synopsis, or summary, description, and example.
- Now that we can use `strlen`, we can try to print each character of our string with a loop:

```
#include <cs50.h>
#include <stdio.h>
```

```c
#include <string.h>

int main(void)
{
    string s = get_string("Input:  ");
    printf("Output: ");
    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c", s[i]);
    }
    printf("\n");
}
```

```
$ make string
$ ./string
Input:  HI!
Output: HI!
$ ./string
Input:  BYE!
Output: BYE!
```

- We can print each character of `s` with `s[i]`, and we can use a for loop since we know the length of the string, so we know when to stop.

- We can improve the design of this program. Our loop was a little inefficient, since we check the length of the string, after each character is printed, in our condition. But since the length of the string doesn't change, we can check the length of the string once:

```c
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Input:  ");
    printf("Output: \n");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c\n", s[i]);
    }
}
```

- Now, at the start of our for loop, we initialize both an `i` and `n` variable, and remember the length of our string in `n`. Then, we can check the values without having to call `strlen` to calculate the length of the string each time.

    - And we did need to use a little more memory to store `n`, but this saves us some time with not having to check the length of the string each time.
- We can now combine what we've seen, to write a program that can capitalize letters:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            printf("%c", s[i] - 32);
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

```
$ make uppercase
$ ./uppercase
Before: hi
After:  HI
$ ./uppercase
Before: david
After:  DAVID
```

    - First, we get a string `s` from the user. Then, for each character in the string, if it's lowercase (which means it has a value between that of `a` and `z`, inclusive), we convert it to uppercase. Otherwise, we just print it.
    - We can convert a lowercase letter to its uppercase equivalent by subtracting the difference between their ASCII values, which is `32` between `a` and `A`, and `b` and `B`, and so on.
- We can search in the manual pages for "lowercase", and it looks like there's another library, `ctype.h`, that we can use:

```c
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (islower(s[i]))
        {
            printf("%c", s[i] - 32);
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

    - Based on the manual pages, `islower` (https://manual.cs50.io/3/islower) will return a non-zero value if `c`, the character passed in, is lowercase. And a non-zero value is treated as `true` in Boolean expressions. (`0` is equivalent to `false`.)
- We can simplify even further, and just pass in each character to another function `toupper` (https://manual.cs50.io/3/toupper), since it capitalizes lowercase characters for us, and returns non-lowercase characters as they originally were:

```c
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c", toupper(s[i]));
    }
    printf("\n");
```

```
}
```

Command-line arguments

- Programs of our own can also take in command-line arguments, or inputs given to our program in the command we use to run it.
- We can change what our `main` function to no longer take in `void`, or no arguments, and instead:

```
#include <stdio.h>

int main(int argc, string argv[])
{
  ...
}
```

- `argc` and `argv` are two variables that our `main` function will now get automatically when our program is run from the command line. `argc` is the *argument count*, or number of arguments (words) typed in. `argv[]`, *argument vector* (or argument list), is an array of the arguments (words) themselves, and there's no size specified since we don't know how big that will be ahead of time.
- Let's try to print arguments:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    printf("hello, %s\n", argv[0]);
}
```

```
$ make argv
$ ./argv David
hello, ./argv
```

  - The first argument, `argv[0]`, is the name of our program (the first word typed, like `./hello`).
- We'll change our program to print the argument we want:

```
#include <cs50.h>
#include <stdio.h>
```

```
int main(int argc, string argv[])
{
    printf("hello, %s\n", argv[1]);
}
```

```
$ make argv
$ ./argv
hello, (null)
$ ./argv David
hello, David
```

- When we run our program without a second argument, we see `(null)` printed.

- We should make sure that we have the right number of arguments before we try to print something that isn't there:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

```
$ make argv
$ ./argv
hello, world
$ ./argv David
hello, David
$ ./argv David Malan
hello, world
```

- Now, we'll always print something valid, though we'll have to update our program to support more than two arguments.

- With command-line arguments, we can run our programs with input more easily and quickly.
- It turns out that our `main` function also returns an integer value called an **exit status**. By default, our `main` function returns `0` to indicate nothing went wrong, but we can write a

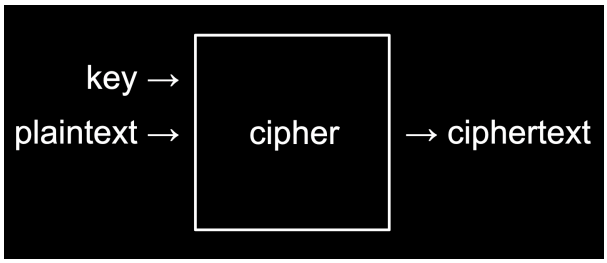
program to return a different value:

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc != 2)
    {
        printf("missing command-line argument\n");
        return 1;
    }
    printf("hello, %s\n", argv[1]);
    return 0;
}
```

- A non-zero exit status indicates some error to the system that runs our program. Once we run `return 1;` our program will exit early with an exit status of `1`. We might have seen error codes in the past when programs we used encountered errors as well.
- We'll write `return 0` explicitly at the end of our program here, even though we don't technically need to since C will automatically return `0` for us.

Applications

- We can consider phrases text and their level of readability, based on factors like how long and complicated the words and sentences are.
- For example, "One fish. Two fish. Red fish. Blue fish." is said to have a reading level of before grade 1.
- "Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much. They were the last people you'd expect to be involved in anything strange or mysterious, because they just didn't hold with such nonsense..." might be measured to have a level of grade 7.
- **Cryptography** is the art of scrambling information to hide its contents. If we wanted to send a message to someone, we might want to **encrypt**, or somehow scramble that message so that it would be hard for others to read. The original message, or input to our algorithm, is called **plaintext**, and the encrypted message, or output, is called **ciphertext**. And the algorithm that does the scrambling is called a **cipher**. A cipher generally requires another input in addition to the plaintext. A **key**, like a number, is some other input that is kept secret:



- For example, if we wanted to send a message like `HI!` with a key of `1`, we can use a simple cipher that rotates each letter forward by 1 to get `IJ!`.
- We could also use `-1` as the key for a message of `UIJT XBT DT50` to get `THIS WAS CS50` as the result.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

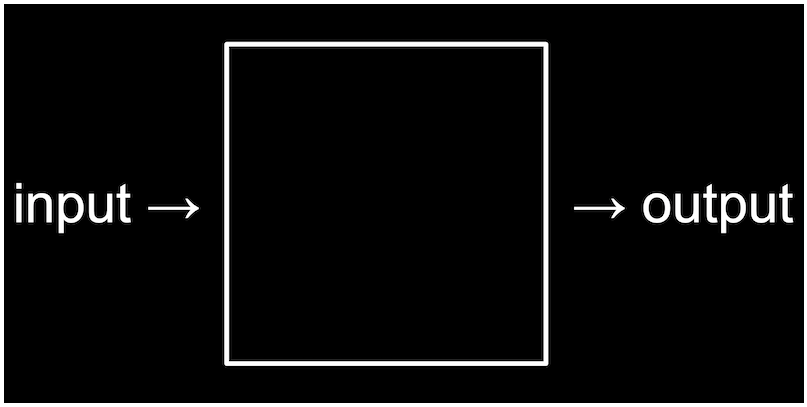
Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 3

Last week

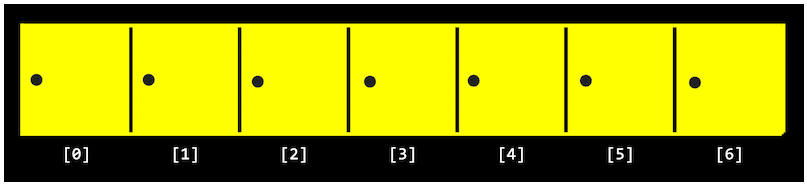
- Recall that our purpose for learning a new programming language and other tools is to solve problems. And we solve those problems by creating some output from input:



- We learned about memory, which allows us to store data as bytes, and strings, which are arrays of characters.

Searching

- Today we'll focus on algorithms that solve problems with arrays.
- It turns out that, with arrays, a computer can't look at all of the elements at once. Instead, a computer can only to look at them one at a time, though we can look in any order, like only being able to open one locker at a time:



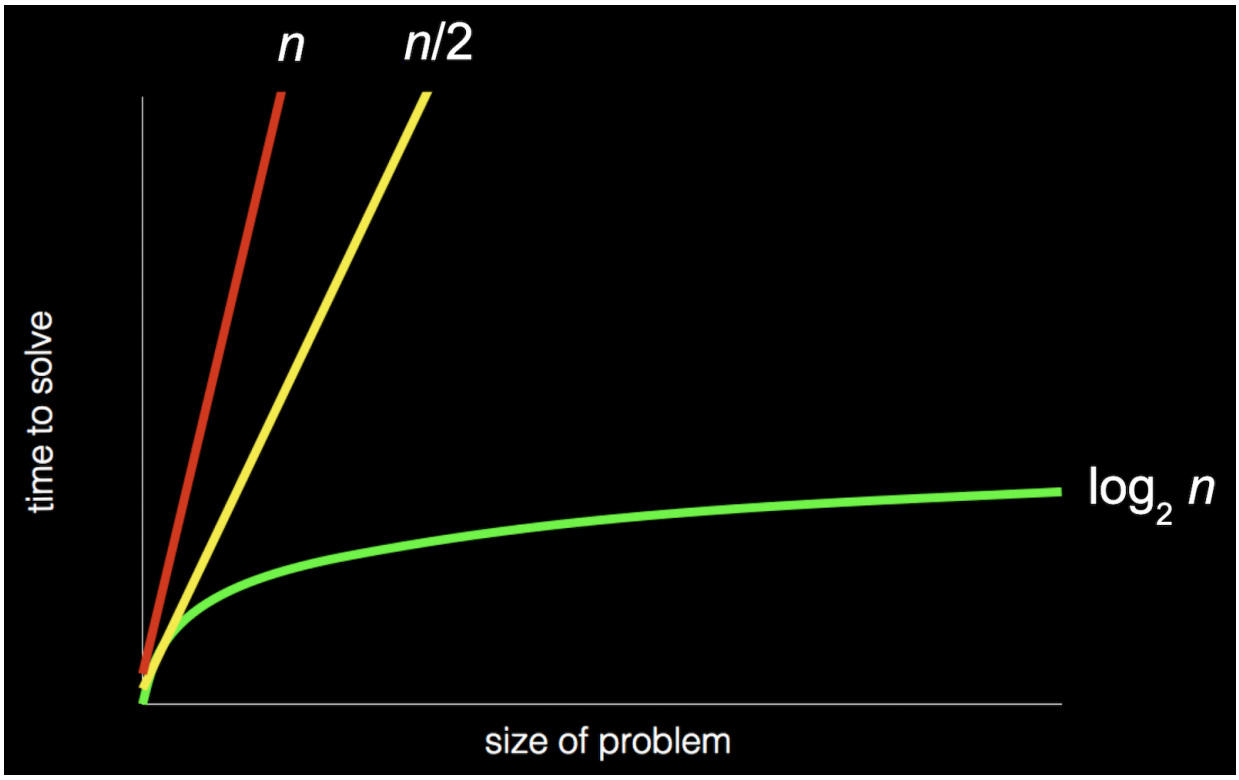
    - Recall that arrays are zero-indexed, meaning that the first item has an index of 0. And with n items, the highest index would be $n - 1$.
- **Searching** is how we solve the problem of finding information. A simple problem has an input of some values, and an output of a `bool`, whether or not a particular value is in the array.

Big O

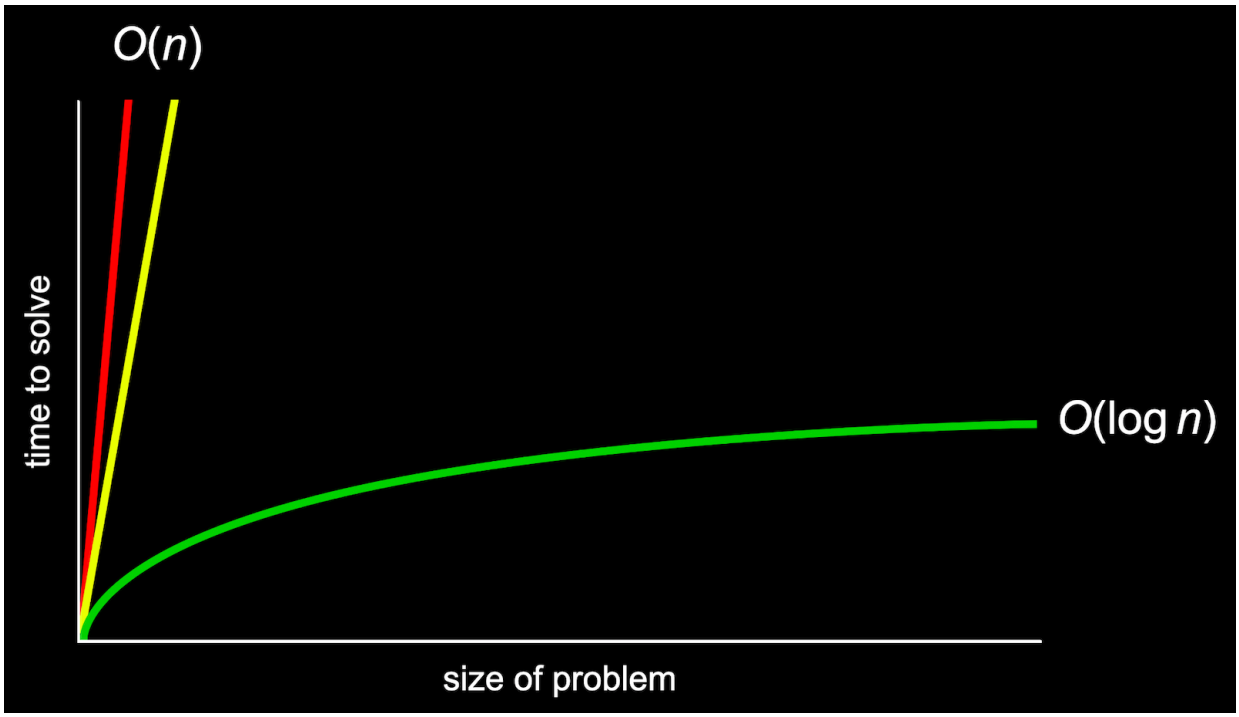
- Today we'll look at algorithms for searching. To compare their efficiency, we'll consider **running time**, or how long an algorithm takes to run given some size of input.
- Computer scientists tend to describe running time with **big O notation**, which we can think of as "on the order of" something, as though we want to convey an idea of running

time and not an exact number of milliseconds or steps.

- In week 0, we saw a chart with different types of algorithms and the times they might take to solve a problem:



  - Recall that the red line is searching linearly, one page at a time; the yellow line is searching two pages at a time; and the green line is dividing and conquering, starting in the middle and dividing the problem in half each time.
- In the chart above, if we zoomed out and changed the units on our axes, we would see the red and yellow lines end up very close together:



    - So we use big O notation to describe both the red and yellow lines, since they end up being very similar as n becomes larger and larger. We would describe them both as having "big O of n" or "on the order of n" running time.
    - The green line, though, is fundamentally different in its shape, even as n becomes very large, so it takes "big O of $\log n$" steps. (The base of the logarithm, 2, is also removed since it's a constant factor.)
- We'll see some common running times:
    - $O(n^2)$
    - $O(n \log n)$
    - $O(n)$
    - $O(\log n)$
    - $O(1)$
- Computer scientists might also use big Ω, big Omega notation, which describes the lower bound of number of steps for our algorithm, or how few steps it might take, in the best case. Big O, on the order of, is the upper bound of number of steps, or how many steps it might take, in the worst case.
- We have a similar set of the most common big Ω running times:
    - $\Omega(n^2)$
    - $\Omega(n \log n)$
    - $\Omega(n)$
    - $\Omega(\log n)$
    - $\Omega(1)$
- Finally, there is another notation, Θ, big Theta, which we use to describe running times of algorithms if the upper bound and lower bound is the same.
    - $\Theta(n^2)$
    - $\Theta(n \log n)$
    - $\Theta(n)$
    - $\Theta(\log n)$
    - $\Theta(1)$
- An algorithm with running time of $O(1)$ means that a constant number of steps is required, no matter how big the problem is.
- Let's take a look at some algorithms that we can describe with these running times.

Linear search, binary search

- On stage, we have seven lockers with closed doors, with numbers hidden behind them.

Since a computer can only look at one element in an array at a time, we can only open one door at a time as well.

- If we want to look for the number zero, for example, we would have to open one door at a time, and if we didn't know anything about the numbers behind the doors, the simplest algorithm would be going from left to right.
- This algorithm, **linear search**, would be correct but not very efficient. We might write pseudocode with:

```
For each door from left to right
    If number is behind door
        Return true
Return false
```

  - `Return false` is outside the for loop, since we only want to do that after we've looked behind all the doors.
- We can rewrite our pseudocode to be a little closer to C:

```
For i from 0 to n–1
    If number behind doors[i]
        Return true
Return false
```

  - Now, we're using a variable, `i`, to look at each location in an array called `doors`.
- With `n` doors, we'll need to look at all `n` of them. And what we do for each of the `n` doors, which is looking inside and possibly returning `true`, takes a constant number of steps each time. So the big O running time for this algorithm would be $O(n)$.
- The lower bound, big Omega, would be $\Omega(1)$, since we might be lucky and find the number we're looking for right away, which takes a constant number of steps.
- If we know that the numbers behind the doors are sorted, then we can start in the middle, and find our value more efficiently since we know we can go left or right, dividing the problem in half each time.
- For **binary search**, the pseudocode for our algorithm might look like:

```
If no doors
    Return false
If number behind middle door
    Return true
Else if number < middle door
    Search left half
Else if number > middle door
    Search right half
```

    - We remember to check whether there are no doors left, since that means our number isn't behind any of them.
- We can write this pseudocode to be more like C:

```
If no doors
    Return false
If number behind doors[middle]
    Return true
Else if number < doors[middle]
    Search doors[0] through doors[middle - 1]
Else if number > doors[middle]
    Search doors [middle + 1] through doors[n - 1]
```

    - We can determine the index of the middle door with a bit of math, and then we can divide the problem into searching either the doors with indices `0` through `middle - 1`, or `middle + 1` through `n - 1`.
- The upper bound for binary search is $O(\log n)$, since we might have to keep dividing the number of doors by two until there are no more doors left. The lower bound $\Omega(1)$, if the number we're looking for is in the middle, where we happen to start.
- Even though binary search might be much faster than linear search, it requires our array to be sorted first. If we're planning to search our data many times, it might be worth taking the time to sort it first, so we can use binary search.
- Other resources we might consider beyond the time it takes to run some code include the time it takes to write the code, or the amount of memory required for our code.

Searching with code

- Let's take a look at `numbers.c` (https://cdn.cs50.net/2021/fall/lectures/3/src3/numbers.c?highlight):

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int numbers[] = {4, 6, 8, 2, 7, 5, 0};

    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == 0)
        {
            printf("Found\n");
```

```
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

- Here we initialize an array with values by using curly braces, and we check the items in the array one at a time, in order, to see if they're equal to zero.
- If we find the value of zero, we return an exit code of 0 (to indicate success). Otherwise, *after* our for loop, we call `return 1` (to indicate an error code).
- This is how we might implement linear search.
- We can compile our program and run it to see that it works:

```
$ make numbers
$ ./numbers
Found
```

- And we can change what we're looking for to `-1`, and see that our program doesn't find it:

```
...
if (numbers[i] == -1)
...
```

```
$ make numbers
$ ./numbers
Not found
```

- We can do the same for strings in `names.c` (https://cdn.cs50.net/2021/fall/lectures/3/src3/names.c?highlight):

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string names[] = {"Bill", "Charlie", "Fred", "George", "Ginny", "Per

    for (int i = 0; i < 7; i++)
    {
        if (names[i] == "Ron")
        {
```

```
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

- But when we try to compile our program, we get:

```
$ make names
names.c:11:22: error: result of comparison against a string literal
        if (names[i] == "Ron")
                     ^  ~~~~~
1 error generated.
make: *** [<builtin>: names] Error 1
```

- It turns out that we can't compare strings directly in C, since they're not a simple data type built into the language, but rather an array of many characters. Luckily, the `string` library has function, `strcmp`, *string compare*, which compares strings for us. `strcmp` returns a negative value if the first string comes before the second string, `0` if the strings are the same, and a positive value if the first string comes after the second string.
- We'll change our conditional to `if (strcmp(names[i], "Ron") == 0)`, so we can check whether our two strings are actually equal.
- And if we wrote `if (strcmp(names[i], "Ron"))`, then any non-zero value, positive or negative, would be considered `true`, which would be the *opposite* of what we want.
    - We could actually write `if (!strcmp(names[i], "Ron"))` to invert the value, which would work in this case, but it would be arguably worse design since it doesn't explicitly check for the value of `0` as the documentation indicates.

Structs

- We might want to implement a phone book, with names and phone numbers, in `phonebook0.c` (https://cdn.cs50.net/2021/fall/lectures/3/src3/phonebook0.c?highlight):

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
    string names[] = {"Carter", "David"};
    string numbers[] = {"+1-617-495-1000", "+1-949-468-2750"};

    for (int i = 0; i < 2; i++)
    {
        if (strcmp(names[i], "David") == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - We have two arrays, `names` and `numbers`, and we'll make sure that each person's phone number has the same index as their name in each array.
  - We'll search our `names` array for someone's name, and then return their corresponding phone number at the same index.
  - This program is correct, but not well-designed since we'll have to maintain both arrays carefully to make sure that the names and numbers line up.
  - And feel free to call or text David for a surprise!
- It turns out in C that we can define our own data type, or **data structure**. It would be a better design for our program to have some array with a data type `person` that includes both their name and phone number, so we can just say `person people[];`.
- In C, we can create a **struct** with the following syntax:

```
typedef struct
{
    string name;
    string number;
}
person;
```

  - `typedef struct` tells the compiler that we're defining our own data structure. And `person` at the end of the curly braces will be the name of this data structure.
  - Inside our struct, we'll have two strings, `name` and `number`. We'll use strings for phone numbers since they might include punctuation, and other types of numbers, like zip codes, might have a leading 0 that would disappear if treated as a number.
- We'll use this in `phonebook1.c` (https://cdn.cs50.net/2021/fall/lectures/3/src3

/phonebook1.c?highlight):

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

typedef struct
{
    string name;
    string number;
}
person;

int main(void)
{
    person people[2];

    people[0].name = "Carter";
    people[0].number = "+1-617-495-1000";

    people[1].name = "David";
    people[1].number = "+1-949-468-2750";

    for (int i = 0; i < 2; i++)
    {
        if (strcmp(people[i].name, "David") == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

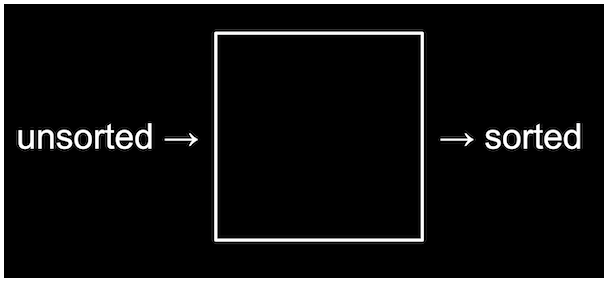
- We see new syntax to set the values for each variable inside each `person` struct by using the dot operator, `.`.
- In our loop, we can also use `.name` or `.number` to access variables in our structs, and be certain that they are from the same `person`.

- In computer science, **encapsulation** is the idea that related data is grouped together, and here we've encapsulated two pieces of information, `name` and `number` into the same data structure. The color of a pixel, with red, green, and blue values, might also be well-represented by a data structure as well.
- We can also imagine that a struct can be used to store precise decimal values or large

integer values, perhaps with arrays that we can use to store large numbers of digits.

Sorting

- **Sorting** is solving the problem which takes in an unsorted list of numbers as input, and producing an output of a sorted list of numbers:



  - For example, `6 3 8 5 2 7 4 1` might be input, and output would be `1 2 3 4 5 6 7 8`.

Selection sort demonstration

- We'll have eight volunteers come up to the stage, in unsorted order:

```
5 2 7 4 1 6 3 0
```

- We ask our volunteers to sort themselves, and everyone moves to the correct position:

```
0 1 2 3 4 5 6 7
```

- Unfortunately, computers can only look at one number and move one of them at a time, at least with programs we have written so far.
- We'll start with unsorted numbers again, and going step-by-step, we'll look for the smallest number first, looking at each number in order:

```
5 2 7 4 1 6 3 0
              ^
```

- Now, we'll swap the smallest number with the number at the beginning, since it's easier than shifting all of the other numbers down:

```
0 | 2 7 4 1 6 3 5
```

- Now, our problem has gotten smaller, since we know at least the beginning of our list is sorted. So we can repeat what we did, starting from the second number in the list:

```
0 | 2 7 4 1 6 3 5
```

```
          ^
0 | 1 7 4 2 6 3 5
```

- `1` is the smallest number now, so we'll swap it with the second number.

- We'll repeat this again …

```
0 1 | 7 4 2 6 3 5
          ^
0 1 | 2 4 7 6 3 5
```

- … and again …

```
0 1 2 | 4 7 6 3 5
              ^
0 1 2 | 3 7 6 4 5
```

- … and again …

```
0 1 2 3 | 7 6 4 5
              ^
0 1 2 3 | 4 6 7 5
```

- … and again …

```
0 1 2 3 4 | 6 7 5
                ^
0 1 2 3 4 | 5 7 6
```

- … and again, until we've swapped the last number in our list:

```
0 1 2 3 4 5 | 7 6
                  ^
0 1 2 3 4 5 | 6 7
```

Bubble sort demonstration

- We'll start with our unsorted list, but this time we'll look at pairs of numbers and swap them if they are out of order:

```
5 2 7 4 1 6 3 0
^ ^
2 5 7 4 1 6 3 0
  ^ ^
2 5 7 4 1 6 3 0
    ^ ^
```

```
2 5 4 7 1 6 3 0
      ^ ^
2 5 4 1 7 6 3 0
        ^ ^
2 5 4 1 6 7 3 0
          ^ ^
2 5 4 1 6 3 7 0
            ^ ^
2 5 4 1 6 3 0 7
```

- Now, the highest number is all the way to the right, so we've improved our problem.

- We'll repeat this again:

```
2 5 4 1 6 3 0 | 7
^ ^
2 5 4 1 6 3 0 | 7
  ^ ^
2 4 5 1 6 3 0 | 7
    ^ ^
2 4 1 5 6 3 0 | 7
      ^ ^
2 4 1 5 6 3 0 | 7
        ^ ^
2 4 1 5 3 6 0 | 7
          ^ ^
2 4 1 5 3 0 6 | 7
```

- Now the two biggest values are on the right.

- We'll repeat again ...

```
2 4 1 5 3 0 | 6 7
^ ^
2 4 1 5 3 0 | 6 7
  ^ ^
2 1 4 5 3 0 | 6 7
    ^ ^
2 1 4 5 3 0 | 6 7
      ^ ^
2 1 4 3 5 0 | 6 7
        ^ ^
2 1 4 3 0 5 | 6 7
```

- ... and again ...

```
2 1 4 3 0 | 5 6 7
```

```
^ ^
1 2 4 3 0 | 5 6 7
  ^ ^
1 2 3 4 0 | 5 6 7
    ^ ^
1 2 3 4 0 | 5 6 7
      ^ ^
1 2 3 0 4 | 5 6 7
```

- ... and again ...

```
1 2 3 0 | 4 5 6 7
^ ^
1 2 3 0 | 4 5 6 7
  ^ ^
1 2 3 0 | 4 5 6 7
    ^ ^
1 2 0 3 | 4 5 6 7
```

- ... and again ...

```
1 2 0 | 3 4 5 6 7
^ ^
1 2 0 | 3 4 5 6 7
  ^ ^
1 0 2 | 3 4 5 6 7
```

- ... and finally:

```
1 0 | 2 3 4 5 6 7
^ ^
0 1 | 2 3 4 5 6 7
```

- Notice that, as we go through our list, we know more and more of it becomes sorted, so we only need to look at the pairs of numbers that haven't been sorted yet.

Selection sort

- The first algorithm we saw is called **selection sort**, and we might write pseudocode like:

```
For i from 0 to n–1
    Find smallest number between numbers[i] and numbers[n–1]
    Swap smallest number with numbers[i]
```

  - We need to sort all `n` numbers in our list, so we'll have a loop with `i` to keep track of how many numbers we've sorted so far.

    - The first step in the loop is to look for the smallest number in the unsorted part of the list, from index `i` to `n-1`.
    - Then, we swap the smallest number we found with the number at index `i`, which makes everything up to `i` sorted.
    - And we'll repeat this until the entire list is sorted, from left to right, as `i` goes from `0` to `n-1`.
- For this algorithm, we started with looking at all n elements, then only $n - 1$, then $n - 2$, and so on:
$n + (n–1) + (n–2) + \ldots + 1$
$n(n + 1)/2$
$(n^2 + n)/2$
$n^2/2 + n/2$
$O(n^2)$
    - We can use some math formulas to get to $n^2/2 + n/2$ steps. Since n^2 is the biggest, or dominant, factor as n gets really large, we can say that the algorithm has an upper bound for running time of $O(n^2)$.
- In the best case, where the list is already sorted, our selection sort algorithm will still look at all the numbers and repeat the loop, so it has a lower bound for running time of $\Omega(n^2)$.

Bubble sort

- The pseudocode for bubble sort might look like:

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
```

    - Since we are comparing each pair of numbers at `i` and `i+1`, we only need to go up to `n-2` for `i`.
    - Then, we swap the two numbers if they're out of order.
    - We need to repeat this `n-1` times since each time we go over the list, only one number moves all the way to the right.
- To determine the running time for bubble sort, we have $n–1$ comparisons in the loop, and $n–1$ loops:
$(n - 1) \times (n - 1)$
$n^2 - 1n - 1n + 1$
$n^2 - 2n + 1$

$O(n^2)$

- The largest factor is again n^2 as n gets larger and larger, so we can say that bubble sort has an upper bound for running time of $O(n^2)$.
- If we add a little logic to our algorithm, we can optimize the running time in the best case:

```
Repeat n−1 times
    For i from 0 to n−2
        If numbers[i] and numbers[i+1] out of order
            Swap them
    If no swaps
        Quit
```

  - If we looked at all the numbers, and didn't make any swaps, then we know that our list has been sorted.
- The lower bound for running time of bubble sort would be $\Omega(n)$.
- We look at a visualization (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) with animations for how numbers are moved for selection sort and bubble sort.

Recursion

- **Recursion** is the ability for a function to call itself. We haven't seen this in code yet, but we've seen this in pseudocode for binary search:

```
If no doors
    Return false
If number behind middle door
    Return true
Else if number < middle door
    Search left half
Else if number > middle door
    Search right half
```

  - We're using the same "search" algorithm for each half. This seems like a cyclical process that will never end, but we're actually changing the input to the function and dividing the problem in half each time, stopping once there are no more doors left.
- Our pseudocode from week 0 used "go back" to repeat some steps, like a loop:

```
1  Pick up phone book
2  Open to middle of phone book
```

```
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

- But we can similarly have our algorithm use itself after it divides the problem:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Search left half of book
8  Else if person is later in book
9     Search right half of book
10 Else
11     Quit
```

- In week 1, too, we implemented a “pyramid” of blocks in the following shape:

```
#
##
###
####
```

- Our code to print this out might look like:

```c
#include <cs50.h>
#include <stdio.h>

void draw(int n);

int main(void)
{
    int height = get_int("Height: ");

    draw(height);
}
```

```
void draw(int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

- We have a `draw` function which takes in an argument, `n`, and uses a loop to print `n` rows with more and more bricks in each row.
- We can run our code and see that it works:

```
$ make iteration
$ ./iteration
Height: 4
#
##
###
####
```

- But notice that a pyramid of height 4 is actually a pyramid of height 3 with an extra row of 4 blocks added on. And a pyramid of height 3 is a pyramid of height 2 with an extra row of 3 blocks. A pyramid of height 2 is a pyramid of height 1 with an extra row of 2 blocks. And finally, a pyramid of height 1 is a pyramid of height 0 (no blocks) with a row of 1 block added.
- Since a pyramid is a recursive structure, we can write a recursive function to draw a pyramid, a function that calls itself to draw a smaller pyramid before adding another row:

```
#include <cs50.h>
#include <stdio.h>

void draw(int n);

int main(void)
{
    int height = get_int("Height: ");

    draw(height);
}
```

```c
void draw(int n)
{
    if (n <= 0)
    {
        return;
    }

    draw(n - 1);

    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- We can rewrite our `draw` function to use itself.
- If `n` is `0` (or negative somehow) we'll stop without printing anything. And we need to make sure we stop for some **base case**, so our function doesn't call itself over and over forever.
- Otherwise, we'll call `draw` again, to print a pyramid of size `n - 1` first.
- Then, we'll print the row of blocks we need for our pyramid, of size `n`.
- We can run our code and see that it works:

```
$ make recursion
$ ./recursion
Height: 4
#
##
###
####
```

- We can change our conditional to `if (n == 0)`, and type in a negative number to see what happens:

```
$ make recursion
$ ./recursion
Height: -100
Segmentation fault (core dumped)
```

- A segmentation fault means that we've touched memory in our computer that we shouldn't have, and this happened since our function has called itself over and over so many times and ended up using too much memory.

Merge sort

- We can take the idea of recusion to sorting, with another algorithm called **merge sort**. The pseudocode might look like:

```
If only one number
  Quit
Else
    Sort left half of number
    Sort right half of number
    Merge sorted halves
```

- Our strategy will be sorting each half, and then merge them together. Let's start with two sorted halves that look like:

```
2 4 5 7 | 0 1 3 6
```

  - We'll look at just the first number in each list, and move the smaller, `0`, into the start of a new list. Then we can look at the next number in the list:

```
2 4 5 7 | 0 1 3 6
^         ^

2 4 5 7 |   1 3 6
^           ^
0
```

  - We'll move the next smallest number, `1`, down, and look at the next number in that list:

```
2 4 5 7 |     3 6
^             ^
0 1
```

  - This time, the `2` is the next smallest, so we'll move that down, and repeat this process:

```
  4 5 7 |     3 6
  ^           ^
0 1 2

  4 5 7 |       6
  ^             ^
0 1 2 3

    5 7 |       6
    ^           ^
0 1 2 3 4

      7 |       6
      ^         ^
0 1 2 3 4 5

      7 |
      ^
0 1 2 3 4 5 6

        |
0 1 2 3 4 5 6 7
```

- So this is how we would merge two sorted halves together.
- We'll start at the beginning now with an entirely unsorted list of numbers:

  ```
  5 2 7 4 1 6 3 0
  ```

  - We start by looking at the first half, which is a list of size 4:

    ```
    5 2 7 4
    ```

    - We'll have to sort the left half, which is a list of size 2:

      ```
      5 2
      ```

      - The left half is size 1, so we're done, and the right half is also size 1, so we can merge both halves together:

        ```
        2 5
        ```

    - Now we'll need to sort the right half:

      ```
      7 4
      ```

      - Again, we'll merge both of these halves together, by taking the smallest

element from each list

```
4 7
```

- Now we've sorted both halves, each of size 2, and can merge them together:

```
2 5 | 4 7
^     ^

  5 | 4 7
  ^   ^
2

  5 |   7
  ^     ^
2 4

    |   7
        ^
2 4 5

2 4 5 7
```

- We'll repeat this for the right half, another list of size 4:

```
1 6 3 0
```

  - Sorting the left half gives us `1 6`, and the right half merges to `0 3`.
  - We'll merge those with:

```
1 6 | 0 3
^     ^

1 6 |   3
^       ^
0

  6 |   3
  ^     ^
0 1

  6 |
  ^
0 1 3

0 1 3 6
```

    - Now, we have two sorted halves, `2 4 5 7` and `0 1 3 6`, which we can merge together as we saw above.
- Every time we merged two halves, we only needed to look at each number once. And we divided our list of 8 numbers three times, or $\log n$ times. We needed more memory to merge our new lists into, but the upper bound for running time for merge sort is only $O(n \log n)$. Since $\log n$ is less than n, $O(n \log n)$ is less than $O(n^2)$.
- The lower bound of our merge sort is still $\Omega(n \log n)$, since we have to do all the work even if the list is sorted. So merge sort also has $\Theta(n \log n)$.
- We look at a final visualization (https://www.youtube.com/watch?v=ZZuD6iUe3Pc) of sorting algorithms with a larger number of inputs, running at the same time.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

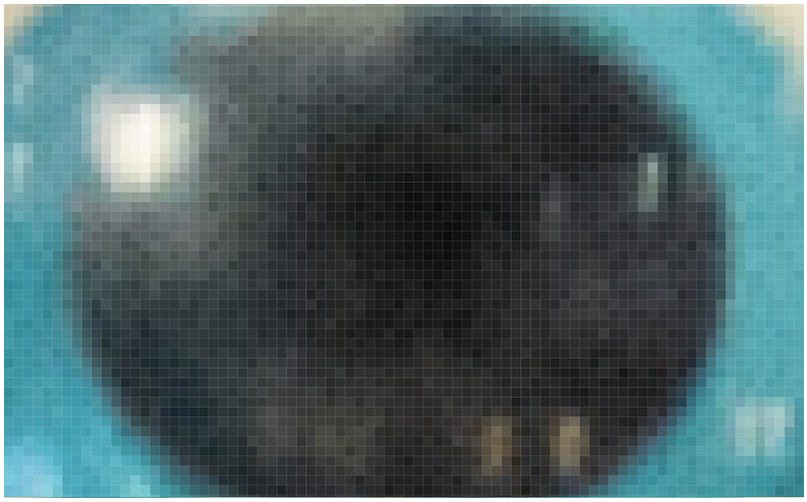
Lecture 4

- Pixels
- Hexadecimal
- Addresses, pointers
- Strings
- Pointer arithmetic
- Compare and copy
- Memory allocation
- valgrind
- Garbage values
- Swap
- Memory layout
- scanf
- Files

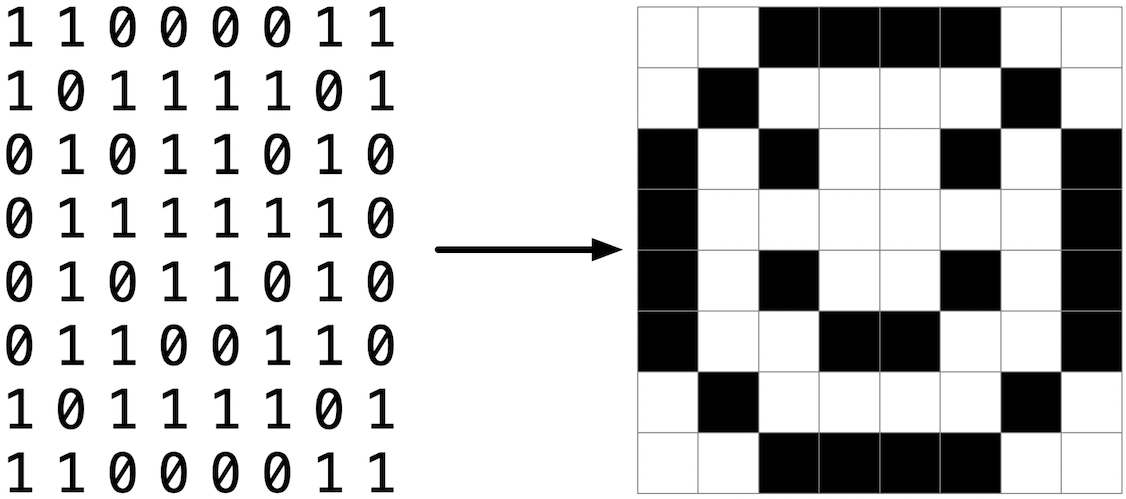
- JPEG

Pixels

- Last week, we took a look at memory and how we could use arrays to store data.
- We might zoom in further and further in an image, but very soon we'll see individual pixels, like the ones in this puppet's eye:

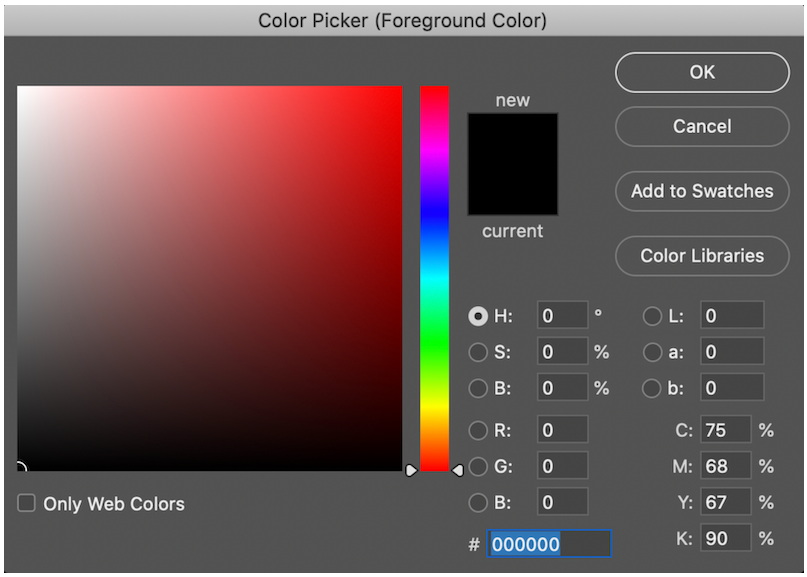


  - Since this image is stored with a finite number of bytes, each perhaps representing a red, green, or blue value for each pixel, there is in turn a finite number of pixels we can see.
- A simpler image of a smiley face might be represented with a single bit per pixel:



- We can visit cs50.ly/art (https://cs50.ly/art) with a Google Account to make a copy of a spreadsheet, which we can then fill with colors to create our own pixel art (https://en.wikipedia.org/wiki/Pixel_art).
- Adobe Photoshop (https://en.wikipedia.org/wiki/Adobe_Photoshop), a popular image

editing software, includes a color picker that looks like this:



- Here, the color black is selected, and we see that the values for R, G, and B, or red, green, and blue respectively, are all 0. Furthermore, we see another value, #000000 that seems to represent all three values for the color black.

- We take a look at a few more colors:
  - white, with R: 255, G: 255, and B: 255, and #FFFFFF
  - red, with R: 255, G: 0, and B: 0, and #FF0000
  - green, with R: 0, G: 255, and B: 0, and #00FF00
  - blue, with R: 0, G: 0, and B: 255, and #0000FF

Hexadecimal

- We might notice a pattern for the new notation, where it appears that each value for red, green, and blue are represented with two character. It turns out that there's another base system, **hexadecimal**, or base-16, where there are 16 digits:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

  - The digits start with 0 through 9, and continue with A through F as equivalents to the decimal values of 10 through 15.

- Let's consider a two-digit hexadecimal number:

```
16^1 16^0
  0    0
```

  - Here, the `0` in the ones place (since $16^0 = 1$) has a decimal value of 0. We'll count up, and after `09` we'll use `0A` to represent 10 in decimal.

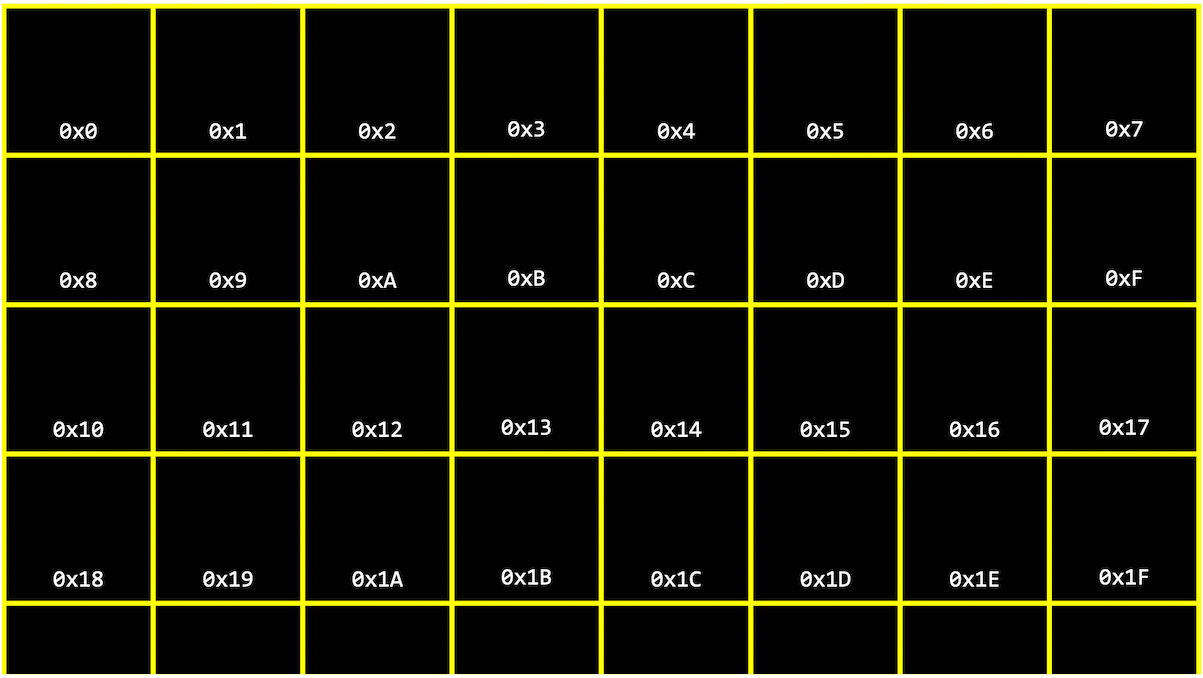
    - We can keep counting until `0F`, which is equivalent to 15 in decimal.
- After `0F`, we need to carry the one, as we would go from 9 to 10 in decimal:

```
16^1 16^0
   1    0
```

    - Here, the `1` has a value of $16^1 \times 1 = 16$, so `10` in hexadecimal is 16 in decimal.
- With two digits, we can have a maximum value of `FF`, or $16^1 \times 15 + 16^0 \times 15 = 16 \times 15 + 1 \times 15 = 240 + 15 = 255$.
    - The values in a computer's memory are still stored as binary, but this way of representation helps us humans represent larger numeric values with fewer digits needed.
    - With 8 bits in binary, the highest value we can count to is also 255, with `11111111`. So two digits in hexadecimal can conveniently represent the value of a byte in binary. (Each digit in hexadecimal, with 16 values, maps to four bits in binary.)

Addresses, pointers

- For our computer's memory, too, we'll see hexadecimal used to describe each address or location:

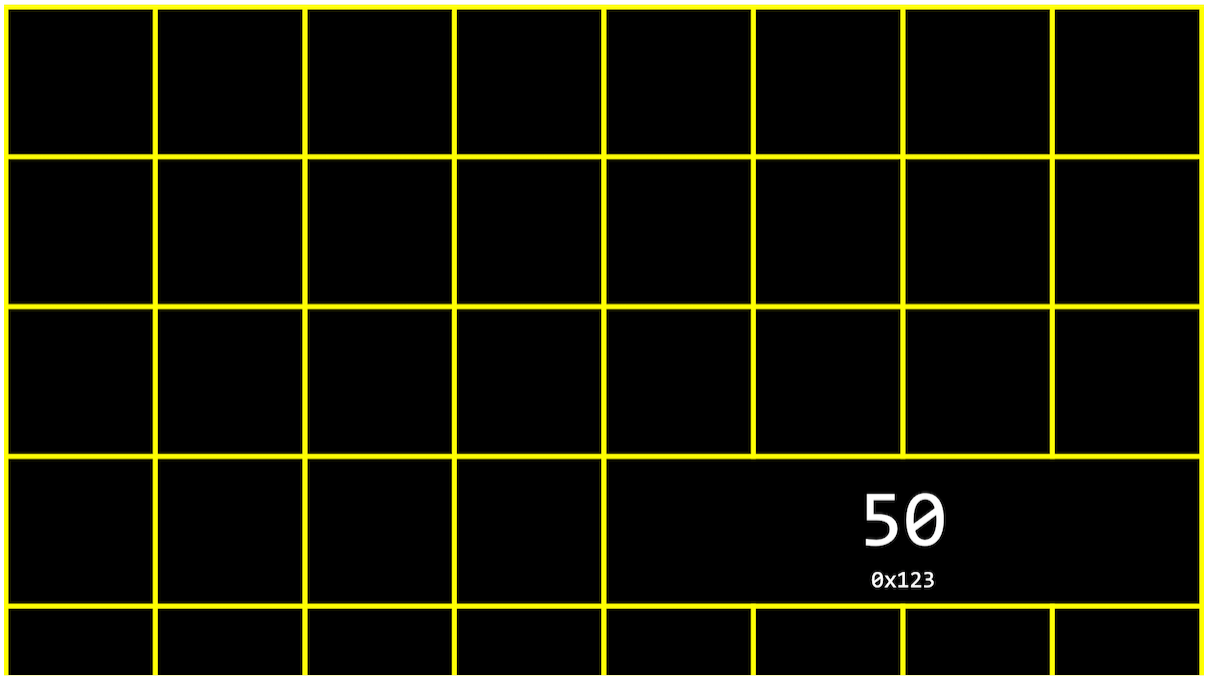


    - By writing `0x` in front of a hexadecimal value, we can distinguish them from decimal values.
- We might create a value `n`, and print it out:

```
#include <stdio.h>
```

```c
int main(void)
{
    int n = 50;
    printf("%i\n", n);
}
```

- In our computer's memory, there are now 4 bytes somewhere that have the binary value of 50, with some value for its address, like `0x123`:



- A **pointer** is a variable that stores an address in memory, where some other variable might be stored.
- The `&` operator can be used to get the address of some variable, as with `&n`. And the `*` operator declares a variable as a pointer, as with `int *p`, indicating that we have a variable called `p` that *points to* an `int`. So, to store the address of a variable `n` into a pointer `p`, we would write:

```c
int *p = &n;
```

- In C, we can actually see the address with the `&` operator, which means "get the address of this variable":

```c
#include <stdio.h>

int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%p\n", p);
```

```
}
```

```
$ make address
$ ./address
0x7ffcb4578e5c
```

- `%p` is the format code to print an address with `printf`. And we only need to use the name of the variable, `p`, after we've declared it.
- In our instance of VS Code, we see an address with a large value like `0x7ffcb4578e5c`. The value of the address in itself is not significant, since it's just some location in memory that the variable is stored in; instead, the important idea is that we can *use* this address later.
- We can run this program a few times, and see that the address of `n` in memory changes, since different addresses in memory will be available at different times.

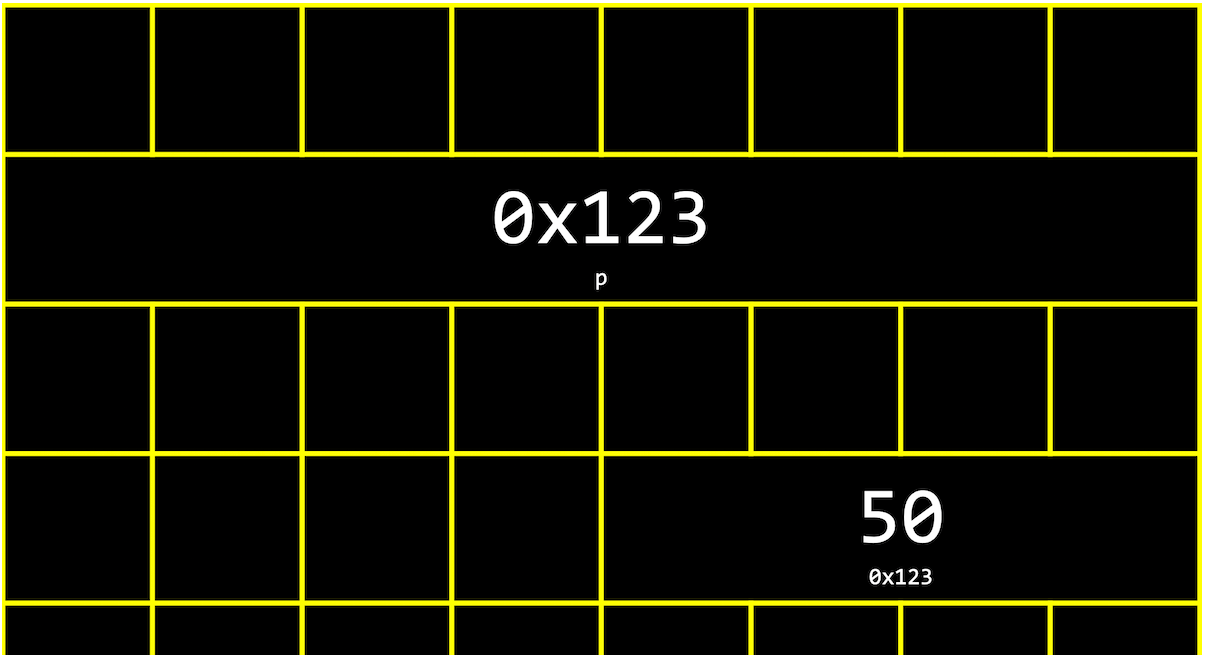
- With C, we can also go to specific addresses in memory, which might cause **segmentation faults**, where we've tried to read or write to memory we don't have permission to.
- The `*` operator is also the **dereference operator**, which goes to an address to get the value stored there. For example, we can say:

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%p\n", p);
    printf("%i\n", *p);
}
```

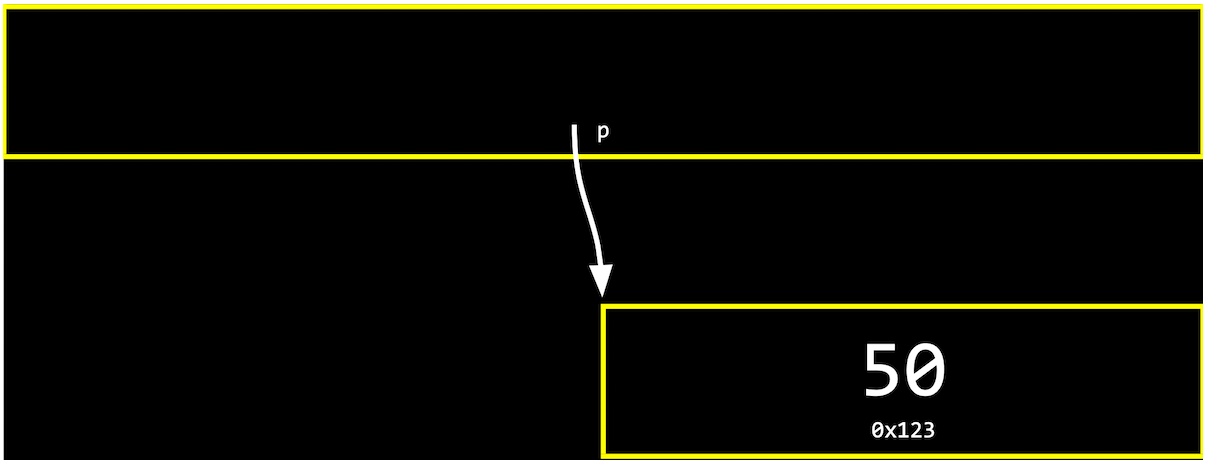
```
$ ./address
0x7ffda0a4767c
50
```

- Now, we see the value of the pointer itself (an address), and then the value at the address with `*p`, which is `50`.
- Since we declared `p` to be an `int *`, the compiler knows that `*p` is an `int`, so the right number of bytes are read.

- In memory, we might have one variable, `p`, with the value of some address, like `0x123`, stored, and another variable, an integer with the value `50`, at that addresss:



  - Notice that `p` takes up 8 bytes, since in modern computer systems, 64 bits are used in order to address the billions of bytes of memory available. With 32 bits, we can only count up to about 4 billion bytes, or about 4GB of memory.
- We can abstract away the actual value of the addresses, since they'll be different as we declare variables in our programs. We can simply think of `p` as pointing at some value in memory:

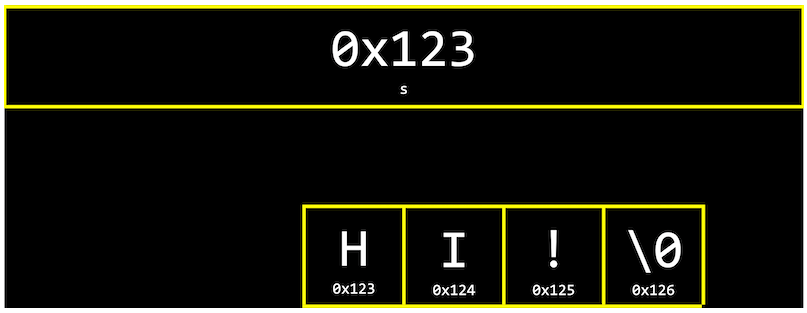


- In the real world, we might have a mailbox labeled "p", among many mailboxes with addresses. Inside our mailbox, we can put a value like `0x123`, which is the address of some other mailbox that's labeled "n".

Strings

- We can declare a string with `string s = "HI!";`, which will be stored one character at a time in memory. And we can access each character with `s[0]`, `s[1]`, `s[2]`, and `s[3]`:

- But it turns out that each character, since it's stored in memory, *also* has some unique address, and `s` is actually just a pointer with the address of the first character:



  - `s` is a variable of type `string`, which is a pointer to a character.
  - Recall that we can read the entire string by starting at the address in `s`, and continue reading one character at a time from memory until we reach `\0`.
- It turns out that `string s = "HI!"` is the same as `char *s = "HI!";`. And we can use strings in C in the exact same way without the CS50 Library, by using `char *`.
- Let's print out a string:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

```
$ make address
$ ./address
HI!
```

- Now, we can remove the CS50 Library, and say:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%s\n", s);
}
```

```
$ make address
```

```
$ ./address
HI!
```

- We can experiment and see the address of characters:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    char c = s[0];
    char *p = &c;
    printf("%p\n", s);
    printf("%p\n", p);
}
```

```
$ make address
$ ./address
0x402004
0x7ffd4227fdd7
```

  - We store the first character of `s` into `c`, and print out its address with `p`. We also print out `s` as an address with `%p`, and we see that the values are difference since we made a copy of the first character with `char c = s[0];`.

- Now, we'll print the address of the first character in `s`:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    char *p = &s[0];
    printf("%p\n", p);
    printf("%p\n", s);
}
```

```
$ make address
$ ./address
0x402004
0x402004
```

  - With `char *p = &s[0];`, we store the address of the first character in `s` into a pointer called `p`. And now, when we print `p` and `s` as addresses, we see the same

value.

- We can see the address of each character in `s`:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%p\n", s);
    printf("%p\n", &s[0]);
    printf("%p\n", &s[1]);
    printf("%p\n", &s[2]);
    printf("%p\n", &s[3]);
}
```

```
$ make address
$ ./address
0x402004
0x402004
0x402005
0x402006
0x402007
```

  - Again, the address of the first character, `&s[0]`, is the same as the value of `s`. And each following character has an address that is one byte higher.

- In the CS50 Library, a string is defined with just `typedef char *string;`. With `typedef`, we're creating a custom data type for the word `string`, making it equivalent to `char *`.

Pointer arithmetic

- We can print out each character in a string:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
    printf("%c\n", s[3]);
```

```
}
```

```
$ make address
$ ./address
H
I
!

$
```

- When we declare a `string` with double quotes, `"`, the compiler figures out where to put those characters in memory as an array.

- Let's simplify our code to use `char *` and show just the printable characters:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
}
```

```
$ make address
$ ./address
H
I
!
```

- But we can go to addresses directly:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", *s);
    printf("%c\n", *(s + 1));
    printf("%c\n", *(s + 2));
}
```

- `*s` goes to the address stored in `s`, and `*(s + 1)` goes to the location in memory with the next character, an address that is one byte higher.

    - `s[1]` is **syntactic sugar**, like an abstraction for `*(s + 1)`, equivalent in function but more human-friendly to read and write.
- **Pointer arithmetic** is the process of applying mathematical operations to pointers, using them just like numbers (which they are).
- We can declare an array of numbers, and access them with pointer arithmetic:

```
#include <stdio.h>

int main(void)
{
    int numbers[] = {4, 6, 8, 2, 7, 5, 0};

    printf("%i\n", *numbers);
    printf("%i\n", *(numbers + 1));
    printf("%i\n", *(numbers + 2));
    printf("%i\n", *(numbers + 3));
    printf("%i\n", *(numbers + 4));
    printf("%i\n", *(numbers + 5));
    printf("%i\n", *(numbers + 6));
}
```

```
$ make address
$ ./address
4
6
8
2
7
5
0
```

    - It turns out that we only need to add `1` to the address of `numbers`, instead of `4` (even though `int`s are 4 bytes in size), since the compiler already knows that the type of each value in `numbers` is 4 bytes. With `+ 1`, we're telling the compiler to move the next value in the array, not the next byte.
    - And notice that `numbers` is an array, but we can use it as a pointer with `*numbers`.

Compare and copy

- Let's try to compare two integers from the user:

```
#include <cs50.h>
```

```c
#include <stdio.h>

int main(void)
{
    int i = get_int("i: ");
    int j = get_int("j: ");

    if (i == j)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

```
$ make compare
$ ./compare
i: 50
j: 50
Same
$ ./compare
i: 50
j: 42
Different
```

    - We compile and run our program, and it works as we'd expect, with the same values of the two integers giving us "Same" and different values "Different".

- When we try to compare two strings, we see that the same inputs are causing our program to print "Different":

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    if (s == t)
    {
        printf("Same\n");
    }
    else
```

```
    {
        printf("Different\n");
    }
}
```

```
$ make compare
$ ./compare
s: HI!
t: BYE!
Different
$ ./compare
s: HI!
t: HI!
Different
```

- Even when our inputs are the same, we see “Different” printed.
- Each “string” is a pointer, `char *`, to a different location in memory, where the first character of each string is stored. So even if the characters in the string are the same, this will always print “Different”.

- And `get_string`, this whole time, has been returning just a `char *`, or a pointer to the first character of a string from the user. Since we called `get_string` twice, we got two different pointers back.
- We can fix our program with:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    if (strcmp(s, t) == 0)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

```
$ make compare
```

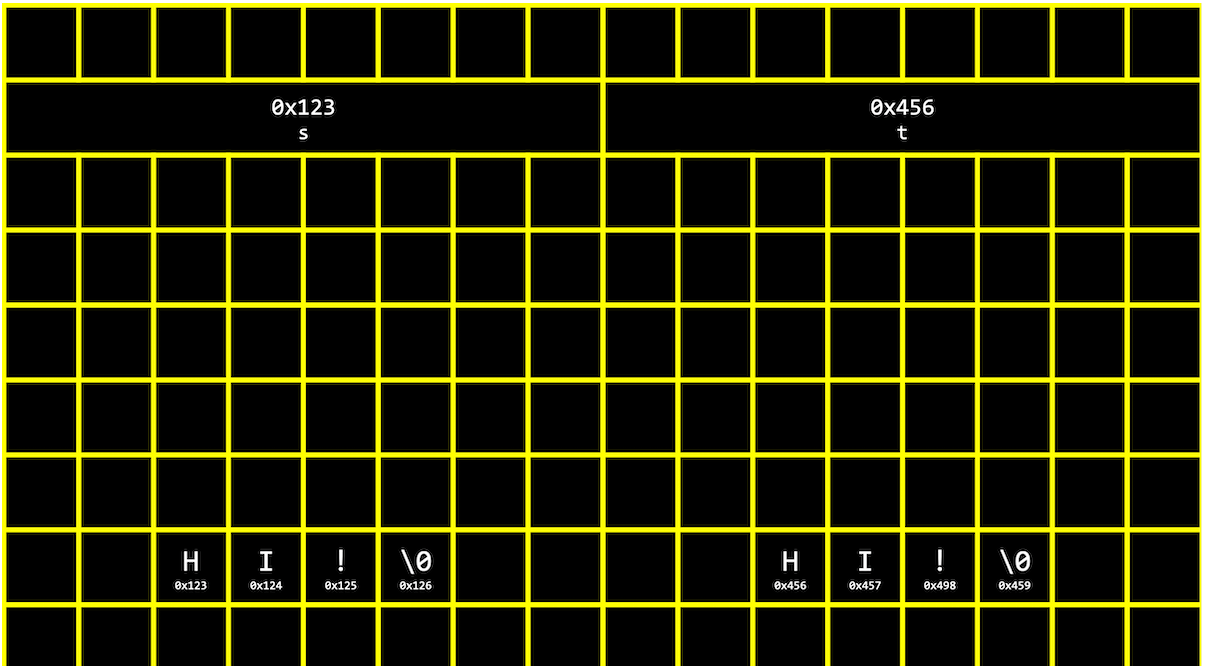
```
$ ./compare
s: HI!
t: HI!
Same
```

- We'll take a look at the values of `s` and `t` as pointers:

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    printf("%p\n", s);
    printf("%p\n", t);
}
```

```
$ make compare
$ ./compare
s: HI!
t: HI!
0x19e06b0
0x19e06f0
```

    - We see that the addresses of our two strings are indeed different.
- Let's visualize how this might look in our computer's memory. Our first string might be at address 0x123, our second might be at 0x456, and `s` will have the value of `0x123`, pointing at that location, and `t` will have the value of `0x456`, pointing at another location:



  - Since our computer puts each string at some location in memory for us, we need `s` and `t` to point to each of them. And now we see why comparing `s` and `t` directly will always print "Different". `strcmp`, in contrast, will go to each string and compare them character by character.
- In C, we can also get the address of `s` or `t`, and store them in a variable of the type `char **`, a pointer to a pointer.
- Let's try to copy a string:

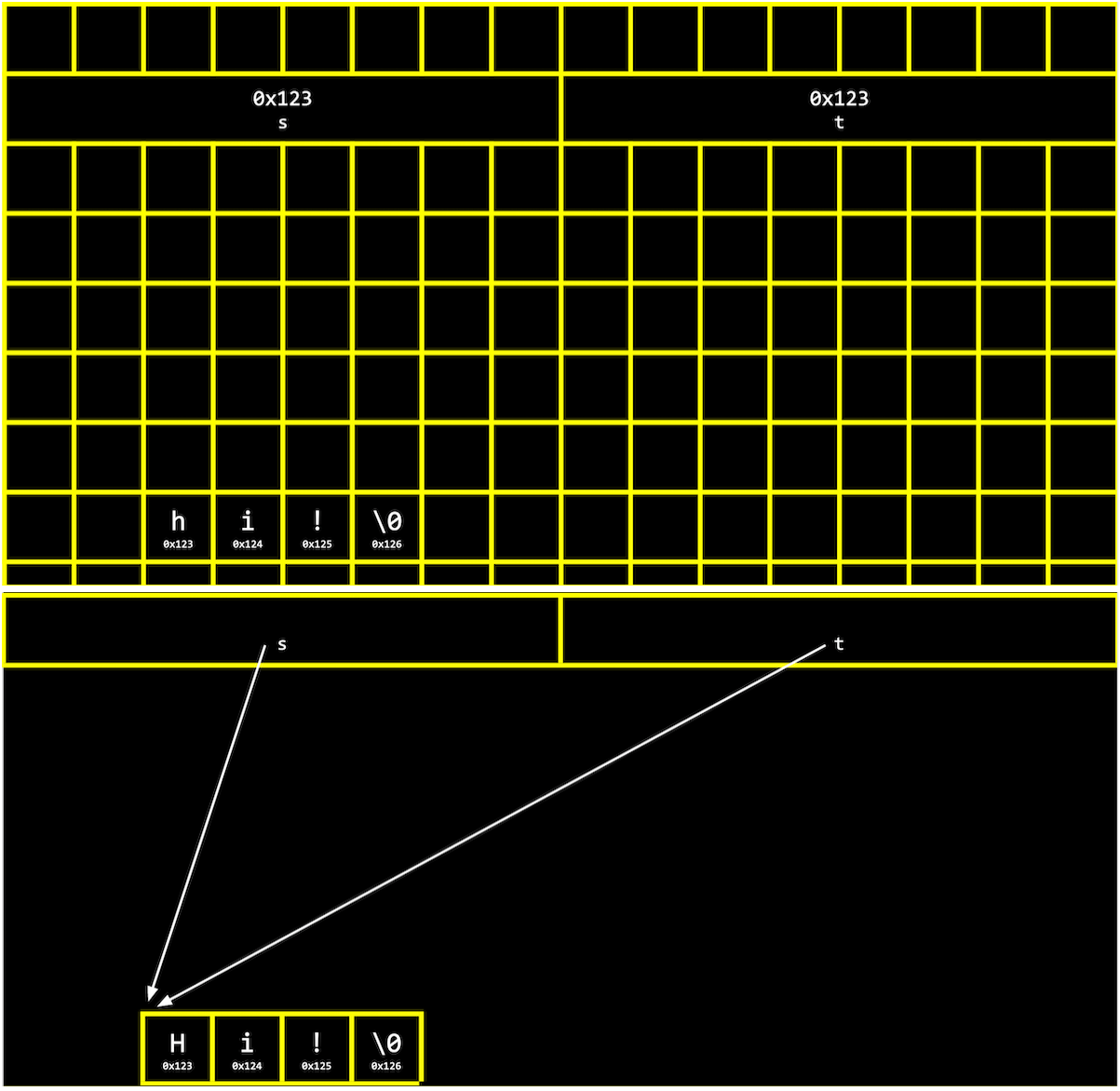
```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("s: ");

    string t = s;

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

```
$ make copy
$ ./copy
s: hi!
```

```
s: Hi!
t: Hi!
```

- We get a string `s`, and copy the value of `s` into `t`. Then, we capitalize the first letter in `t`.
- But when we run our program, we see that both `s` and `t` are now capitalized.

- Since we set `s` and `t` to the same value, or the same address, they're both pointing to the same character, and so we capitalized the same character in memory:



Memory allocation

- To actually make a copy of a string, we have to do a little more work, and copy each character in `s` to somewhere else in memory.
- We'll need to use a new function, **`malloc`**, to *allocate* some number of bytes in memory. And we'll use **`free`** to mark memory as usable when we're done with it, so the operating

system can do something else with it.

    - Our computers might slow down if a program we're running has a bug where it allocates more and more memory but never frees it. The operating system will take longer and longer to find enough available memory for our program.

- Let's copy a string now:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

    char *t = malloc(strlen(s) + 1);

    for (int i = 0, n = strlen(s) + 1; i < n; i++)
    {
        t[i] = s[i];
    }

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

    - We create a new variable to point to a new string with `char *t`. The argument to `malloc` is the number of bytes we'd like to use. We already know the length of `s`, but we need to add 1 for the terminating null character.
    - Then, we copy each character, one at a time, with a `for` loop. We use `strlen(s) + 1` since we want to copy the null character too to end the string. In the loop, we set `t[i] = s[i]`, copying the characters.

- We could also use a library function, `strcpy`:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
int main(void)
{
    char *s = get_string("s: ");

    char *t = malloc(strlen(s) + 1);

    strcpy(t, s);

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);

    free(t);
}
```

```
$ make copy
$ ./copy
s: hi!
s: hi!
t: Hi!
```

  - Now, we can capitalize the first letter of `t`.
  - We'll remember to call `free` on `t`, since we allocated it ourselves.

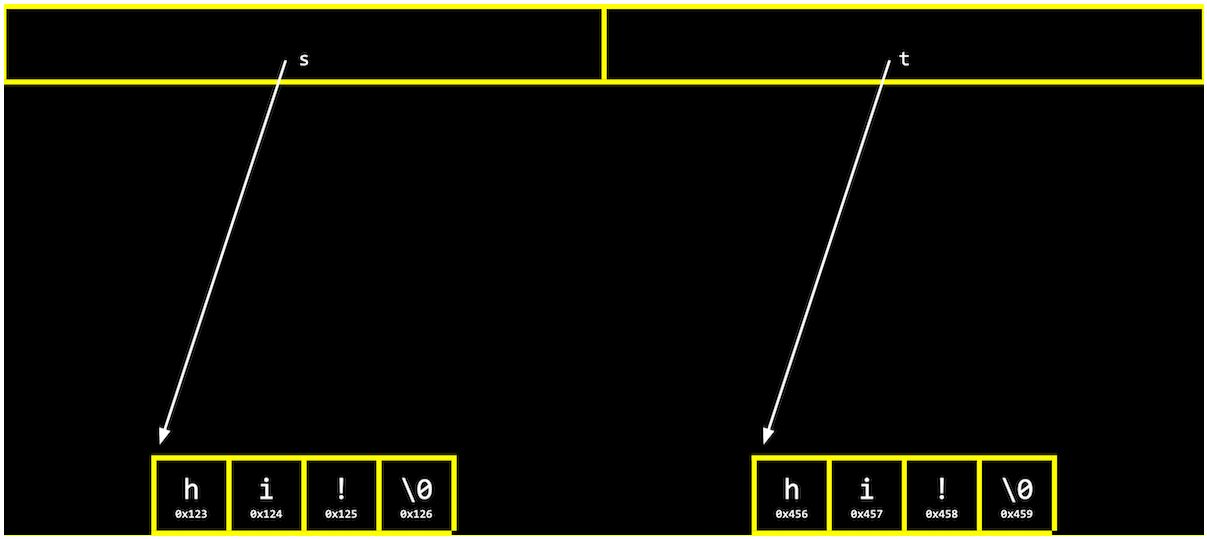
- We can add some error-checking to our program:

```c
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

    char *t = malloc(strlen(s) + 1);
    if (t == NULL)
    {
        return 1;
    }

    strcpy(t, s);

    if (strlen(t) > 0)
```

```
    {
        t[0] = toupper(t[0]);
    }

    printf("s: %s\n", s);
    printf("t: %s\n", t);

    free(t);
}
```

- If our computer is out of memory, `malloc` will return `NULL`, the null pointer, or a special value of all `0` bits that indicates there isn't an address to point to. So we should check for that case, and exit if `t` is `NULL`.
- We should also check that `t` has a length, before trying to capitalize the first character.

- We can visualize how this looks in our computer's memory:



- We've allocated memory at `0x456` and set `t` to point to it. Then, we used `strcpy` to copy each character's value, starting from the address `s` is pointing to, to the address `t` is pointing to.

valgrind

- Let's allocate memory for some integers:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
```

```
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
}
```

```
$ make memory
$ ./memory
$
```

- We'll use `malloc` to get enough memory for 3 times the size of an `int`, which we can find out with `sizeof`.
- We've deliberately made a mistake where we forgot arrays are 0-indexed, and started at `x[1]` instead. Then, with `x[3]`, we're trying to access memory beyond the bounds of what we have access to.
- We also don't free the memory we've allocated.
- When we compile and run our program, though, nothing seems to happen. It turns out that our mistake wasn't bad enough to cause a segmentation fault this time, though it might next time.

- **`valgrind`** is a command-line tool that we can use to run our program and see if it has any memory-related issues.
- We'll run `valgrind ./memory` after compiling, and we'll see a lot of output:

```
$ valgrind ./memory
==5902== Memcheck, a memory error detector
==5902== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5902== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright i
==5902== Command: ./memory
==5902==
==5902== Invalid write of size 4
==5902==    at 0x401162: main (memory.c:9)
==5902==  Address 0x4bd604c is 0 bytes after a block of size 12 alloc'd
==5902==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
==5902==    by 0x401141: main (memory.c:6)
==5902==
==5902==
==5902== HEAP SUMMARY:
==5902==     in use at exit: 12 bytes in 1 blocks
==5902==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==5902==
==5902== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5902==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
==5902==    by 0x401141: main (memory.c:6)
==5902==
```

```
==5902== LEAK SUMMARY:
==5902==    definitely lost: 12 bytes in 1 blocks
==5902==    indirectly lost: 0 bytes in 0 blocks
==5902==      possibly lost: 0 bytes in 0 blocks
==5902==    still reachable: 0 bytes in 0 blocks
==5902==         suppressed: 0 bytes in 0 blocks
==5902==
==5902== For lists of detected and suppressed errors, rerun with: -s
==5902== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

- We'll see some snippets like `Invalid write of size 4 at ... memory.c:9`, which gives us a hint to look at line 9, where we're using `x[3]`.

- We'll fix that mistake:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[0] = 72;
    x[1] = 73;
    x[2] = 33;
}
```

```
$ make memory
$ ./memory
$ valgrind ./memory
==6435== Memcheck, a memory error detector
==6435== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6435== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright i
==6435== Command: ./memory
==6435==
==6435==
==6435== HEAP SUMMARY:
==6435==     in use at exit: 12 bytes in 1 blocks
==6435==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==6435==
==6435== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6435==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
==6435==    by 0x401141: main (memory.c:6)
==6435==
==6435== LEAK SUMMARY:
==6435==    definitely lost: 12 bytes in 1 blocks
==6435==    indirectly lost: 0 bytes in 0 blocks
```

```
==6435==      possibly lost: 0 bytes in 0 blocks
==6435==    still reachable: 0 bytes in 0 blocks
==6435==         suppressed: 0 bytes in 0 blocks
==6435==
==6435== For lists of detected and suppressed errors, rerun with: -s
==6435== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

  - Now, we see less output, with only one error telling us `12 bytes ... are definitely lost`, in that we've allocated them, but not freed them.

- Once we free our memory, we'll see no errors:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[0] = 72;
    x[1] = 73;
    x[2] = 33;
    free(x);
}
```

```
$ make memory
$ ./memory
$ valgrind ./memory
==6812== Memcheck, a memory error detector
==6812== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6812== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright i
==6812== Command: ./memory
==6812==
==6812==
==6812== HEAP SUMMARY:
==6812==     in use at exit: 0 bytes in 0 blocks
==6812==   total heap usage: 1 allocs, 1 frees, 12 bytes allocated
==6812==
==6812== All heap blocks were freed -- no leaks are possible
==6812==
==6812== For lists of detected and suppressed errors, rerun with: -s
==6812== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Garbage values

- Let's take a look at this program:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", scores[i]);
    }
}
```

```
$ make garbage
$ ./garbage
68476128
32765
0
```

  - We declare an array, `scores`, but we didn't initialize it with any values.
  - The values in the array are **garbage values**, or whatever unknown values that were in memory, from whatever program was running in our computer before.
- If we aren't careful with how our programs access memory, users might end up seeing data from previous programs, like passwords. And if we try to go to an address that's a garbage value, our program is likely to crash from a segmentation fault.
- We watch Pointer Fun with Binky (https://www.youtube.com/watch?v=3uLKjb973HU), an animated video demonstrating pointers, `malloc`, and dereferencing. The code from the video might look like this in one program:

```c
int main(void)
{
    int *x;
    int *y;

    x = malloc(sizeof(int));

    *x = 42;
    *y = 13;

    y = x;

    *y = 13;
```

```
}
```

- In the first two lines, we declare two pointers. Then, we allocate memory for `x`, but not `y`, so we can assign a value to the memory `x` is pointing to with `*x = 42;`. But `*y = 13;` is problematic, since we haven't allocated any memory for `y`, and the garbage value there points to some area in memory we likely don't have access to.
- We can write `y = x;` so that `y` points to the same allocated memory as `x`, and use `*y = 13;` to set the value there.

Swap

- We'll have a volunteer on stage try to swap two liquids, one orange liquid in one glass, and one purple liquid in another. But we need a third glass to temporarily pour one liquid into, such as the orange liquid in the first glass. Then, we can pour the purple liquid into the first glass, and finally the orange liquid from the third glass into the second.
- Let's try to swap the values of two integers:

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
$ make swap
$ ./swap
x is 1, y is 2
```

```
x is 1, y is 2
```

  - In our `swap` function, we have a third variable to use as temporary storage space as well. We put `a` into `tmp`, and then set `a` to the value of `b`, and finally `b` can be changed to the original value of `a`, now in `tmp`.
  - But, if we tried to use that function in a program, we don't see any changes.

- It turns out that the `swap` function gets passed in copies of variables, `a` and `b`, which are **local variables** that only the surrounding function can access. Changing those values won't change `x` and `y` in the `main` function:

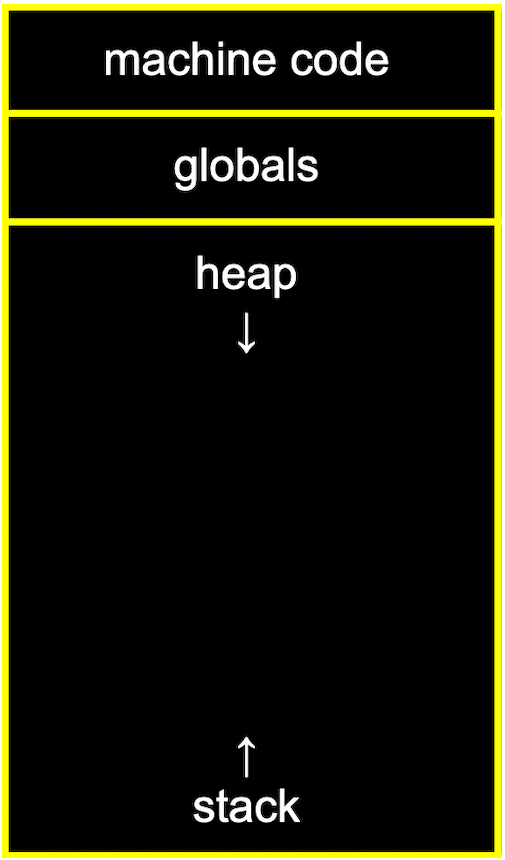
```c
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int a, int b)
{
    printf("a is %i, b is %i\n", a, b);
    int tmp = a;
    a = b;
    b = tmp;
    printf("a is %i, b is %i\n", a, b);
}
```

```
$ make swap
$ ./swap
x is 1, y is 2
a is 1, b is 2
a is 2, b is 1
x is 1, y is 2
```

  - Our `swap` function works while we're inside it.

Memory layout

- Within our computer's memory, different types of data that need to be stored for our program are organized into different sections:



  - The **machine code** section is our compiled program's binary code. When we run our program, that code is loaded into memory.
  - Just below, or in the next part of memory, are **global variables** we declared in our program.
  - The **heap** section is an empty area from where `malloc` can get free memory for our program to use. As we call `malloc`, we start allocating memory from the top down.
  - The **stack** section is used by functions and local variables in our program as they are called, and grows upwards.
- If we call `malloc` for too much memory, we will have a **heap overflow**, since we end up going past our heap. Or, if we call too many functions without returning from them, we will have a **stack overflow**, where our stack has too much memory allocated as well.
- Our program for swapping integers might have a stack that looks like this:

```
 ------------------------
 |  |  |  |  |  |  |  |  |
 ------------------------
 |  |  |  |  |  |  |  |  |
 ------------------------
 |  |  |  |  |  |  |  |  |
 ------------------------
```

```
     |  |  |  |  |  |  |  |  |
     ------------------------
     | tmp       |  |  |  |  |
swap ------------------------
     | a    1    | b    2    |
     ------------------------
main | x    1    | y    2    |
     ------------------------
```

- Our `main` function has two local variables, `x` and `y`. Our `swap` function is created on top of `main` when it's called, and has three local variables, `a`, `b`, and `tmp`.
- Once `swap` returns, its memory is freed and its values are now garbage values, and the variables in `main` haven't been changed.

- By passing in the address of `x` and `y`, our `swap` function will be able to change the original values:

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- The addresses of `x` and `y` are passed in from `main` to `swap` with `&x` and `&y`, and we use the `int *a` syntax to declare that our `swap` function takes in pointers.
- We save the first value to `tmp` by following the pointer `a`, and then set the second value to location pointed to by `a` by following the second pointer `b`.
- Finally, we store the value of `tmp` to the location pointed to by `b`.

- Our stack might look like this:

```
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     | tmp       |  |  |  |  |
swap ------------------------
     | a  0x123  | b  0x127  |
     ------------------------
```

```
main | x    1    | y    2    |
     ----------------------
```

- If `x` is at `0x123`, `a` will contain that value. `b` will have the address of `y`, `0x127`.

- Our first step will be putting the value of `x` into `tmp` by following the pointer `a`:

```
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     | tmp  1    |  |  |  |  |
swap ----------------------
     | a  0x123  | b  0x127  |
     ----------------------
main | x    1    | y    2    |
     ----------------------
```

- Then, we'll follow the pointer `b`, and store the value there into the location pointed to by `a`:

```
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
     | tmp  1    |  |  |  |  |
swap ----------------------
     | a  0x123  | b  0x127  |
     ----------------------
main | x    2    | y    2    |
     ----------------------
```

- Finally, we'll go to the location pointed to by `b`, and put the value of `tmp` into it:

```
     ----------------------
     |  |  |  |  |  |  |  |  |
     ----------------------
```

```
     |  |  |  |  |  |  |  |  |
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     |  |  |  |  |  |  |  |  |
     ------------------------
     | tmp  1    |  |  |  |  |
swap ------------------------
     | a  0x123  | b  0x127  |
     ------------------------
main | x    2    | y    1    |
     ------------------------
```

- Now, `swap` can return, and the variables in `main` will still be changed.
- In our program, we'll need to pass in the addresses of `x` and `y` to our `swap` function:

```c
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(&x, &y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
$ make swap
$ ./swap
x is 1, y is 2
x is 2, y is 1
```

  - With `&x`, we can get the address of `x` to pass in.

scanf

- We can get an integer from the user with a C library function, `scanf`:

```
#include <stdio.h>

int main(void)
{
    int x;
    printf("x: ");
    scanf("%i", &x);
    printf("x: %i\n", x);
}
```

```
$ make scanf
$ ./scanf
x: 50
x: 50
```

  - `scanf` takes a format, `%i`, so the input is "scanned" for that format. We also pass in the address in memory where we want that input to go with `&x`.

- We can try to get a string the same way:

```
#include <stdio.h>

int main(void)
{
    char *s;
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

```
$ clang -o scanf scanf.c
$ ./scanf
s: HI!
s: (null)
```

  - `make` prevents us from making this mistake, so we'll use `clang` to demonstrate.
  - We haven't actually allocated any memory for `s`, so `scanf` is writing our string to an unknown address in memory.

- We can call `malloc` to allocate memory:

```c
#include <stdio.h>

int main(void)
{
    char *s = malloc(4);
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

```
$ clang -o scanf scanf.c
$ ./scanf
s: HI!
s: HI!
```

- And we can declare an array of 4 characters:

```c
#include <stdio.h>

int main(void)
{
    char s[4];
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

```
$ clang -o scanf scanf.c
$ ./scanf
s: hellooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
s: hellooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
Segmentation fault (core dumped)
```

  - Now, if the user types in a string of length 3 or less, our program will work safely. But if the user types in a longer string, `scanf` might be trying to write past the end of our array into unknown memory, causing our program to crash.
- `get_string` from the CS50 library continuously allocates more memory as `scanf` reads in more characters, so it doesn't have this issue.

Files

- With the ability to use pointers, we can also open files, like a digital phone book in `phonebook.c` (https://cdn.cs50.net/2021/fall/lectures/4/src4/phonebook.c?highlight):

```c
// Saves names and numbers to a CSV file

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Open CSV file
    FILE *file = fopen("phonebook.csv", "a");
    if (!file)
    {
        return 1;
    }

    // Get name and number
    string name = get_string("Name: ");
    string number = get_string("Number: ");

    // Print to file
    fprintf(file, "%s,%s\n", name, number);

    // Close file
    fclose(file);
}
```

- `fopen` is a new function we can use to open a file with a new type, `FILE`.
- We can use `fprintf` to write to a file.
- We'll see more details about working with files in this week's problem set.

JPEG

- Let's look at a program that opens a file and tells us if it's a JPEG file, a particular format for image files, with `jpeg.c` (https://cdn.cs50.net/2021/fall/lectures/4/src4/jpeg.c?highlight):

```c
// Detects if a file is a JPEG

#include <stdint.h>
#include <stdio.h>

typedef uint8_t BYTE;
```

```c
int main(int argc, char *argv[])
{
    // Check usage
    if (argc != 2)
    {
        return 1;
    }

    // Open file
    FILE *file = fopen(argv[1], "r");
    if (!file)
    {
        return 1;
    }

    // Read first three bytes
    BYTE bytes[3];
    fread(bytes, sizeof(BYTE), 3, file);

    // Check first three bytes
    if (bytes[0] == 0xff && bytes[1] == 0xd8 && bytes[2] == 0xff)
    {
        printf("Yes, possibly\n");
    }
    else
    {
        printf("No\n");
    }

    // Close file
    fclose(file);
}
```

```
$ make jpeg
$ ./jpeg .src4/lecture.jpg
Yes, possibly
```

- First, we define a `BYTE` as 8 bits, so we can refer to a byte as a type more easily in C.
- Then, we'll read from a file with a function called `fread`.
- We can compare the first three bytes (in hexadecimal) to the three bytes required to begin a JPEG file. If they're the same, then our file is likely to be a JPEG file (though, other types of files may still begin with those bytes). But if they're not the same, we know it's definitely not a JPEG file.

- It turns out that BMP files, another format for images, have even more bytes in its header, or beginning of the file.
- We'll learn more about these in this week's problem set as well, and even implement our own version of image filters, like one that only shows the color red:

```
#include "helpers.h"

// Only let red through
void filter(int height, int width, RGBTRIPLE image[height][width])
{
    // Loop over all pixels
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            image[i][j].rgbtBlue = 0x00;
            image[i][j].rgbtGreen = 0x00;
        }
    }
}
```

  - Here, we have a loop that iterates over all the pixels in a two-dimensional array, and sets the blue and green values to 0.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

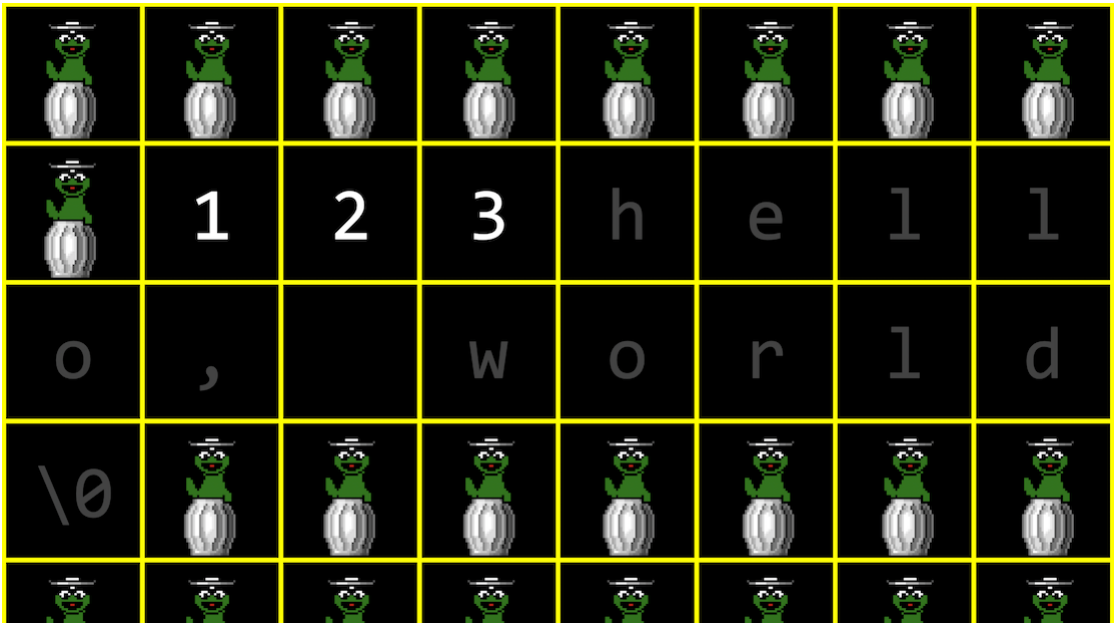
Lecture 5

Recap

- Next week, we'll be introduced to another programming language, Python, where we'll be able to build even more sophisticated programs, with less syntax.
- Last week, we learned about memory. Before that, we learned about arrays, like lists of

values back-to-back in memory.

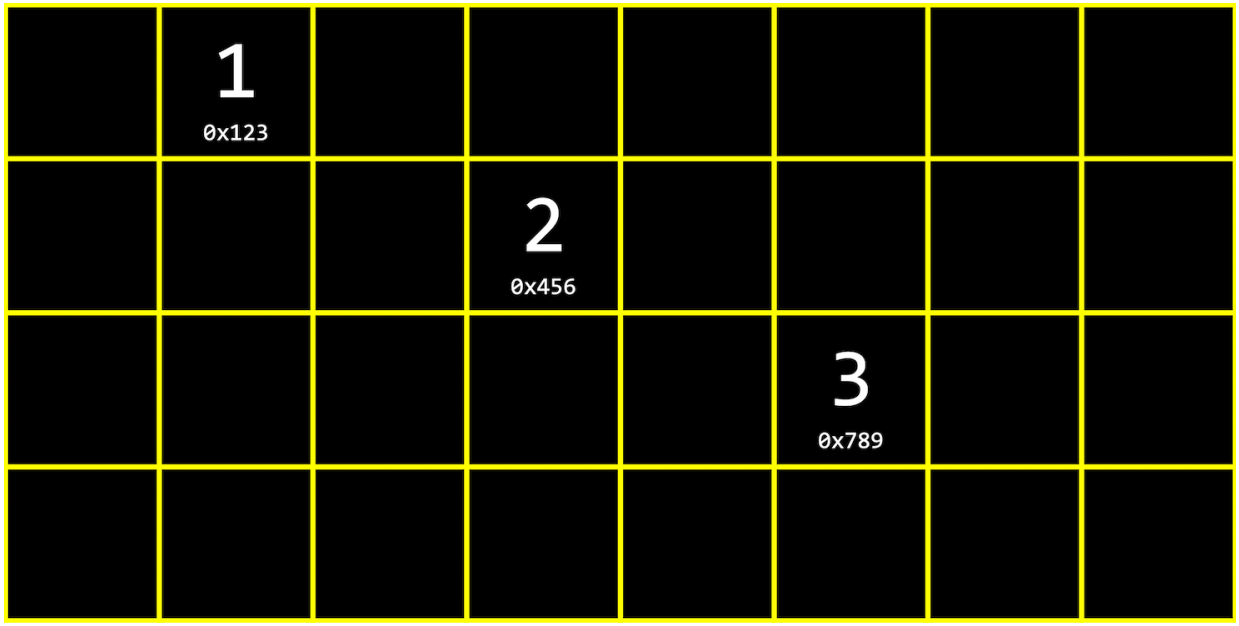
- Let's say we have an array of three numbers, that we want to add another number to. But in our computer's memory, there might already be another value right after, like a string:



  - The free memory, containing garbage values, is represented by a cartoon Oscar (https://en.wikipedia.org/wiki/Oscar_the_Grouch).
- So one solution might be to allocate more memory where there's enough space, and move our array there. But we'll need to copy each of the original numbers first, and then add our new number.
- With a sorted array, we have running time of $O(\log n)$ for search, and $O(n)$ for insert, or adding a new value.
- The best case running times for insert and search both have $\Omega(1)$, since we might get lucky and find our value immediately, or have free memory after our array to add a new value to.
- Recall that we've used these tools before:
  - `struct` to create custom data types
  - `.` to access fields, or values, in a structure
  - `*` to go to an address in memory pointed to by a pointer
  - `->` to access fields in a structure pointed to by a pointer

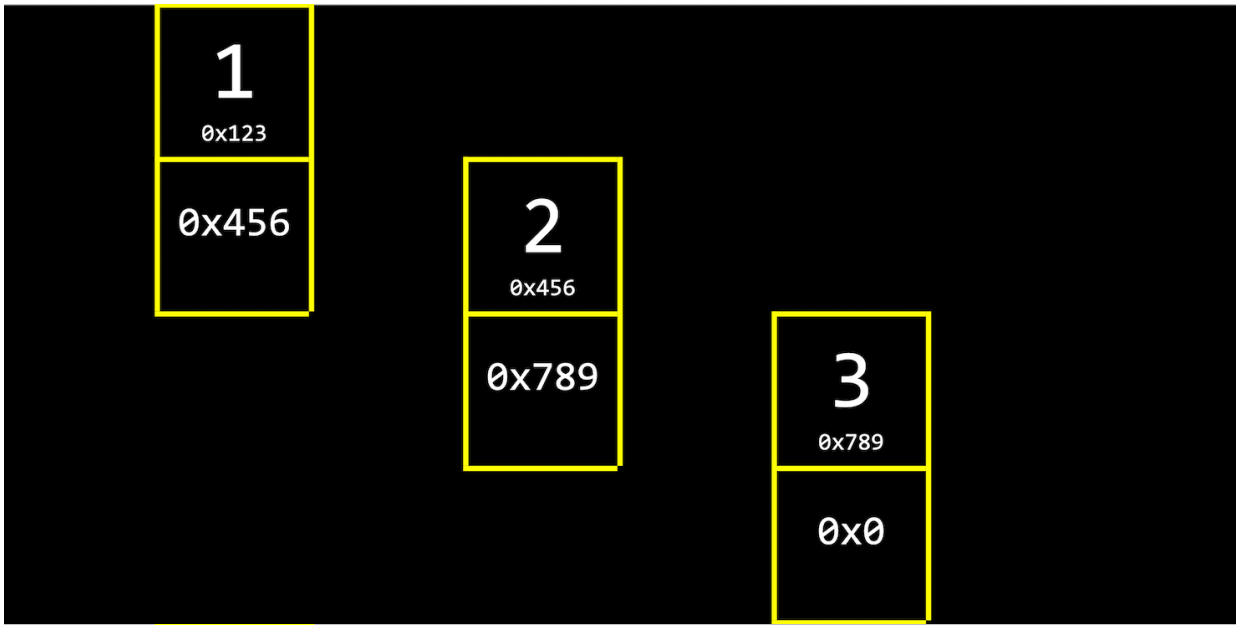
Linked lists

- With a **linked list**, we can store a list of values in different parts of memory:



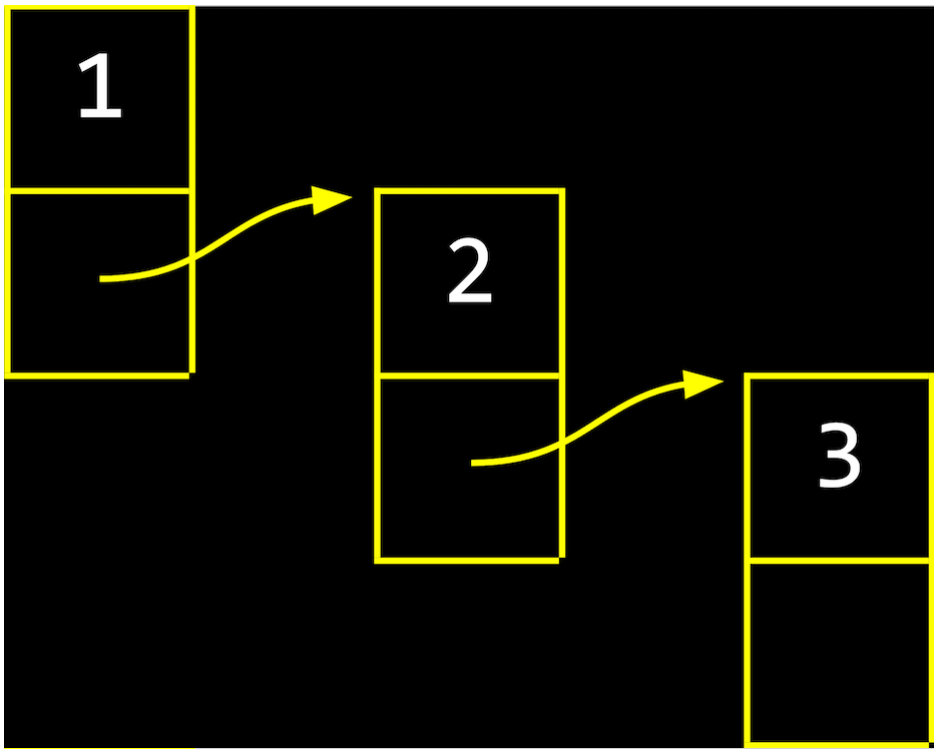
- We have the values `1`, `2`, and `3`, each stored in some address in memory, like `0x123`, `0x456`, and `0x789`.
- This is different than an array since our values are no longer next to one another in memory. We can use whatever locations in memory that are free.

- When we want to insert a new value, we allocate enough memory for both the value we want to store, and the address of the next value:



- Next to our value of `1`, for example, we also store a pointer, `0x456`, to the next value and pointer. (We'll draw them vertically for visualization, but in memory the value and pointer will be adjacent.)
- For our last group of boxes with value `3`, we have the null pointer, `0x0`, since there's no next group.
- We can also visualize these addresses as just pointers, since we don't need to know

what the addresses actually are:



- With a linked list, we have the tradeoff of needing to allocate more memory for each value and pointer, in order to spend less time adding values. (When we copy an array, we do need to allocate more memory, but we free the old array once we finish copying it.)
- We'll call the group of boxes with a value and pointer a **node**, a component of a data structure encapsulates some information. We can implement a node with a struct:

```
typedef struct node
{
    int number;
    struct node *next;
}
node;
```

  - We start this struct with `typedef struct node` so that we can refer to a `struct node` inside our struct.
  - Then, we'll have an `int` called `number`, for the value we want to store, and then a pointer to the next node with `struct node`. (We haven't fully defined `node` yet, so the compiler needs to know it's a custom struct still.)
  - Finally, `node` at the end lets us use just `node` in the rest of our program.
- We can build a linked list in code starting with our struct. First, we'll want to remember an empty list, so we can use the null pointer: `node *list = NULL;`.
- To add a node, we'll first need to allocate some memory:

```
node *n = malloc(sizeof(node));
```

- Recall that we can use `sizeof` to get the size of some data type, including structs. We want to allocate enough memory for both a value and a pointer, and we'll point to that with `n`, a pointer to a `node`.

- If we were able to get memory back from `malloc`, then we'll set the value of `number`:

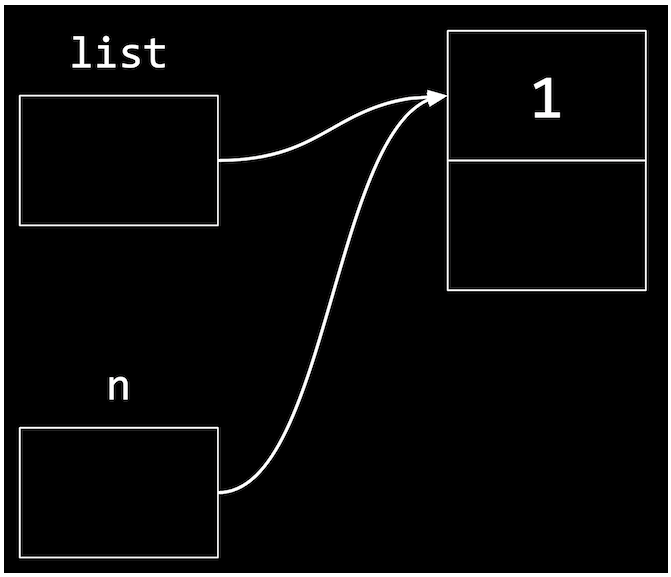
```
if (n != NULL)
{
    (*n).number = 1;
}
```

  - Since `n` is a pointer, we need to go to the `node` there first, and then use the `.` operator to set a value.
  - And instead of `(*n).number`, we can write `n->number`, which has the same effect.

- We'll also want to set the pointer to the `next` node to null:

```
if (n != NULL)
{
    n->number = 1;
    n->next = NULL;
}
```

- Finally, our list needs to point to the node: `list = n;`:



  - We want our `list` pointer to have the same address as `n`, since `n` is a temporary variable and we want our `list` variable to refer to it as the first node in our list.

Growing arrays

- A program that uses an array of three numbers might look like this:

```
#include <stdio.h>

int main(void)
{
    int list[3];

    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", list[i]);
    }
}
```

```
$ make list
$ ./list
1
2
3
```

- If we wanted to have an array with memory from the heap with `malloc`, our program might look like this:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    list[0] = 1;
    list[1] = 2;
    list[2] = 3;
}
```

    - First, we'll allocate enough memory for three integers and point to the first one with `int *list`. (If `malloc` fails, our pointer will be null, and we'll exit our program

with `return 1`.)

- Since arrays in C are equivalent to pointers, we can use the same notation to set the values in our list with `list[0]`, `list[1]`, and `list[2]`. The compiler will perform the correct pointer arithmetic to set values at the right addresses.

- Then, we'll allocate more memory to add another value:

```
// Time passes

int *tmp = malloc(4 * sizeof(int));
if (tmp == NULL)
{
    free(list);
    return 1;
}

for (int i = 0; i < 3; i++)
{
    tmp[i] = list[i];
}
tmp[3] = 4;
```

  - After we allocate enough memory for four integers, we need a temporary pointer, `tmp`, since we need to copy values from our original list into the new chunk of memory. (If `malloc` fails, we'll free the original memory and exit our program with `return 1`.)
  - We'll use a for loop to copy the values from `list`, and set the final value in `tmp`.

- Now, we free our original chunk of memory, and then set `list` to point to the new list:

```
free(list);

list = tmp;

for (int i = 0; i < 4; i++)
{
    printf("%i\n", list[i]);
}

free(list);
return 0;
```

```
$ make list
$ ./list
1
```

```
2
3
4
```

    - We'll print out the list of values to demonstrate, and free it at the end of our program when we're done with it. (Since `list` now points to the same chunk of memory as `tmp`, we can just call `free(list)`.)
- Finally, we can run `valgrind ./list`, and see that there are no memory-related errors:

```
$ valgrind ./list
==9764== Memcheck, a memory error detector
==9764== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9764== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright i
==9764== Command: ./list
==9764==
1
2
3
4
==9764==
==9764== HEAP SUMMARY:
==9764==     in use at exit: 0 bytes in 0 blocks
==9764==   total heap usage: 2 allocs, 2 frees, 28 bytes allocated
==9764==
==9764== All heap blocks were freed -- no leaks are possible
==9764==
==9764== For lists of detected and suppressed errors, rerun with: -s
==9764== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- We can add comments and use another library function, `realloc`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Dynamically allocate an array of size 3
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Assign three numbers to that array
    list[0] = 1;
```

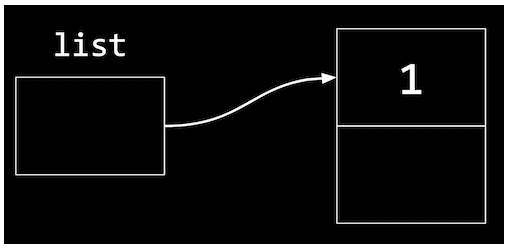
```
    list[1] = 2;
    list[2] = 3;

    // Time passes

    // Resize old array to be of size 4
    int *tmp = realloc(list, 4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
        return 1;
    }

    // Add fourth number to new array
    tmp[3] = 4;

    // Remember new array
    list = tmp;

    // Print new array
    for (int i = 0; i < 4; i++)
    {
        printf("%i\n", list[i]);
    }

    // Free new array
    free(list);
    return 0;
}
```

- Now, instead of allocating new memory and copying the old array to the new array, we can write `int *tmp = realloc(list, 4 * sizeof(int));`. We pass in the pointer to the original chunk of memory, and how much memory we would like to use. `realloc` will grow the original chunk for us if there's enough free memory after it, by allocating it to the same chunk. Otherwise, it will move the chunk of memory for us to a new area, and free the original chunk of memory for us as well.

Growing linked lists

- When we have a large enough array, there might not be enough free memory contiguously, in a row, to store all of our values.
- With a linked list, we can use smaller chunks of free memory for each node, stitching them together with pointers.

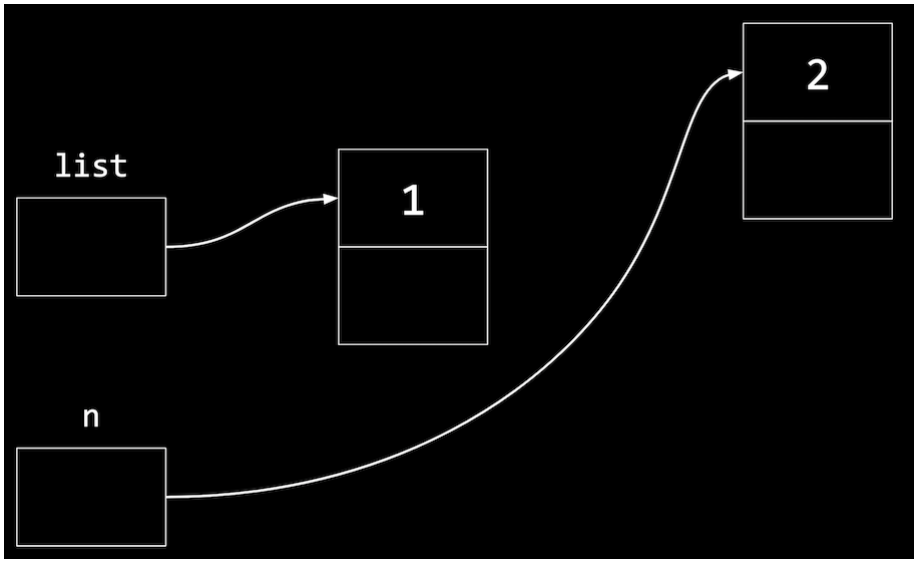
- Let's start with the list we saw earlier, with one node:



- To add to the list, we'll create a new node the same way by allocating more memory:

```
n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 2;
    n->next = NULL;
}
```

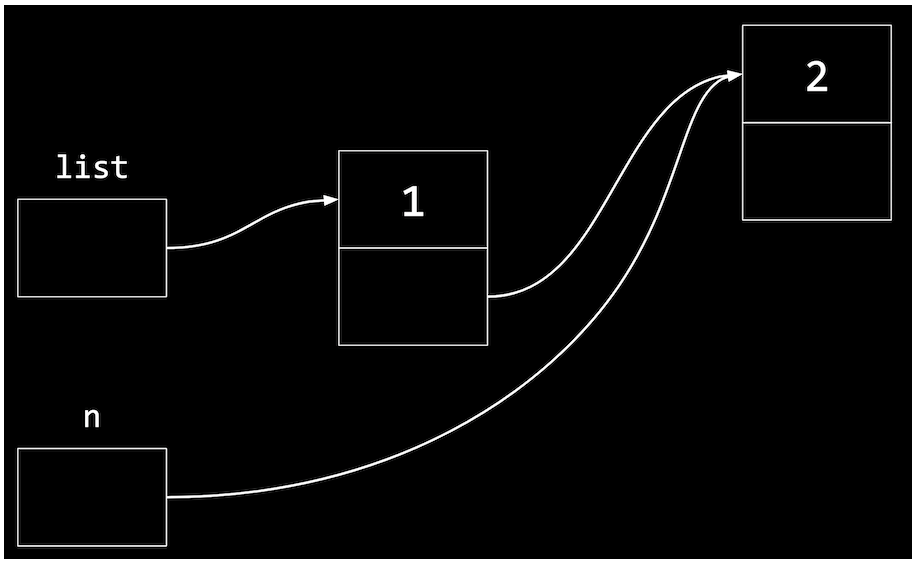
  - `n` is a temporary variable we use to point to this new node:



- And now we need to update the pointer in our first node to point to our new `n`, since we want to maintain a sorted list:

```
list->next = n;
```

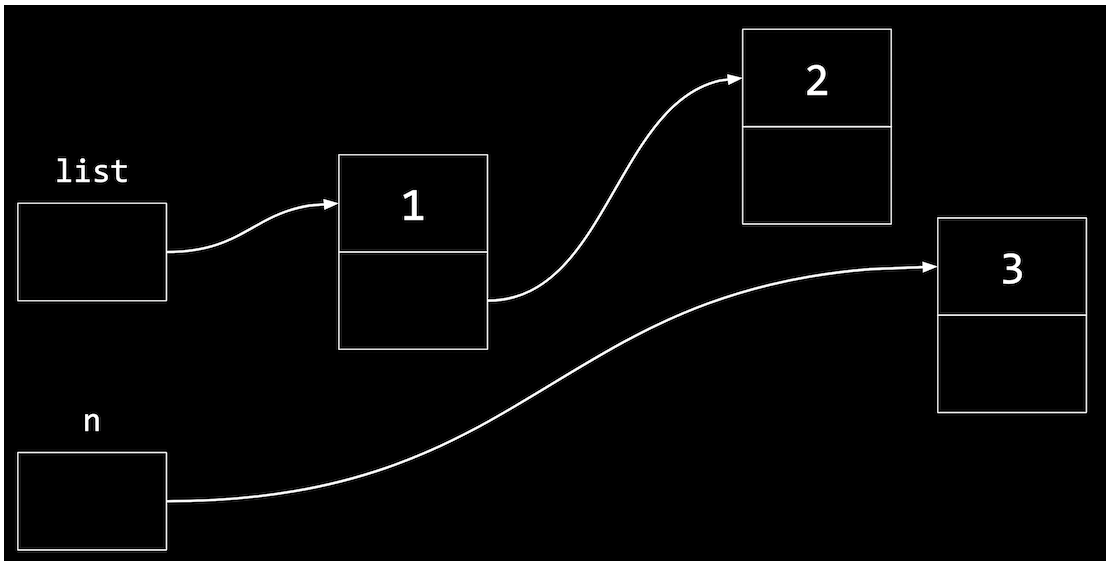
  - This follows the pointer `list`, and sets the `next` field to point to the same node as `n`, since `n` is also a pointer:



- To add a third node, we'll allocate more memory again:

```
node *n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 3;
    n->next = NULL;
}
```

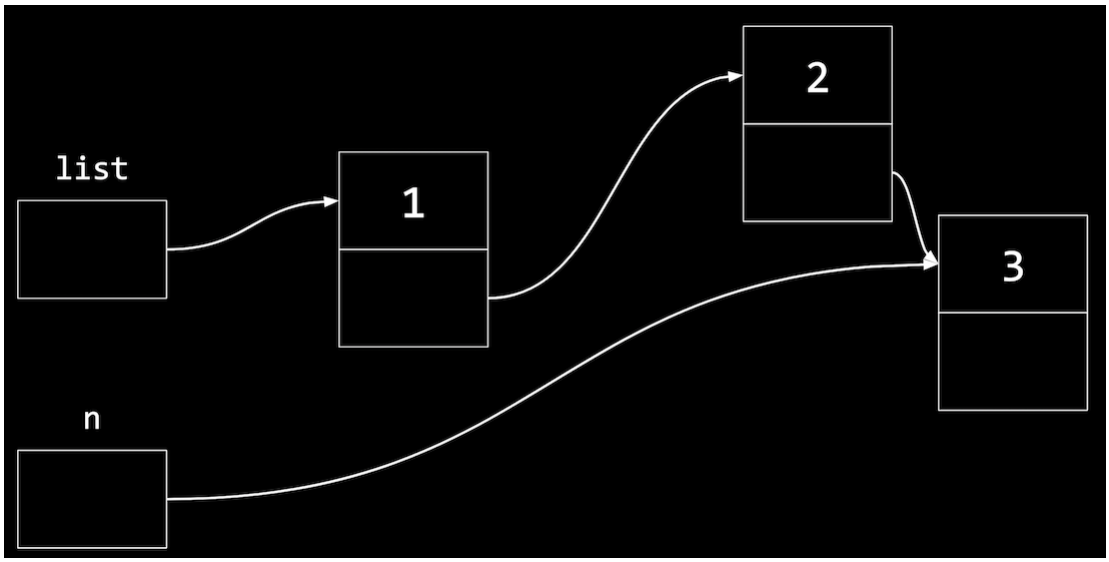
  - Now, `n` points to a new node in memory:



- To insert this node in our list, we'll want to follow the `next` pointer in the first node that `list` points to (the node with value `1`), then setting the `next` pointer in *that node* (with value `2`) to point to the new node:

```
list->next->next = n;
```

  - In general, we'll use a loop to move through our list, but this will manually connect our nodes to look like:



- Even though we're using more memory, and taking multiple steps to insert new nodes in this case (since we're adding to the end of the list), we're able to use small amounts of free space in memory, instead of having to look for a large chunk of contiguous memory.

Implementing linked lists

- Let's combine our snippets of code from earlier into a program that implements a linked list. We'll start by defining a struct called `node`:

```
#include <stdio.h>
#include <stdlib.h>

// Represents a node
typedef struct node
{
    int number;
    struct node *next;
}
node;
```

- Then, we'll allocate memory for the first node, set its values, and point `list` to the new node:

```
int main(void)
{
    // List of size 0
    node *list = NULL;

    // Add number to list
    node *n = malloc(sizeof(node));
```

```
    if (n == NULL)
    {
        return 1;
    }
    n->number = 1;
    n->next = NULL;

    // Update list to point to new node
    list = n;
}
```

- To add a new node, we'll reuse `n` as a pointer, but allocate more memory for the second node:

```
// Add a number to list
n = malloc(sizeof(node));
if (n == NULL)
{
    free(list);
    return 1;
}
n->number = 2;
n->next = NULL;
list->next = n;
```

  - If we somehow couldn't allocate more memory, we'll free the node in our list and exit.
  - Otherwise, we'll set the values for `n`, and set the first node, `list->next`, to point to it.

- Now we can add a third node:

```
// Add a number to list
n = malloc(sizeof(node));
if (n == NULL)
{
    free(list->next);
    free(list);
    return 1;
}
n->number = 3;
n->next = NULL;
list->next->next = n;
```

  - We're starting to see some repetition, and we'll eventually want to use loops, but for now we'll manually demonstrate everything.

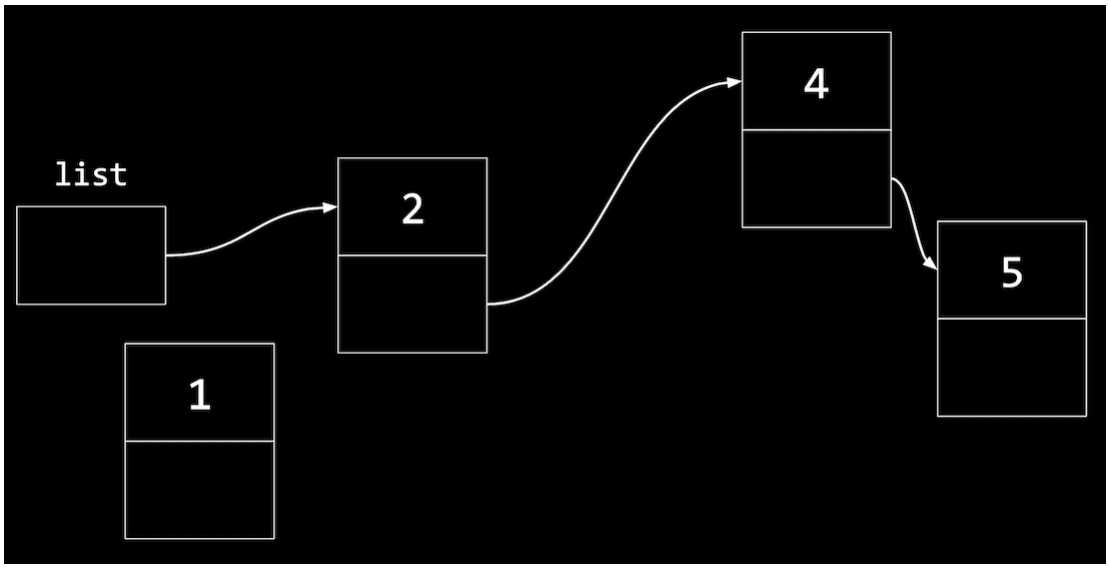
    - Notice that we need to free `list->next`, the second node, and *then* `list`, the first node, since we need to follow it to the second node first.
    - Then, we'll follow the `next` pointer in the first node, and set the `next` pointer in *that node* to point to the new node `n`.
- Finally, we can print our list, and free it with a loop:

```
// Print numbers
for (node *tmp = list; tmp != NULL; tmp = tmp->next)
{
    printf("%i\n", tmp->number);
}

// Free list
while (list != NULL)
{
    node *tmp = list->next;
    free(list);
    list = tmp;
}
return 0;
```

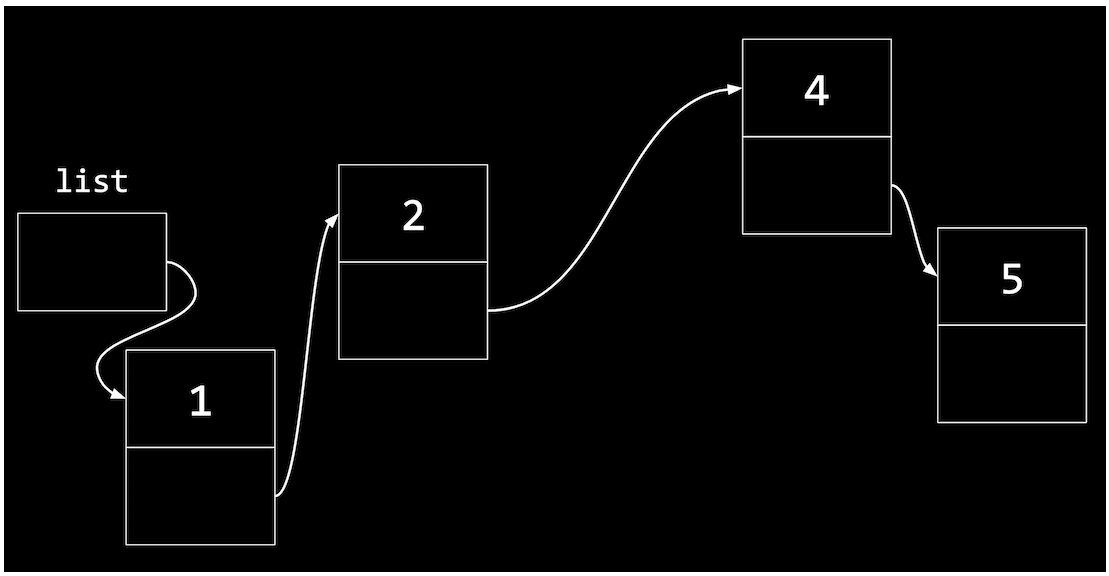
    - We use a temporary pointer, `tmp`, to follow each of our nodes. We'll initialize it to `list` with `node *tmp = list`, which lets us point to the first node in our list.
    - Then, within our loop, we can print `tmp->number`.
    - After each iteration of the loop, we'll update `tmp` to `tmp->next`, which is the pointer to the next node.
    - Finally, the loop will continue while `tmp != NULL`. In other words, our loop will end when `tmp` *is* null, meaning that the current node isn't pointing to another node.
    - Since `tmp` is a pointer that we didn't allocate additional memory for, we don't need to free it.
    - Instead, we'll use a loop to free our list, by using another `tmp` pointer to remember the next node *before* we free the current node. Then, `free(list)` will free the memory for the node that `list` points to. After we do that, we can set `list` to `tmp`, the next node. Our loop will repeat until `list` is null, when no more nodes are left.
    - Recall that we allocated the memory for an array all at once, so we can free it all at once as well. With a linked list, we're responsible for freeing the memory for each node separately, since we allocated it separately as well.
- In Problem Set 5, we'll further explore the implementation of linked lists, and when we

learn about Python, we'll see how the programming language will manage our memory for us.

- With a few volunteers on stage, we demonstrate linked lists. Each volunteer points at another with foam fingers, with some volunteers changing who they point to as new "nodes" as added.
- For example, we'll add a new node, `1`, that needs to be in the middle of our list:



- We'll have to first update the `next` pointer in the node with `1` to point to the next node, *before* updating `list`:



  - Our code to do this might look like:

```
n->next = list;
list = n;
```

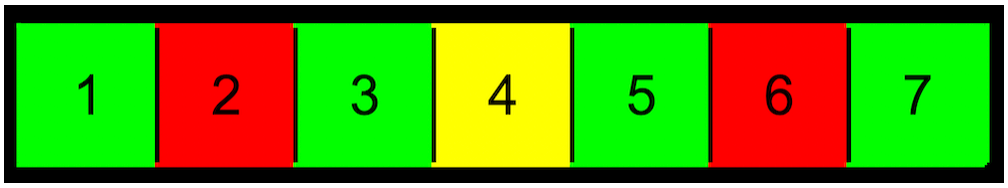
  - If we wrote `list = n` first, we wouldn't know where `n->next` should point to.
- Similarly, to insert a node in the middle of our list, we change the `next` pointer of the new node first to point to the rest of the list, then update the previous node to point to

the new node.

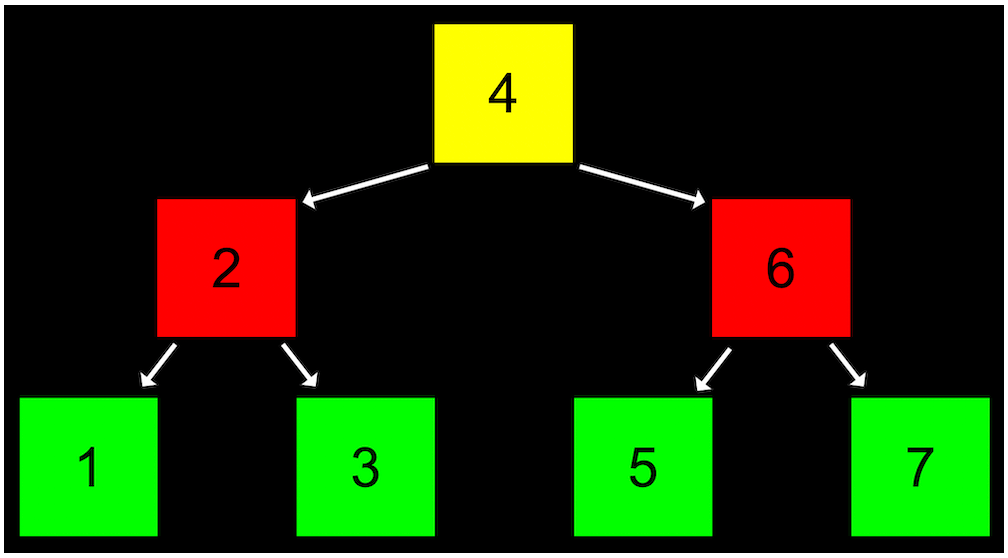
- With a linked list, we have running time of $O(n)$ for search, since we need to follow each node, one at a time. We won't be able to use binary search, since we can't calculate where all of our nodes are. Inserting a node into a sorted list will have running time of $O(n)$ as well, since we might need to insert our node at the end. But if we didn't want to maintain a sorted list, the running time will be $O(1)$, since we can insert at the beginning with just one step.
- The best case running times for insert and search both have $\Omega(1)$, since we might get lucky and find our value immediately, or be able to insert at the beginning of our list for even a sorted list.

Trees

- Recall that with a sorted array, we can use binary search to find an element, starting at the middle (yellow), then the middle of either half (red), and finally left or right (green) as needed:



  - With an array, we can randomly access elements in $O(1)$ time, since we can use arithmetic to go to an element at any index.
- A **tree** is another data structure where each node points to other nodes. We might have a tree where each node points to one to the left (with a smaller value) and one to the right (with a larger value):



  - Notice that we now visualize this data structure in two dimensions (even though the nodes in memory can be at any location).

    - Each node has not one but two pointers to other nodes. All the values to the left of a node are smaller, and all the values of nodes to the right are greater, which allows this to be used as a **binary search tree**.
    - Each node has at most two **children**, or nodes it is pointing to.
    - And like a linked list, we'll want to keep a pointer to just the beginning of the list, but in this case we want to point to the **root**, or topmost node of the tree (the 4).
    - To search for a number, we'll start at the root node, and be able to recursively search the left or right subtree.
    - The height of this tree is 3, or $\log_2 n$, since each **parent** node has up to two children.
- We can define a node with not one but two pointers:

```
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;
```

- Let's use that definition to write a program that uses a tree:

```
int main(void)
{
    // Tree of size 0
    node *tree = NULL;

    // Add number to list
    node *n = malloc(sizeof(node));
    if (n == NULL)
    {
        return 1;
    }
    n->number = 2;
    n->left = NULL;
    n->right = NULL;
    tree = n;
```

    - First, we have a tree with no nodes, so the root is null. Then we allocate memory for a node `n`, and set its value and pointers to children nodes to null. Then, we can set our `tree` to point to that node.
- To add a node, we allocate more memory for another node, and set `tree->left = n`,

since this node should be the left child of the root node.

```
// Add number to list
n = malloc(sizeof(node));
if (n == NULL)
{
    free_tree(tree);
    return 1;
}
n->number = 1;
n->left = NULL;
n->right = NULL;
tree->left = n;
```

- We have a `free_tree` function, which we'll see later.

- We'll add our third node, which will be the right child:

```
// Add number to list
n = malloc(sizeof(node));
if (n == NULL)
{
    free_tree(tree);
    return 1;
}
n->number = 3;
n->left = NULL;
n->right = NULL;
tree->right = n;

// Print tree
print_tree(tree);

// Free tree
free_tree(tree);
return 0;
```

- The `print_tree` function will start at the root node, and recursively print the tree:

```
void print_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    print_tree(root->left);
```

```
    printf("%i\n", root->number);
    print_tree(root->right);
}
```

- Notice that this function recursively prints the left subtree first, then the root node's value, then the right subtree. Since all the values to the left are lower, and all the values to the right will be higher, the values will be printed in order:

```
$ make tree
$ ./tree
1
2
3
```

- We can even swap `print_tree(root->left);` and `print_tree(root->right);` to print our tree in reverse order:

```
void print_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    print_tree(root->right);
    printf("%i\n", root->number);
    print_tree(root->left);
}
```

```
$ make tree
$ ./tree
3
2
1
```

- To free the memory for each of the nodes in our tree, we'll have to recursively free both children first:

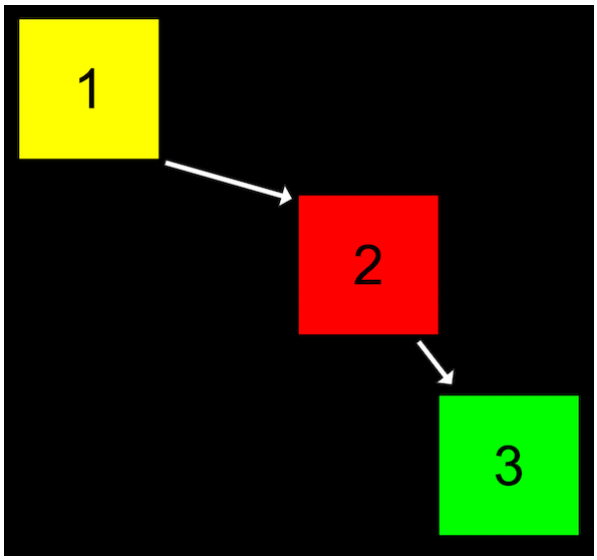
```
void free_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    free_tree(root->left);
    free_tree(root->right);
```

```
    free(root);
}
```

- We can also search our tree with an implementation of binary search:

```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
```

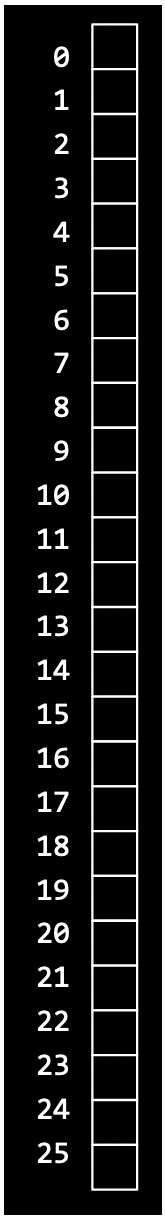
    - If we don't have any more nodes to look at, then we know the number we're looking for isn't in the tree, and we can return `false`.
    - Otherwise, we can search either the left or the right subtrees.
    - And if the number is at the node we're looking at, we can return `true`.
    - The last conditional can be simplified to `else`, since there's no other case possible.
- If we add nodes in inefficient ways, though, our binary search tree might start to look like a linked list:



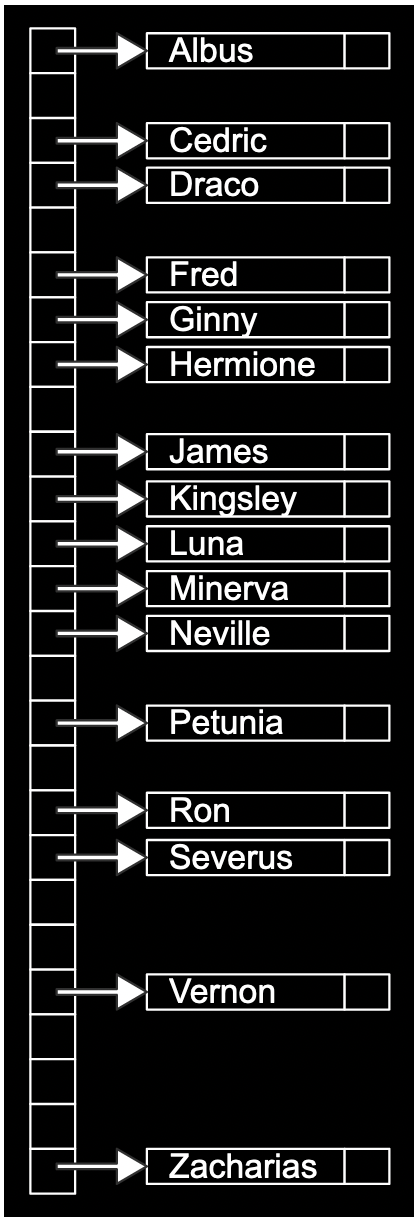
    - We started our tree with a node with value of `1`, then added the node with value `2`, and finally added the node with value `3`. Even though this tree follows the constraints of a binary search tree, it's not as efficient as it could be.
    - We can make the tree balanced, or more optimal, by making the node with value `2` the new root node.
- With a balanced binary search tree, the running time for search and insert will be $O(\log n)$. But if our tree isn't balanced, it can devolve into a linked list, with running time for search and insert of $O(n)$.

More data structures

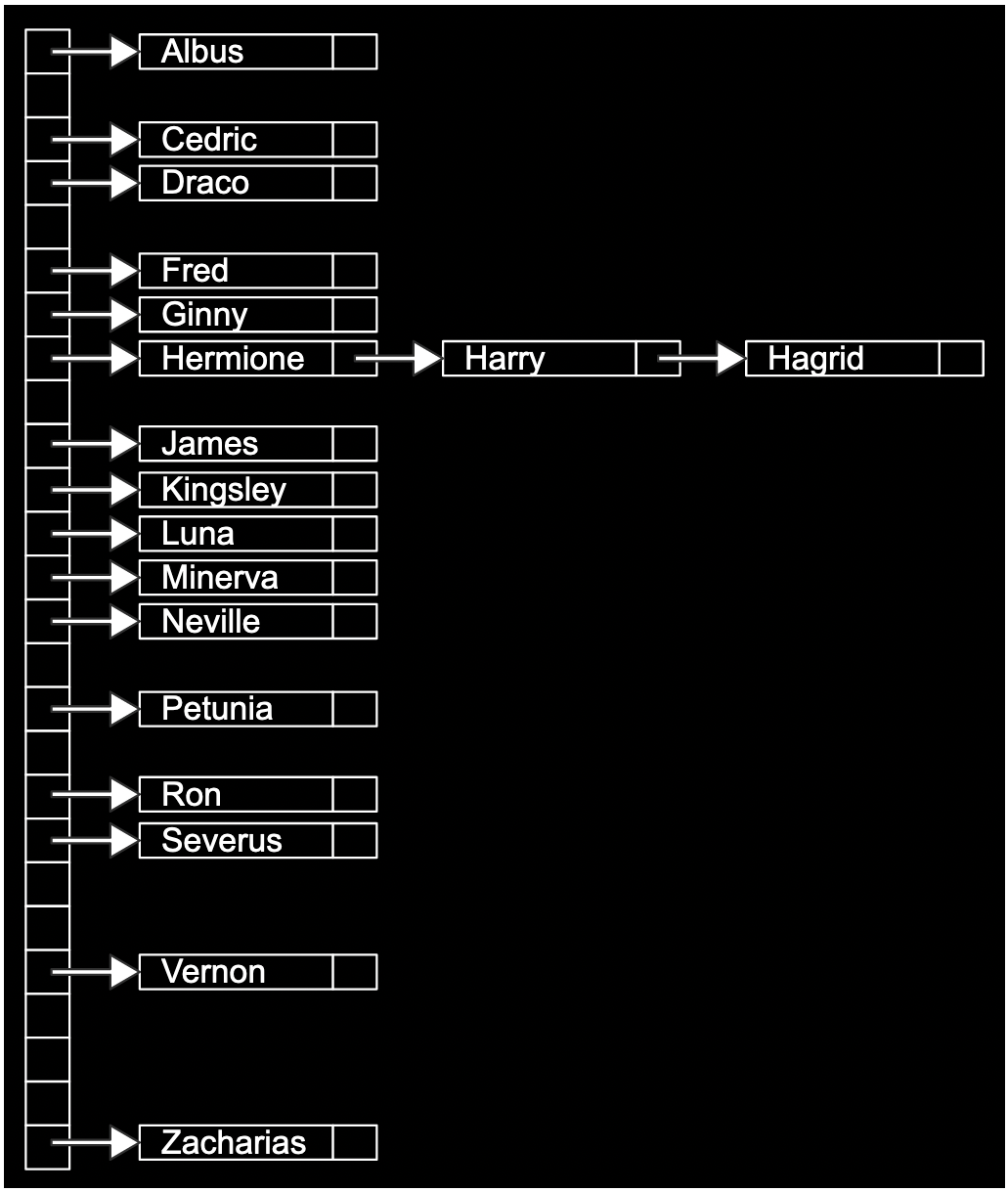
- A **hash table** is a data structure that allows us to associate keys with values. It looks like an array, where we can jump to each location by its index:



- We can think of each location as labeled with a letter from A through Z, and insert names into each location:



- If we have multiple names with the same first letter, we can add them with a linked list:



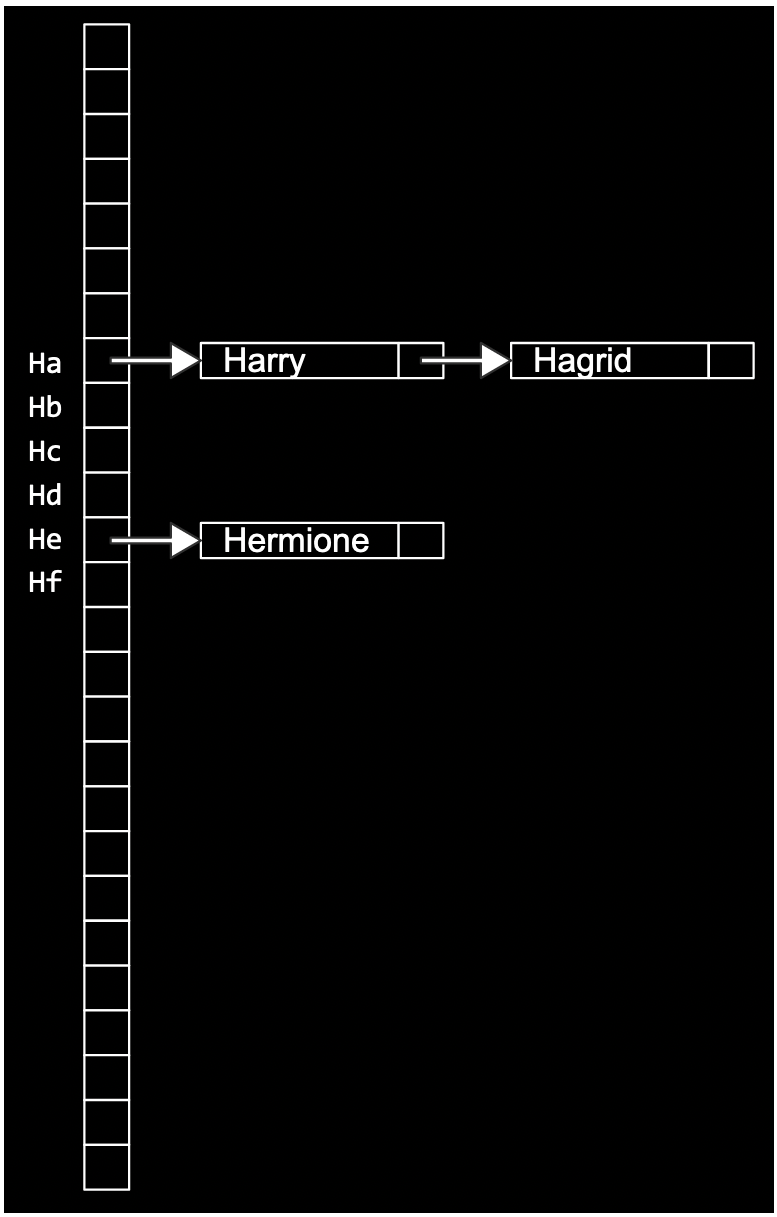
  - The array has 26 pointers, some of which are null, but some pointing to a name in a node, each of which may also point to another name in another node.
- We can describe each node in code with:

```
typedef struct node
{
    char word[LONGEST_WORD + 1];
    struct node *next;
}
node;
```

  - Each node will have an array of characters already allocated, of maximum size `LONGEST_WORD + 1`, called `word`, that it's storing. Then, a `next` pointer will point to another node, if there is one.
- And to create the hash table, we might write:

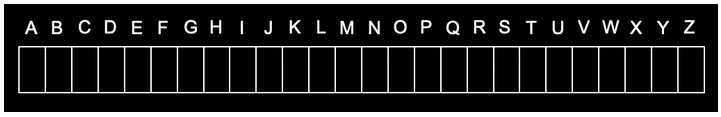
```
node *hash_table[NUMBER_OF_BUCKETS];
```

    - The hash table will be an array of pointers to nodes, with `NUMBER_OF_BUCKETS` as its size.
- To decide which bucket, or location in the array, that a value should be placed in, we use a **hash function**, which takes some input and produces an index, or location. In our example, the hash function just returns an index corresponding to the first letter of the name, such as "0" for "Albus" and "25" for "Zacharias".
    - We might start sorting a shuffled deck of cards by dividing them into four buckets, each labeled by suit, and then sort each of the suits.
- We can try to have smaller chains in our hash table by using two letters, instead of just one:

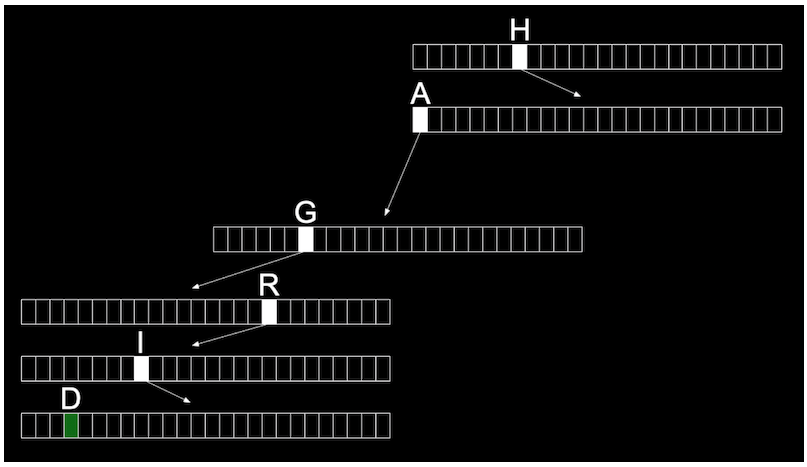


    - Now, we'll have 676 buckets total, for all the combinations of the first two letters.

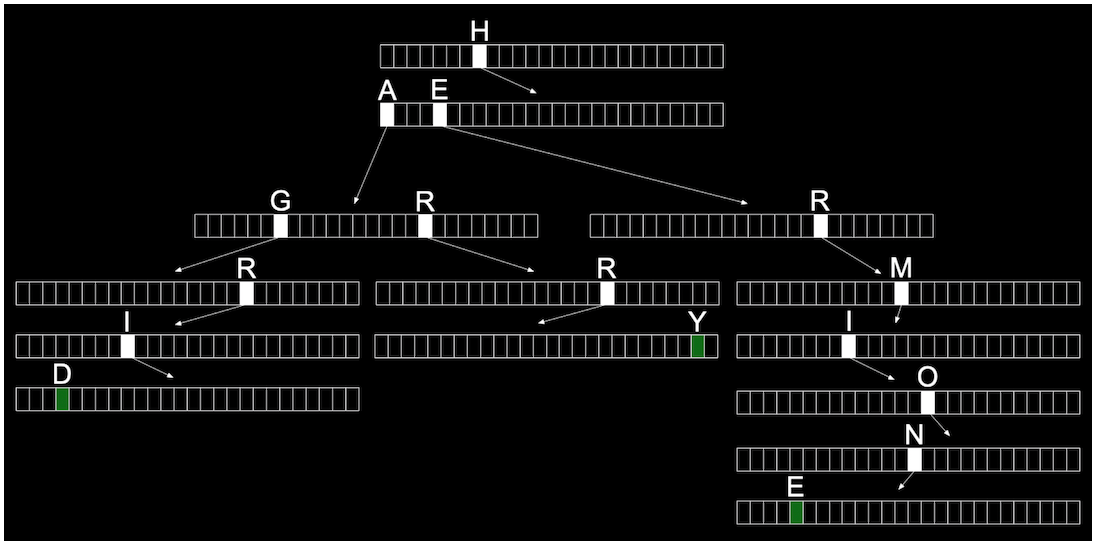
- We can consider the first three letters with even more buckets, but we'll be using more space in memory. Some of those buckets will be empty, but we're more likely to only need one step to look for a value, reducing our running time for searching.
- It turns out that the worst case running time for searching a hash table is $O(n)$, since all of our values might be in the same bucket, devolving into a linked list as well. In practice, though, the running time will likely be much faster.
- We can use another data structure called a **trie** (pronounced like "try", and is short for "retrieval"). A trie is a tree with arrays as nodes:



  - Each array will have locations that represent each letter, A-Z.
- For each word, the first letter will point to an array, where the next valid letter will point to another array, and so on, until we reach a boolean value indicating the end of a valid word, marked in green:



- With multiple names, we start seeing some of the space being reused for the first letters that are shared:



- We might define a trie in code with:

```
typedef struct node
{
    bool is_word;
    struct node *children[SIZE_OF_ALPHABET];
}
node;
```

    - At each node, or array, we'll have a boolean value that indicates if it's a valid word (whether or not it should be green). Then, we'll have an array of `SIZE_OF_ALPHABET` pointers to other nodes, called `children`.
- Now, the height of our tree is the length of the longest word we want to store.
- And even if our data structure has lots of words, the maximum lookup time will be just the length of the word we're looking for. This might be a fixed maximum, so we have a constant time, $O(1)$, for searching and insertion.
- The cost for this, though, is that we need lots of memory to store mostly null pointers.
- There are even higher-level constructs, **abstract data structures**, where we use our building blocks of arrays, linked lists, trees, hash tables, and tries to solve some other problem.
- For example, one abstract data structure is a **queue**, like a line of people waiting, where the first value we put in are the first values that are removed, or first-in-first-out (FIFO). To add a value we **enqueue** it, and to remove a value we **dequeue** it. We could use an array that we have to grow, or we could use a linked list.
- Another abstract data structure is a **stack**, where items most recently added are removed first: last-in-first-out (LIFO). In a dining hall, we might take, or **pop**, the top tray from a stack, and clean trays would be added, or **pushed**, to the top as well.
- We take a look at "Jack Learns the Facts About Queues and Stacks" (https://www.youtube.com/watch?v=ltAG3s6KIEI), an animation about these data structures.
- A restaurant might place food orders in multiple shelves, with areas each labeled by the first letter of the customer's name. This is an example of a **dictionary**, where we can map keys to values.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

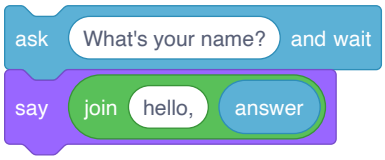
Lecture 6

Python syntax

- Today we'll learn a new programming language called Python, though our main goal is to learn how to teach ourselves new languages.
- Source code in Python looks a lot simpler than C, though it has many of the same ideas. To print "hello, world", all we need to write is:

```
print("hello, world")
```

  - Notice that, unlike in C, we don't need to specify a new line in the `print` function, or use a semicolon to end our line.
- In Scratch and C, we might have had multiple functions:




```
string answer = get_string("What's your name? ");
printf("hello, %s\n", answer);
```

- In Python, the equivalent would look like:

```
answer = get_string("What's your name? ")
print("hello, " + answer)
```

  - We can create a variable called `answer` without specifying the type, and we can join, or concatenate, two strings together with the `+` operator before we pass it into `print`.
  - The `get_string` function also comes from the Python version of the CS50 library.
- We can also write:

```
print(f"hello, {answer}")
```

  - The `f` before the double quotes indicates that this is a format string, which will allow us to use curly braces, `{}`, to include variables that should be substituted, or interpolated.
- We can create variables with just `counter = 0`. To increment a variable, we can write `counter = counter + 1` or `counter += 1`.
- Conditionals look like:

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
```

```
    print("x is equal to y")
```

  - Unlike in C, where curly braces are used to indicate blocks of code, the exact indentation of each line determines the level of nesting in Python. And we don't need parentheses around the Boolean expressions.
  - And instead of `else if`, we just say `elif`.
- We can create a forever loop with a while loop:

```
while True:
    print("meow")
```

  - Both `True` and `False` are capitalized in Python.
- We can write a while loop with a variable:

```
i = 0
while i < 3:
    print("meow")
    i += 1
```

- We can write a `for` loop, where we can do something for each value in a list:

```
for i in [0, 1, 2]:
    print("hello, world")
```

  - Lists in Python, `[0, 1, 2]`, are like arrays or linked lists in C.
  - This `for` loop will set the variable `i` to `0`, run, then to the second value, `1`, run, and so on.
- And we can use a special function, `range`, to get any number of values:

```
for i in range(3):
    print("hello, world")
```

  - `range(3)` will give us a list up to but not including 3, with the values `0`, `1`, and `2`, that we can then use.
  - `range()` takes other arguments as well, so we can have lists that start at different values and have different increments between values.
- In Python, there are built-in data types similar to those in C:
  - `bool`, `True` or `False`
  - `float`, real numbers
  - `int`, integers which can grow as needed
  - `str`, strings
- Other types in Python include:

    - `range`, sequence of numbers
    - `list`, sequence of mutable values, or values we can change
    - `tuple`, sequence of immutable values
    - `dict`, dictionaries, collection of key/value pairs, like a hash table
    - `set`, collection of unique values, or values without duplicates
- The CS50 library for Python includes functions for getting user input as well:
    - `get_float`
    - `get_int`
    - `get_string`
- And we can import an entire library, functions one at a time, or multiple functions:

```
import cs50
```

```
from cs50 import get_float
from cs50 import get_int
from cs50 import get_string
```

```
from cs50 import get_float, get_int, get_string
```

Libraries

- Recall that, in C, we needed to compile a program with `make hello` before we could run it.
- To run a program we wrote in Python, we'll only need to run:

```
python hello.py
```

    - `python` is the name of a program called an **interpreter**, which reads in our source code and translates it to code that our CPU can understand, line by line.
    - Our source code files will also end in `.py`, to indicate that they're written in the Python language.
- For example, if our pseudocode for finding someone in a phone book was in Spanish and we didn't understand Spanish, we would have to slowly translate it, line by line, into English first:

```
1   Recoge guía telefónica
2   Abre a la mitad de guía telefónica
3   Ve la página
4   Si la persona está en la página
```

```
5       Llama a la persona
6   Si no, si la persona está antes de mitad de guía telefónica
7       Abre a la mitad de la mitad izquierda de la guía telefónica
8       Regresa a la línea 3
9   Si no, si la persona está después de mitad de guía telefónica
10      Abre a la mitad de la mitad derecha de la guía telefónica
11      Regresa a la línea 3
12  De lo contrario
13      Abandona
```

- Similarly, programs in Python will take some extra time to be interpreted as they are run.

- We can blur an image with:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.BoxBlur(10))
after.save("out.bmp")
```

  - In Python, we include other libraries with `import`, and here we'll `import` the `Image` and `ImageFilter` names from the `PIL` library.
  - `Image` is an **object**, like a struct in C. Objects in Python can have not just values, but functions that we can access with the `.` syntax, such as with `Image.open`. And `before` is an object with a `filter` function as well, which we can find in the documentation for the library.
  - We can run this with `python blur.py` in the same directory as our `bridge.bmp` file:

```
filter/ $ ls
blur.py  bridge.bmp
filter/ $ python blur.py
filter/ $ ls
blur.py  bridge.bmp  out.bmp
filter/ $
```

- We can also find the edges in the image with:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.FIND_EDGES)
after.save("out.bmp")
```

- We can implement a dictionary with:

```
words = set()

def check(word):
    if word.lower() in words:
        return True
    else:
        return False

def load(dictionary):
  file = open(dictionary, "r")
  for line in file:
      word = line.rstrip()
      words.add(word)
  file.close()
  return True

def size():
    return len(words)

def unload():
    return True
```

  - First, we create a new set called `words`, which we can add values to, and have the language check for duplicates for us.
  - We'll define a function with `def`, and check if `word`, the argument, is in our hash table. (We'll also call a function, `.lower()`, to get the lowercase version of the word.)
  - Our `load` function will take a file name, `dictionary`, and open it for reading. We'll iterate over the lines in the file with `for line in file:`, and add each word after removing each line's newline with `rstrip`.
  - For `size`, we can use `len` to count the number of items in our dictionary, and finally, for `unload`, we don't have to do anything, since Python manages memory for us.
- We're able to run our program with `python speller.py texts/holmes.txt`, but we'll notice that it takes a few seconds longer to run than the C version. Even though it was much faster for us to write, we aren't able to fully optimize our code by way of managing memory and implementing all of the details ourselves.
- It turns out, we can cache, or save, the interpreted version of our Python program, so it runs faster after the first time. And Python is actually partially compiled too, into an intermediate step called bytecode, which is then run by the interpreter.

Input, conditions

- We can practice getting input from the user:

```
from cs50 import get_string

answer = get_string("What's your name? ")
print("hello, " + answer)
```

```
$ python hello.py
What's your name? David
hello, David
```

  - Notice that our program doesn't need a `main` function. Instead, our code will automatically run line by line.
  - We can also use a format string: `print(f"hello, {answer}")`.
- We can also use a function that comes with Python, `input`:

```
answer = input("What's your name? ")
print("hello, " + answer)
```

  - Since we're just getting a string from the user, `input` is the same as `get_string`.
- `get_int` and `get_float` will have error-checking for us, so we can get numeric values more easily:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")
print(x + y)
```

  - Since we're printing just one value, we can pass it to `print` directly.
- If we import the entire library, we see an error with a stack trace, or traceback:

```
import cs50

x = get_int("x: ")
y = get_int("y: ")
print(x + y)
```

```
$ python calculator.py
Traceback (most recent call last):
  File "/workspaces/20377622/calculator.py", line 3, in <module>
```

```
    x = get_int("x: ")
NameError: name 'get_int' is not defined
```

  - It turns out that we need to write `cs50.get_int(...)` when we import the entire library. This allows us to **namespace** functions, or keep their names in different spaces, with different prefixes. Then, multiple libraries with a `get_int` function won't collide.

- If we call input ourselves, we get back strings for our values:

```
x = input("x: ")
y = input("y: ")
print(x + y)
```

```
$ python calculator.py
x: 1
y: 2
12
```

  - And we print the two strings, joined together as another string.

- So we need to **cast**, or convert, each value from `input` into an `int` before we store it:

```
x = int(input("x: "))
y = int(input("y: "))
print(x + y)
```

  - Notice that `int` in Python is a function that we can pass a value into.
  - But if the user didn't type in a number, we'll need to do error-checking or our program will crash:

```
$ python calculator.py
x: cat
Traceback (most recent call last):
  File "/workspaces/20377622/calculator.py", line 1, in <module>
    x = int(input("x: "))
ValueError: invalid literal for int() with base 10: 'cat'
```

- `ValueError` is a type of **exception**, or something that goes wrong when our code is running. In Python, we can try to do something, and detect if there is an exception:

```
try:
    x = int(input("x: "))
except ValueError:
    print("That is not an int!")
    exit()
```

```
try:
    y = int(input("y: "))
except ValueError:
    print("That is not an int!")
    exit()
print(x + y)
```

- Now, if there's an exception with converting the input into an integer, we can print a message and exit without crashing.

- We can divide values:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")

z = x / y
print(z)
```

```
$ python calculator.py
x: 1
y: 10
0.1
```

- Notice that we get floating-point, decimal values back, even if we divided two integers. The division operator in Python doesn't truncate those values by default. (We can get the same behavior as in C, truncation, with the `//` operator, like `z = x // y`.)

- We can use a format string to print out more digits after the decimal point:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")

z = x / y
print(f"{z:.50f}")
```

```
$ python calculator.py
x: 1
y: 10
0.10000000000000000555111512312578270211815834045410
```

- Unfortunately, we still have floating-point imprecision.

- Comments in Python start with a `#`:

```
from cs50 import get_int

# Prompt user for points
points = get_int("How many points did you lose? ")

# Compare points against mine
if points < 2:
    print("You lost fewer points than me.")
elif points > 2:
    print("You lost more points than me.")
else:
    print("You lost the same number of points as me.")
```

  - Our code is also much shorter than the same program in C.
- We can check the parity of a number with the remainder operator, `%`:

```
from cs50 import get_int

n = get_int("n: ")

if n % 2 == 0:
    print("even")
else:
    print("odd")
```

```
$ python parity.py
n: 50
even
```

- To compare strings, we can say:

```
from cs50 import get_string

s = get_string("Do you agree? ")

if s == "Y" or s == "y":
    print("Agreed.")
elif s == "N" or s == "n":
    print("Not agreed.")
```

```
$ python agree.py
Do you agree? y
Agreed.
```

    - Python doesn't have a data type for single characters, so we check `Y` and other letters as strings. (We can use either single or double quotes for strings, too, as long as we're consistent.)
    - We can compare strings directly with `==`, and we can use `or` and `and` in our Boolean expressions.
- We can also check if our string is in a list, after converting it to lowercase first:

```
from cs50 import get_string

s = get_string("Do you agree? ")

s = s.lower()

if s in ["y", "yes"]:
    print("Agreed.")
elif s in ["n", "no"]:
    print("Not agreed.")
```

    - We call `s.lower()` to get the lowercase version of the string, and then store it back in `s`.
    - We can even just say `s = get_string("Do you agree? ").lower()` to convert the input to lowercase immediately, before we store it in `s`.
- In Python, strings are also immutable, or unchangeable. When we make changes to a string, a new copy is made for us, along with all the memory management.

meow

- We can demonstrate design improvements to our `meow` program, too:

```
print("meow")
print("meow")
print("meow")
```

    - Here, we have the same line of code three times.
- We can use a loop:

```
for i in range(3):
    print("meow")
```

- We can define a function that we can reuse:

```
for i in range(3):
    meow()
```

```
def meow():
    print("meow")
```

- But this causes an error when we try to run it: `NameError: name 'meow' is not defined`. It turns out that, like in C, we need to define our function before we use it. So we'll define a `main` function first:

```
def main():
    for i in range(3):
        meow()

def meow():
    print("meow")

main()
```

- Now, by the time we actually call our `main` function at the end of our program, the `meow` function will already have been defined.
- The important part of our code will still be at the top of our file, so it's easy to find.

- We might also see examples that call a `main` function with:

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

- This solves problems with including our code in libraries, but we won't need to consider that yet, so we can simply call `main()`.

- Our functions can take arguments, too:

```
def main():
    meow(3)

def meow(n):
    for i in range(n):
        print("meow")

main()
```

- Our `meow` function takes in a parameter, `n`, and passes it to `range` in a for loop.
- Notice that we don't need to specify the type of an argument.

- We can create global variables by initializing them outside of `main`, though Python doesn't have constants.

Mario

- We can print out a row of hash symbols on the screen:

```
from cs50 import get_int

n = get_int("Height: ")

for i in range(n):
    print("#")
```

```
$ python mario.py
Height: -1
```

  - If the user passes in a negative number, we see no output. Instead, we should prompt the user again.

- In Python, there is no do while loop, but we can achieve the same effect:

```
from cs50 import get_int

while True:
    n = get_int("Height: ")
    if n > 0:
        break

for i in range(n):
    print("#")
```

  - We'll write an infinite loop, so we do something at least once, and then use `break` to exit the loop if we've met some condition.

- We can use a helper function:

```
from cs50 import get_int

def main():
    height = get_height()
    for i in range(height):
        print("#")

def get_height():
    while True:
        n = get_int("Height: ")
        if n > 0:
```

```
            break
    return n

main()
```

- Our `get_height()` function will return `n` after it meets our condition. Notice that, in Python, variables are scoped to a function, meaning we can use them outside of the loop they're created in.

- We can use `input` ourselves:

```
def main():
    height = get_height()
    for i in range(height):
        print("#")

def get_height():
    while True:
        n = int(input("Height: "))
        if n > 0:
            break
    return n

main()
```

```
$ python mario.py
Height: cat
Traceback (most recent call last):
  File "/workspaces/20377622/mario.py", line 13, in <module>
    main()
  File "/workspaces/20377622/mario.py", line 2, in main
    height = get_height()
  File "/workspaces/20377622/mario.py", line 8, in get_height
    n = int(input("Height: "))
ValueError: invalid literal for int() with base 10: 'cat'
```

- If we don't get a valid input, our traceback shows the stack of functions that led to the exception.

- We'll `try` to convert the input to an integer, and `print` a message if there is an exception. But we don't need to exit, since our loop will prompt the user again. If there isn't an exception, we'll continue to check if the value is positive and `break` if so:

```
def main():
    height = get_height()
    for i in range(height):
```

```
        print("#")

def get_height():
    while True:
        try:
            n = int(input("Height: "))
            if n > 0:
                break
        except ValueError:
            print("That's not an integer!")
    return n

main()
```

- Now we can try to print question marks on the same line:

```
for i in range(4):
    print("?", end="")
print()
```

```
$ python mario.py
????
```

  - When we print each question mark, we don't want the automatic new line, so we can pass a **named argument**, also known as keyword argument, to the `print` function. (So far, we've only seen **positional arguments**, where arguments are set based on their position in the function call.)
  - Here, we pass in `end=""` to specify that nothing should be printed at the end of our string. If we look at the documentation for `print`, we'll see that the default value for `end` is `\n`, a new line.
  - Finally, after we print our row with the loop, we can call `print` with no other arguments to get a new line.
- We can also use the multiply operator to join a string to itself many times, and print that directly with: `print("?" * 4)`.
- We can implement nested loops, too:

```
for i in range(3):
    for j in range(3):
        print("#", end="")
    print()
```

```
$ python mario.py
###
```

```
###
###
```

Documentation

- The official Python documentation (https://docs.python.org/) includes references for built-in functions.
- We can use the search function to find a page about functions that come with strings, for example, including the `lower()` (https://docs.python.org/3/library/stdtypes.html?highlight=lower#str.lower) function for converting a string to lowercase. On the same page, we'll see lots of other functions, though we shouldn't worry about learning all of them immediately.

Lists, strings

- We can create a list:

```
scores = [72, 73, 33]

average = sum(scores) / len(scores)
print(f"Average: {average}")
```

  - We can use `sum`, a function built into Python, to add up the values in our list, and divide it by the number of scores, using the `len` function to get the length of the list.

- We can add items to a list with:

```
from cs50 import get_int

scores = []
for i in range(3):
    score = get_int("Score: ")
    scores.append(score)

average = sum(scores) / len(scores)
print(f"Average: {average}")
```

  - With the `append` method, a function built into list objects, we can add new values to `scores`.
  - We can also join two lists with `scores += [score]`. Notice that we need to put `score` into a list of its own.

- We can iterate over each character in a string:

```
from cs50 import get_string

before = get_string("Before:  ")
print("After:  ", end="")
for c in before:
    print(c.upper(), end="")
print()
```

  - Python will iterate over each character in the string for us with just `for c in before:`.
- To make a string uppercase, we can also just write `after = before.upper()`, without having to iterate over each character ourselves.

Command-line arguments, exit codes

- We can take command-line arguments with:

```
from sys import argv

if len(argv) == 2:
    print(f"hello, {argv[1]}")
else:
    print("hello, world")
```

```
$ python argv.py
hello, world
$ python argv.py David
hello, David
```

  - We import `argv` from `sys`, the system module, built into Python.
  - Since `argv` is a list, we can get the second item with `argv[1]`.
  - `argv[0]` would be the name of our program, like `argv.py`, and not `python`.
- We can also let Python iterate over the list for us:

```
from sys import argv

for arg in argv:
    print(arg)
```

```
$ python argv.py
argv.py
```

```
$ python argv.py foo bar baz
argv.py
foo
bar
baz
```

- With Python, we can start at a different index in a list:

```
for arg in argv[1:]:
    print(arg)
```

  - This lets us **slice** the list from 1 to the end.
  - We can write `argv[:-1]` to get everything in the list except the last element.

- We can return exit codes when our program exits, too:

```
from sys import argv, exit

if len(argv) != 2:
    print("Missing command-line argument")
    exit(1)

print(f"hello, {argv[1]}")
exit(0)
```

  - Now, we can use `exit()` to exit our program with a specific code.

- We can import the entire `sys` library, and make it clear in our program where these functions come from:

```
import sys

if len(sys.argv) != 2:
    print("Missing command-line argument")
    sys.exit(1)

print(f"hello, {sys.argv[1]}")
sys.exit(0)
```

```
$ python exit.py
Missing command-line argument
$ python exit.py David
hello, David
```

Algorithms

- We can implement linear search by checking each element in a list:

```
import sys

numbers = [4, 6, 8, 2, 7, 5, 0]

if 0 in numbers:
    print("Found")
    sys.exit(0)

print("Not found")
sys.exit(1)
```

  - With `if 0 in numbers:`, we're asking Python to check the list for us.
- A list of strings, too, can be searched with:

```
import sys

names = ["Bill", "Charlie", "Fred", "George", "Ginny", "Percy", "Ron"]

if "Ron" in names:
    print("Found")
    sys.exit(0)

print("Not found")
sys.exit(1)
```

- If we have a dictionary, a set of key-value pairs, we can also check for a particular key, and look at the value stored for it:

```
from cs50 import get_string

people = {
    "Carter": "+1-617-495-1000",
    "David": "+1-949-468-2750"
}

name = get_string("Name: ")
if name in people:
    number = people[name]
    print(f"Number: {number}")
```

```
$ python phonebook.py
Name: David
Number: +1-949-468-2750
```

- We first declare a dictionary, `people`, where the keys are strings of each name we want to store, and the value for each key is a string of a corresponding phone number.
- Then, we use `if name in people:` to search the keys of our dictionary for a `name`. If the key exists, then we can get the value with the bracket notation, `people[name]`.

Files

- Let's open a CSV file, with comma-separated values:

```
import csv
from cs50 import get_string

file = open("phonebook.csv", "a")

name = get_string("Name: ")
number = get_string("Number: ")

writer = csv.writer(file)
writer.writerow([name, number])

file.close()
```

```
$ python phonebook.py
Name: Carter
Number: +1-617-495-1000
$ ls
phonebook.csv  phonebook.py
```

  - It turns out that Python also has a `csv` library that helps us work with CSV files, so after we open the file for appending, we can call `csv.writer` to create a `writer` object from the file. Then, we can use a method inside it, `writer.writerow`, to write a list as a row.
- Our `phonebook.csv` file will have our data:

```
Carter,+1-617-495-1000
```

  - We can run our program again, and see new data being added to our file.
- We can use the `with` keyword, which will close the file for us after we're finished:

```
...
```

```
with open("phonebook.csv", "a") as file:
    writer = csv.writer(file)
    writer.writerow((name, number))
```

- We'll visit a Google Form, and select a "house" (https://harrypotter.fandom.com/wiki/Hogwarts_Houses) we might want to be in.
- We'll download the data as a CSV file, which looks like this:

```
  Timestamp,House
  10/13/2021 16:00:07,Ravenclaw
  10/13/2021 16:00:07,Gryffindor
  10/13/2021 16:00:09,Ravenclaw
  10/13/2021 16:00:10,Gryffindor
  10/13/2021 16:00:10,Gryffindor
  ...
```

- Now we can tally the number of times a house appears:

```
import csv

houses = {
    "Gryffindor": 0,
    "Hufflepuff": 0,
    "Ravenclaw": 0,
    "Slytherin": 0
}

with open("hogwarts.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        house = row[1]
        houses[house] += 1

for house in houses:
    count = houses[house]
    print(f"{house}: {count}")
```

  - We use the `reader` function from the `csv` library, skip the header row with `next(reader)`, and then iterate over each of the rest of the rows.
  - The second item in each row, `row[1]`, is the string of a house, so we can use that to access the value stored in `houses` for that key, and add one to it with `houses[house] += 1`.
  - Finally, we'll print out the count for each house.

- We can improve our program by reading each row as a dictionary, using the first row in the file as the keys for each value:

```
...
with open("hogwarts.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        house = row["House"]
        houses[house] += 1
...
```

  - Now, we can say `house = row["House"]` to get the value in that column.

More libraries

- On our own Mac or PC, we can use another library to convert text to speech (since VS Code in the cloud doesn't support audio):

```
import pyttsx3

engine = pyttsx3.init()
engine.say("hello, world")
engine.runAndWait()
```

  - By reading the documentation, we can use a Python library called `pyttsx3` to play some string as audio.
  - We can even pass in a format string with `engine.say(f"hello, {name}")` to say some input.
- We can use another library, `face_recognition`, to find faces in images with `detect.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/faces/detect.py?highlight):

```
# Find faces in picture
# https://github.com/ageitgey/face_recognition/blob/master/examples/finc

from PIL import Image
import face_recognition

# Load the jpg file into a numpy array
image = face_recognition.load_image_file("office.jpg")

# Find all the faces in the image using the default HOG-based model.
# This method is fairly accurate, but not as accurate as the CNN model a
# See also: find_faces_in_picture_cnn.py
```

```
face_locations = face_recognition.face_locations(image)

for face_location in face_locations:

    # Print the location of each face in this image
    top, right, bottom, left = face_location

    # You can access the actual face itself like this:
    face_image = image[top:bottom, left:right]
    pil_image = Image.fromarray(face_image)
    pil_image.show()
```

- In `recognize.py` (https://cdn.cs50.net/2020/fall/lectures/6/src6/6/faces/recognize.py), we can see a program that finds a match for a particular face.
- In `listen0.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/listen/listen0.py?highlight), we can respond to input from the user:

```
# Recognizes a greeting

# Get input
words = input("Say something!\n").lower()

# Respond to speech
if "hello" in words:
    print("Hello to you too!")
elif "how are you" in words:
    print("I am well, thanks!")
elif "goodbye" in words:
    print("Goodbye to you too!")
else:
    print("Huh?")
```

- We can recognize audio input from a microphone and respond with `listen2.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/listen/listen2.py?highlight):

```
# Responds to a greeting
# https://pypi.org/project/SpeechRecognition/

import speech_recognition

# Obtain audio from the microphone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() as source:
    print("Say something:")
    audio = recognizer.listen(source)
```

```
# Recognize speech using Google Speech Recognition
words = recognizer.recognize_google(audio)

# Respond to speech
if "hello" in words:
    print("Hello to you too!")
elif "how are you" in words:
    print("I am well, thanks!")
elif "goodbye" in words:
    print("Goodbye to you too!")
else:
    print("Huh?")
```

- We can even add more logic to listen for a name:

```
# Responds to a name
# https://pypi.org/project/SpeechRecognition/

import re
import speech_recognition

# Obtain audio from the microphone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() as source:
    print("Say something:")
    audio = recognizer.listen(source)

# Recognize speech using Google Speech Recognition
words = recognizer.recognize_google(audio)

# Respond to speech
matches = re.search("my name is (.*)", words)
if matches:
    print(f"Hey, {matches[1]}.")
else:
    print("Hey, you.")
```

- We can create a QR code (https://en.wikipedia.org/wiki/QR_code), or two-dimensional barcode, with another library:

```
import os
import qrcode

img = qrcode.make("https://youtu.be/xvFZjo5PgG0")
img.save("qr.png", "PNG")
```

```
os.system("open qr.png")
```

- Now, when we run our program, a QR code will be generated and opened.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

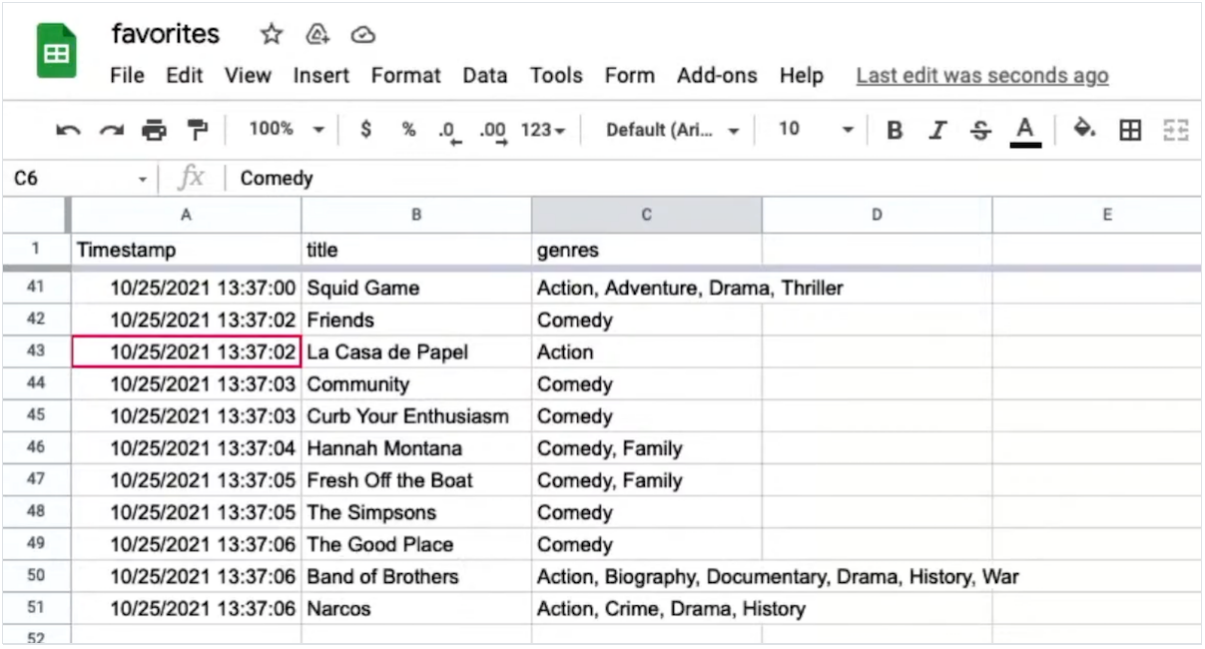
Lecture 7

Data processing

- Last week, we collected a survey of Hogwarts house (https://harrypotter.fandom.com

/wiki/Hogwarts_Houses) preferences, and tallied the data from a CSV file with Python.

- This week, we'll collect some more data about your favorite TV shows and their genres.
- We get hundreds of responses from the audience, and start looking at them on Google Sheets, a web-based spreadsheet application, showing our data in rows and columns:



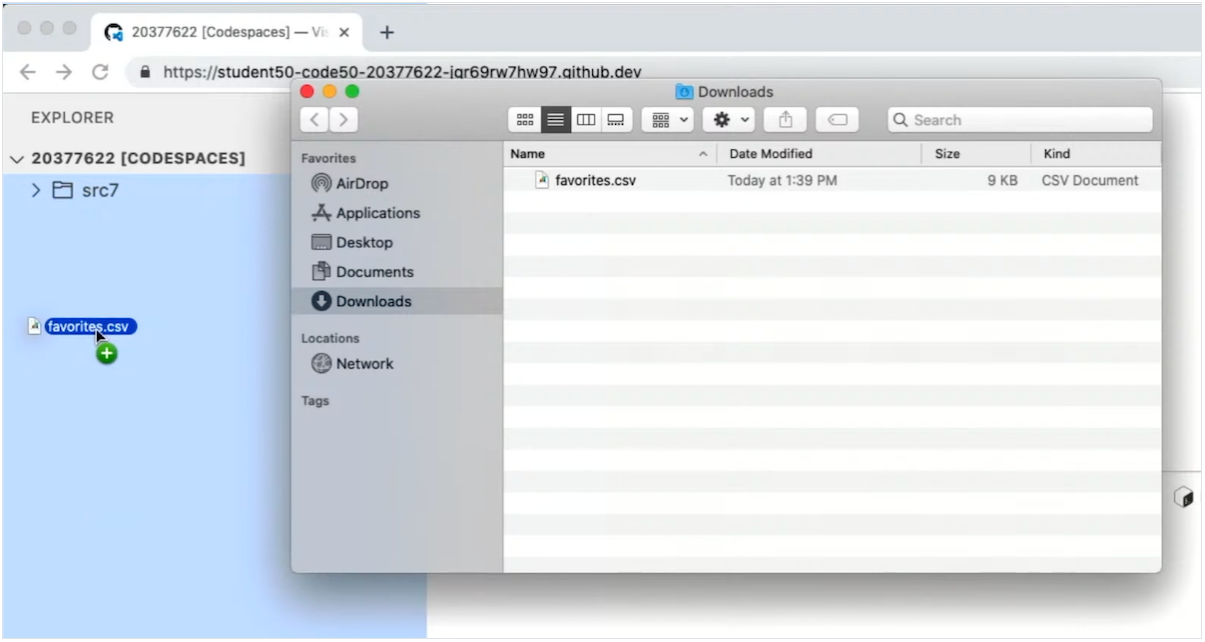
favorites

File Edit View Insert Format Data Tools Form Add-ons Help Last edit was seconds ago

C6 | Comedy

	A	B	C	D	E
1	Timestamp	title	genres		
41	10/25/2021 13:37:00	Squid Game	Action, Adventure, Drama, Thriller		
42	10/25/2021 13:37:02	Friends	Comedy		
43	10/25/2021 13:37:02	La Casa de Papel	Action		
44	10/25/2021 13:37:03	Community	Comedy		
45	10/25/2021 13:37:03	Curb Your Enthusiasm	Comedy		
46	10/25/2021 13:37:04	Hannah Montana	Comedy, Family		
47	10/25/2021 13:37:05	Fresh Off the Boat	Comedy, Family		
48	10/25/2021 13:37:05	The Simpsons	Comedy		
49	10/25/2021 13:37:06	The Good Place	Comedy		
50	10/25/2021 13:37:06	Band of Brothers	Action, Biography, Documentary, Drama, History, War		
51	10/25/2021 13:37:06	Narcos	Action, Crime, Drama, History		
52					

- Like we did last week, we can download our data as a CSV file, which is an example of a **flat-file database**, where the data for each column is separated by commas, and each row is on a new line, saved simply as a text file in ASCII or Unicode.
    - A flat-file database is completely portable, which means that we can open it on nearly any operating system without special software like Microsoft Excel or Apple Numbers.
- We'll upload the CSV file to our instance of VS Code by dragging and dropping it:



- Then, we'll see the file opened in an editor:

favorites.csv

```
Timestamp,title,genres
10/25/2021 11:21:46,How i met your mother,Comedy
10/25/2021 12:19:26,The Sopranos,"Comedy, Crime, Drama, Horror, Sci-Fi, Talk-Show, Thriller"
10/25/2021 13:34:34,Friday Night Lights,"Drama, Family, Sport"
10/25/2021 13:36:10,Family Guy,"Animation, Comedy"
10/25/2021 13:36:18,New Girl,Comedy
10/25/2021 13:36:22,Friends,Comedy
10/25/2021 13:36:23,The Office,Comedy
10/25/2021 13:36:25,Breaking Bad,"Crime, Drama"
10/25/2021 13:36:28,Modern Family,Comedy
10/25/2021 13:36:34,The Office,Comedy
10/25/2021 13:36:35,White Collar,"Action, Crime, Drama"
10/25/2021 13:36:36,Modern Family,"Comedy, Family"
10/25/2021 13:36:36,Game of Thrones,"Action, Adventure, Drama, Fantasy, Thriller, War"
10/25/2021 13:36:36,the Untamed,"Action, Drama, Fantasy, History, Romance"
10/25/2021 13:36:36,The 100,"Action, Sci-Fi"
10/25/2021 13:36:37,Modern Family,"Comedy, Family"
10/25/2021 13:36:38,The Office,Comedy
10/25/2021 13:36:38,New Girl,Comedy
10/25/2021 13:36:39,The Office,Comedy
10/25/2021 13:36:40,Avatar: The Last Airbender,"Adventure, Animation"
10/25/2021 13:36:40,Cobra Kai,"Action, Sport"
```

- Notice that some rows have multiple genres, and those are surrounded by quotes, like `"Crime, Drama"`, so that the commas *within* our data aren't misinterpreted.

- Let's write a new program, `favorites.py`, to read our CSV file:

```
import csv

with open("favorites.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        print(row[1])
```

  - We'll open the file with a reference called `file`, using the `with` keyword in Python that will close our file for us.
  - The `csv` library has a `reader` function that will create a `reader` variable we can use to read in the file as a CSV.
  - We'll call `next` to skip the first row, since that's the header row.
  - Then, we'll use a loop to print the second column in each row, which is the title.

- Now, if we run our program, we'll see a list of show titles:

```
$ python favorites.py
...
Friends
...
friends
...
Friends
...
```

  - But for the show titled "Friends", some entries are capitalized and some are lowercased.

Cleaning

- To improve our program, we'll first use a `DictReader`, dictionary reader, which creates a dictionary from each row, allowing us to access each column by its name. We also don't need to skip the header row in this case, since the `DictReader` will use it automatically.

```
import csv

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        print(row["title"])
```

  - Since the first row in our CSV has the names of the columns, it can be used to label each column in our data as well. Now our program will still work, even if the order of the columns are changed.

- Now let's try to filter out duplicates in our responses:

```
import csv

titles = []

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        if not row["title"] in titles:
            titles.append(row["title"])

for title in titles:
    print(title)
```

  - We'll make a new list called `titles`, and only add each row's title if it's not already in the list. Then, we can print all the titles:

```
$ python favorites.py
...
Friends
...
friends
...
```

    - We see that there are still near-duplicates, since `Friends` and `friends` are

indeed different strings still.

- We'll want to change the current title to all uppercase, and remove whitespace around it, before we add it to our list:

```
import csv

titles = []

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if not title in titles:
            titles.append(title)

for title in titles:
    print(title)
```

  - Now, we've **canonicalized**, or standardized, our data, and our list of titles are much cleaner:

```
$ python favorites.py
...
NEW GIRL
FRIENDS
THE OFFICE
BREAKING BAD
...
```

- It turns out that Python has another data structure built-in, `set`, which ensures that all the values are unique:

```
import csv

titles = set()

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        titles.add(title)

for title in titles:
```

```
    print(title)
```

- Now, we can call `add` on the set, and not have to check ourselves if it's already in the set.

- To sort the titles, we can just change our loop to `for title in sorted(titles)`, which will sort our set before we iterate over it:

```
import csv

titles = set()

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        titles.add(title)

for title in sorted(titles):
    print(title)
```

```
$ python favorites.py
ADVENTURE TIME
ANNE WITH AN E
...
AVATAR
AVATAR THE LAST AIRBENDER
AVATAR: THE LAST AIRBENDER
...
BROOKLYN 99
BROOKLYN-99
...
```

- Now, we see our titles alphabetized, but there were still a few different ways that a show's title could be entered. We'll leave these differences there for now, since it will likely take a bit more effort to fully standardize our data.

Counting

- We can use a dictionary, instead of a set, to count the number of times we've seen each title, with the keys being the titles and the values being an integer counting the number of times we see each of them:

```
import csv
```

```
titles = {}

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        titles[title] += 1

for title in sorted(titles):
    print(title)
```

- As we read each row, we increase the value stored for that title in the dictionary by 1.

- We'll run this program, and see:

```
$ python favorites.py
Traceback (most recent call last):
  File "/workspaces/20377622/favorites.py", line 9, in <module>
    titles[title] += 1
KeyError: 'HOW I MET YOUR MOTHER'
```

- We have a `KeyError`, since the title `HOW I MET YOUR MOTHER` isn't in the dictionary yet.

- We'll have to add each title to our dictionary first, and set the initial value to 1:

```
import csv

titles = {}

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if title in titles:
            titles[title] += 1
        else:
            titles[title] = 1

for title in sorted(titles):
    print(title, titles[title])
```

- We'll add the values, or counts, to our loop that prints every show name.

- We can also set the initial value to 0, and then increment it by 1 no matter what:

```
import csv

titles = {}

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if not title in titles:
            titles[title] = 0
        titles[title] += 1

for title in sorted(titles):
    print(title, titles[title])
```

```
$ python favorites.py
ADVENTURE TIME 1
ANNE WITH AN E 1
ARCHER 1
...
AVATAR THE LAST AIRBENDER 5
...
COMMUNITY 8
...
```

  - Now, the key will exist in the dictionary, and we can safely refer to its value in the dictionary.

- We can sort by the values in the dictionary by changing our loop to:

```
...
def get_value(title):
    return titles[title]

for title in sorted(titles, key=get_value, reverse=True):
    print(title, titles[title])
```

  - We define a function, `f`, which just returns the value of a title in the dictionary with `titles[title]`. The `sorted` function, in turn, will take in that function as the key to sort the dictionary. And we'll also pass in `reverse=True` to sort from largest to smallest, instead of smallest to largest.
  - So now we'll see the most popular shows printed:

```
$ python favorites.py
THE OFFICE 15
FRIENDS 9
COMMUNITY 8
GAME OF THRONES 6
...
```

- We can actually define our function in the same line, with this syntax:

```
for title in sorted(titles, key=lambda title: titles[title], reverse=Tru
    print(title, titles[title])
```

  - We can write and pass in a **lambda**, or anonymous function, which has no name but takes in some argument or arguments, and returns a value immediately.
  - Notice that there are no parentheses or `return` keyword, but concisely has the same effect as our `get_value` function earlier.
- We can also try to count all the occurrences of a specific title:

```
import csv

counter = 0

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if title == "THE OFFICE":
            counter += 1

print(f"Number of people who like The Office: {counter}")
```

```
$ python favorites.py
Number of people who like The Office: 15
```

  - We'll have a simple `counter` variable, and add one to it.
- Now, if our data referred to the same show in different ways, we can try to check if the word "OFFICE" was in the title at all:

```
import csv

counter = 0

with open("favorites.csv", "r") as file:
```

```
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if "OFFICE" in title:
            counter += 1

print(f"Number of people who like The Office: {counter}")
```

```
$ python favorites.py
Number of people who like The Office: 16
```

- It turns out that a row has a typo, “Thevoffice”, so now our count is correct.

- We can also use **regular expressions**, a standardized way to represent a pattern that a string must match.
- For example, we can write a regular expression that matches email addresses:

```
.*@.*\..*
```

- The first period, `.`, indicates any character. The following asterisk, `*`, indicates 0 or more times. Then, we want an at sign, `@`. Then we want 0 or more characters again, `.*`, and then a literal period in our string, escaped with `\.`. Finally, we want 0 or more characters again with `.*`.

- Since we probably want at least 1 character in each segment of an email address, we should change our regular expression to:

```
.+@.+\..+
```

- The plus sign, `+`, means we are matching for the previous character 1 or more times.
- We can restrict the domain of the email to `.edu` by changing our regular expression to `.+@.+\.edu`.

- Languages like Python and JavaScript support regular expressions, which are like a mini-language in themselves, with syntax like:
  - `.` for any character
  - `.*` for 0 or more characters
  - `.+` for 1 or more characters
  - `?` for an optional character
  - `^` for start of input
  - `$` for end of input

    - ...
- We can change our program earlier to use `re`, a Python library for regular expressions:

```
import csv
import re

counter = 0

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if re.search("OFFICE", title):
            counter += 1

print(f"Number of people who like The Office: {counter}")
```

```
$ python favorites.py
Number of people who like The Office: 16
```

    - The `re` library has a function, `search`, to which we can pass a pattern and string to see if there is a match.
    - We can change our expression to `"^(OFFICE|THE OFFICE)$"`, which will match either `OFFICE` or `THE OFFICE`, but only if they start at the beginning of the string, and stop at the end of the string (i.e., there are no other words before or after).
    - We can even change `THE OFFICE` to `THE.OFFICE`, allowing any character (like a typo) to be in between those words.
- We can also write a program to ask the user for a particular title and report its popularity:

```
import csv

title = input("Title: ").strip().upper()

counter = 0

with open("favorites.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        if row["title"].strip().upper() == title:
            counter += 1

print(counter)
```

```
$ python favorites.py
Title: the office
13
```

- We ask the user for input, and then open our CSV file. Since we're looking for just one title, we can have one `counter` variable that we increment.
- We check for a match after standardizing both the user's input and each row's title.

Relational databases

- **Relational databases** are programs that store data, ultimately in files, but with additional data structures and interfaces that allow us to search and store data more efficiently.
- When working with data, we generally need four types of basic operations with the acronym `CRUD`:
  - `CREATE`
  - `READ`
  - `UPDATE`
  - `DELETE`

SQL

- With another programming language, **SQL** (pronounced like "sequel"), we can interact with databases with verbs like:
  - `CREATE`, `INSERT`
  - `SELECT`
  - `UPDATE`
  - `DELETE`, `DROP`
- Syntax in SQL might look like:

```
CREATE TABLE table (column type, ...);
```

  - With this statement, we can create a **table**, which is like a spreadsheet with rows and columns.
  - In SQL, we choose the types of data that each column will store.
- We'll use a common database program called **SQLite**, one of many available programs that support SQL. Other database programs include Oracle Database, MySQL, PostgreSQL, and Microsoft Access.
- SQLite stores our data in a binary file, with 0s and 1s that represent data efficiently. We'll

interact with our tables of data through a command-line program, `sqlite3`.

- We'll run some commands in VS Code to import our CSV file into a database:

```
$ sqlite3 favorites.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import favorites.csv favorites
```

  - First, we'll run the `sqlite3` program with `favorites.db` as the name of the file for our database.
  - With `.import`, SQLite creates a table in our database with the data from our CSV file.

- Now, we'll see three files, including `favorites.db`:

```
$ ls
favorites.csv  favorites.db  favorites.py
```

- We can open our database file again, and check the schema, or design, of our new table with `.schema`:

```
$ sqlite3 favorites.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE IF NOT EXISTS "favorites"(
  "Timestamp" TEXT,
  "title" TEXT,
  "genres" TEXT
);
```

  - We see that `.import` used the `CREATE TABLE ...` command to create a table called `favorites`, with column names automatically copied from the CSV's header row, and types for each of them assumed to be text.

- We can select, or read data, with:

```
sqlite> SELECT title FROM favorites;
+-----------------------------------+
|               title               |
+-----------------------------------+
| How i met your mother             |
| The Sopranos                      |
| Friday Night Lights               |
...
```

    - With a command in the format `SELECT columns FROM table;`, we can read data from one or more columns. For example, we can write `SELECT title, genre FROM favorites;` to select both the title and genre.
- SQL supports many functions that we can use to count and summarize data:
    - `AVG`
    - `COUNT`
    - `DISTINCT`
    - `LOWER`
    - `MAX`
    - `MIN`
    - `UPPER`
    - ...
- We can clean up our titles as before, converting them to uppercase and printing only the unique values:

```
sqlite> SELECT DISTINCT(UPPER(title)) FROM shows;
...
| LAW AND ORDER                    |
| B99                              |
| GOT                              |
...
```

- We can also get a count of how many responses there are:

```
sqlite> SELECT COUNT(title) FROM favorites;
+--------------+
| COUNT(title) |
+--------------+
| 158          |
+--------------+
```

- We can also add more phrases to our command:
    - `WHERE`, adding a Boolean expression to filter our data
    - `LIKE`, filtering responses more loosely
    - `ORDER BY`
    - `LIMIT`
    - `GROUP BY`
    - ...
- We can limit the number of results:

```
sqlite> SELECT title FROM favorites LIMIT 10;
+----------------------+
|         title        |
+----------------------+
| How i met your mother |
| The Sopranos         |
| Friday Night Lights  |
| Family Guy           |
| New Girl             |
| Friends              |
| Office               |
| Breaking Bad         |
| Modern Family        |
| Office               |
+----------------------+
```

- We can also look for titles matching a string:

```
sqlite> SELECT title FROM favorites WHERE title LIKE "%office%";
+-------------+
|    title    |
+-------------+
| Office      |
| Office      |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| The Office  |
| the office  |
| The Office  |
| ThE OffiCE  |
| The Office  |
| Thevoffice  |
+-------------+
```

  - The `%` character is a placeholder for zero or more other characters, so SQL supports some pattern matching, though not it's not as powerful as regular expressions are.

- We can select just the count in our command:

```
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%office%";
```

```
+--------------+
| COUNT(title) |
+--------------+
| 16           |
+--------------+
```

- If we don't like a show, we can even delete it:

```
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%friends%";
+--------------+
| COUNT(title) |
+--------------+
| 9            |
+--------------+
sqlite> DELETE FROM favorites WHERE title LIKE "%friends%";
sqlite> SELECT COUNT(title) FROM favorites WHERE title LIKE "%friends%";
+--------------+
| COUNT(title) |
+--------------+
| 0            |
+--------------+
```

    - With SQL, we can change our data more easily and quickly than with Python.
- We can update a specific row of data:

```
sqlite> SELECT title FROM favorites WHERE title = "Thevoffice";
+------------+
|   title    |
+------------+
| Thevoffice |
+------------+
sqlite> UPDATE favorites SET title = "The Office" WHERE title = "Thevoff
sqlite> SELECT title FROM favorites WHERE title = "Thevoffice";
sqlite>
```

    - Now, we've changed that row's value.
- We can change the values in multiple rows, too:

```
sqlite> SELECT genres FROM favorites WHERE title = "Game of Thrones";
+-------------------------------------------------------------------------
|                                                  genres
+-------------------------------------------------------------------------
| Action, Adventure, Drama, Fantasy, Thriller, War
| Action, Adventure, Drama
| Action, Adventure, Comedy, Drama, Family, Fantasy, History, Horror, Mu
```

```
| Action, Drama, Family, Fantasy, War
| Fantasy, Thriller, War
+-------------------------------------------------------------------------
sqlite> UPDATE favorites SET genres = "Action, Adventure, Drama, Fantasy
sqlite> SELECT genres FROM favorites WHERE title = "Game of Thrones";
+-----------------------------------------------+
|                    genres                     |
+-----------------------------------------------+
| Action, Adventure, Drama, Fantasy, Thriller, War |
| Action, Adventure, Drama, Fantasy, Thriller, War |
| Action, Adventure, Drama, Fantasy, Thriller, War |
| Action, Adventure, Drama, Fantasy, Thriller, War |
| Action, Adventure, Drama, Fantasy, Thriller, War |
+-----------------------------------------------+
```

- With `DELETE` and `DROP`, we can remove rows and even entire tables as well.
- And notice that in our commands, we've written SQL keywords in all caps, so they stand out more.
- There also isn't a built-in way to undo commands, so if we make a mistake we might have to build our database again!

Tables

- We'll take a look at our schema again:

```
sqlite> .schema
CREATE TABLE IF NOT EXISTS "favorites"(
  "Timestamp" TEXT,
  "title" TEXT,
  "genres" TEXT
);
```

- If we look at our values of genres, we see some redundancy:

```
sqlite> SELECT genres FROM favorites;
+-----------------------------------------------------------+
|                          genres                           |
+-----------------------------------------------------------+
| Comedy                                                    |
| Comedy, Crime, Drama, Horror, Sci-Fi, Talk-Show, Thriller |
| Drama, Family, Sport                                      |
| Animation, Comedy                                         |
| Comedy, Drama                                             |
...
```

- And if we want to search for shows that are comedies, we have to search with not just `SELECT title FROM favorites WHERE genre = "Comedy";`, but also `... WHERE genre = "Comedy, Drama";`, `... WHERE genre = "Comedy, News";`, and so on.
- We can use the `LIKE` keyword again, but two genres, "Music" and "Musical", are similar enough for that to be problematic.
- We can actually write our own Python program that will use SQL to import our CSV data into *two* tables:

```
# Imports titles and genres from CSV into a SQLite database

import cs50
import csv

# Create database
open("favorites8.db", "w").close()
db = cs50.SQL("sqlite:///favorites8.db")

# Create tables
db.execute("CREATE TABLE shows (id INTEGER, title TEXT NOT NULL, PRIMARY
db.execute("CREATE TABLE genres (show_id INTEGER, genre TEXT NOT NULL, F

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Iterate over CSV file
    for row in reader:

        # Canoncalize title
        title = row["title"].strip().upper()

        # Insert title
        show_id = db.execute("INSERT INTO shows (title) VALUES(?)", titl

        # Insert genres
        for genre in row["genres"].split(", "):

            db.execute("INSERT INTO genres (show_id, genre) VALUES(?, ?)
```

  - First, we import the Python `cs50` library so we can run SQL commands more easily.
  - Then, the rest of this code will import each row of `favorites.csv`.

- Now, our database will have this design:

```
$ sqlite3 favorites8.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE shows (id INTEGER, title TEXT NOT NULL, PRIMARY KEY(id));
CREATE TABLE genres (show_id INTEGER, genre TEXT NOT NULL, FOREIGN KEY(s
```

  - We have one table, `shows`, with an `id` column and a `title` column. We can specify that a `title` isn't null, and that `id` is the column we want to use as a primary key.
  - Then, we'll have a table called `genres`, where we have a `show_id` column that references our `shows` table, along with a `genre` column.
  - This is an example of a **relation**, like a link, between rows in different tables in our database.
- In our `shows` table, we'll see each show with an `id` number:

```
sqlite> SELECT * FROM shows;
+-----+---------------------------------+
| id  |               title             |
+-----+---------------------------------+
| 1   | HOW I MET YOUR MOTHER           |
| 2   | THE SOPRANOS                    |
| 3   | FRIDAY NIGHT LIGHTS             |
| 4   | FAMILY GUY                      |
| 5   | NEW GIRL                        |
| 6   | FRIENDS                         |
| 7   | OFFICE                          |
...
```

- And we can see that the `genres` table has one or more rows for each `show_id`:

```
sqlite> SELECT * FROM genres;
+---------+-------------+
| show_id |    genre    |
+---------+-------------+
| 1       | Comedy      |
| 2       | Comedy      |
| 2       | Crime       |
| 2       | Drama       |
| 2       | Horror      |
| 2       | Sci-Fi      |
| 2       | Talk-Show   |
```

```
| 2       | Thriller    |
| 3       | Drama       |
| 3       | Family      |
| 3       | Sport       |
| 4       | Animation   |
| 4       | Comedy      |
| 5       | Comedy      |
| 6       | Comedy      |
| 7       | Comedy      |
...
```

  - Since each show may have more than one genre, we can have more than one row per show in our `genres` table, known as a **one-to-many** relationship.
  - Furthermore, the data is now cleaner, since each genre name is in its own row.
- We can select all the shows are that comedies by selecting from the `genres` table first, and then looking for those `id`s in the `shows` table:

```
sqlite> SELECT title FROM shows WHERE id IN (SELECT show_id FROM genres
+-----------------------------------+
|               title               |
+-----------------------------------+
| HOW I MET YOUR MOTHER             |
| THE SOPRANOS                      |
| FAMILY GUY                        |
| NEW GIRL                          |
| FRIENDS                           |
| OFFICE                            |
| MODERN FAMILY                     |
...
```

  - Notice that we've nested two queries, where the inner one returns a list of show `id`s, and the outer one uses those to select the titles of shows that match.
- Now we can sort and show just the unique titles by adding to our command:

```
sqlite> SELECT DISTINCT(title) FROM shows WHERE id IN (SELECT show_id FR
+-----------------------------------+
|               title               |
+-----------------------------------+
| ARCHER                            |
| ARRESTED DEVELOPMENT              |
| AVATAR THE LAST AIRBENDER         |
| B99                               |
| BILLIONS                          |
| BLACK MIRROR                      |
```

```
...
```

- And we can add new data to each table, in order to add another show. First, we'll add a new row to the `shows` table for Seinfeld:

```
sqlite> INSERT INTO shows (title) VALUES("Seinfeld");
```

- Then, we can get our row's `id` by looking for it in the table:

```
sqlite> SELECT * FROM shows WHERE title = "Seinfeld";
+-----+----------+
| id  |  title   |
+-----+----------+
| 159 | Seinfeld |
+-----+----------+
```

- We'll use that as the `show_id` to add a new row in the `genres` table:

```
sqlite> INSERT INTO genres (show_id, genre) VALUES(159, "Comedy");
```

- Then, we'll use `UPDATE` to set the title to uppercase:

```
sqlite> UPDATE shows SET title = "SEINFELD" WHERE title = "Seinfeld";
```

- Finally, we'll run the same command as before, and see our new show is indeed in the list of comedies:

```
sqlite> SELECT DISTINCT(title) FROM shows WHERE id IN (SELECT show_id FR
...
| SEINFELD                       |
...
```

SQL with Python

- It turns out that we'll be able to write Python code that automates this, so we can imagine building web applications that can programmatically store and look up user data, online shopping orders, and more.
- We can write a program that asks the user for a show title and then prints its popularity:

```
import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")
```

```
title = input("Title: ").strip()

rows = db.execute("SELECT COUNT(*) AS counter FROM favorites WHERE title

row = rows[0]

print(row["counter"])
```

- We'll use the `cs50` library to run SQL commands more easily, and open the `favorites.db` database we created earlier.
- We'll prompt the user for a title, and then execute a command. A `?` in the command will allow us to safely substitute variables in our command.
- The results are returned in a list of rows, and `COUNT(*)` returns just one row. In our command, we'll add `AS counter`, so the count is returned in the row (which is a dictionary) with the column name `counter`.

- We can run our program and search for "The Office":

```
$ python favorites.py
Title: The Office
12
```

- And we can tweak our program to print all the rows that match:

```
import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")

title = input("Title: ").strip()

rows = db.execute("SELECT title FROM favorites WHERE title LIKE ?", titl

for row in rows:
    print(row["title"])
```

```
$ python favorites.py
Title: The Office
The Office
The Office
The Office
The Office
The Office
```

```
The Office
The Office
The Office
the office
The Office
ThE OffiCE
The Office
The Office
```

- Since `LIKE` is case-insensitive, we see all the various ways the titles were capitalized.

IMDb

- IMDb, or the Internet Movie Database, has datasets available for download as TSV (tab-separated values) files.
- We'll open a database that the staff has created beforehand:

```
$ sqlite3 shows.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE shows (
                    id INTEGER,
                    title TEXT NOT NULL,
                    year NUMERIC,
                    episodes INTEGER,
                    PRIMARY KEY(id)
                );
CREATE TABLE genres (
                    show_id INTEGER NOT NULL,
                    genre TEXT NOT NULL,
                    FOREIGN KEY(show_id) REFERENCES shows(id)
                );
CREATE TABLE stars (
                show_id INTEGER NOT NULL,
                person_id INTEGER NOT NULL,
                FOREIGN KEY(show_id) REFERENCES shows(id),
                FOREIGN KEY(person_id) REFERENCES people(id)
            );
CREATE TABLE writers (
                show_id INTEGER NOT NULL,
                person_id INTEGER NOT NULL,
                FOREIGN KEY(show_id) REFERENCES shows(id),
```

```
                FOREIGN KEY(person_id) REFERENCES people(id)
            );
CREATE TABLE ratings (
                show_id INTEGER NOT NULL,
                rating REAL NOT NULL,
                votes INTEGER NOT NULL,
                FOREIGN KEY(show_id) REFERENCES shows(id)
            );
CREATE TABLE people (
                id INTEGER,
                name TEXT NOT NULL,
                birth NUMERIC,
                PRIMARY KEY(id)
            );
```

    - Notice that we have multiple tables, each of which has columns of various data types.
    - In both the `stars` and `writers` table, for example, we have a `show_id` column that references the `id` of some row in the `shows` table, and a `person_id` column that references the `id` of some row in the `people` table. Effectively, they link shows and people by their `id`s.
- It turns out that SQL, too, has its own data types:
    - **`BLOB`**, for "binary large object", raw binary data that might represent files
    - **`INTEGER`**
    - **`NUMERIC`**, number-like but not quite a number, like a date or time
    - **`REAL`**, for floating-point values
    - **`TEXT`**, like strings
- Columns can also have additional attributes:
    - **`PRIMARY KEY`**, like the `id` columns above that will be used to uniquely identify each row
    - **`FOREIGN KEY`**, like the `show_id` column above that refers to a column in some other table
- We can see that there are millions of rows in the `people` table:

```
sqlite> SELECT * FROM people;
...
| 13058200 | Emilio Mancuso                                          |
| 13058201 | Pietro Furnis                                           |
| 13058202 | Ida Lonati Frati                                        |
+----------+---------------------------------------------------------+------
```

- But like before, we can search for just one row:

```
sqlite> SELECT * FROM people WHERE name = "Steve Carell";
+--------+--------------+--------+
|   id   |     name     | birth  |
+--------+--------------+--------+
| 136797 | Steve Carell | 1962   |
+--------+--------------+--------+
```

- It turns out that there are a few shows titled “The Office”:

```
sqlite> SELECT * FROM shows WHERE title = "The Office";
+---------+------------+------+----------+
|   id    |   title    | year | episodes |
+---------+------------+------+----------+
| 112108  | The Office | 1995 | 6        |
| 290978  | The Office | 2001 | 14       |
| 386676  | The Office | 2005 | 188      |
| 1791001 | The Office | 2010 | 30       |
| 2186395 | The Office | 2012 | 8        |
| 8305218 | The Office | 2019 | 28       |
+---------+------------+------+----------+
```

- The most popular one, with 188 episodes, is the one we want, so we can get just that one:

```
sqlite> SELECT * FROM shows WHERE title = "The Office" and year = "2005"
+--------+------------+------+----------+
|   id   |   title    | year | episodes |
+--------+------------+------+----------+
| 386676 | The Office | 2005 | 188      |
+--------+------------+------+----------+
```

- We can turn on a timer and see that our original command took about 0.02 seconds to run:

```
sqlite> .timer on
sqlite> SELECT * FROM shows WHERE title = "The Office";
+---------+------------+------+----------+
|   id    |   title    | year | episodes |
+---------+------------+------+----------+
| 112108  | The Office | 1995 | 6        |
| 290978  | The Office | 2001 | 14       |
| 386676  | The Office | 2005 | 188      |
| 1791001 | The Office | 2010 | 30       |
| 2186395 | The Office | 2012 | 8        |
```

```
| 8305218 | The Office | 2019 | 28       |
+---------+------------+------+----------+
Run Time: real 0.021 user 0.016419 sys 0.004117
```

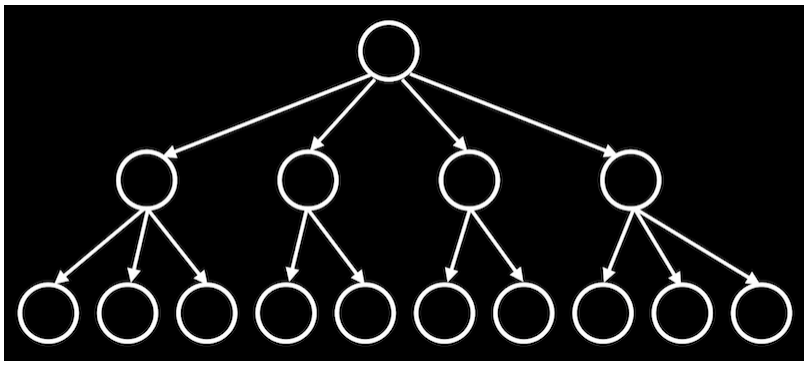
- We can create an **index**, or additional data structures that our database program will use for future searches:

```
sqlite> CREATE INDEX "title_index" ON "shows" ("title");
Run Time: real 0.349 user 0.195206 sys 0.051217
```

- Now, our search command takes nearly no time:

```
sqlite> SELECT * FROM shows WHERE title = "The Office";
+---------+------------+------+----------+
|   id    |   title    | year | episodes |
+---------+------------+------+----------+
| 112108  | The Office | 1995 | 6        |
| 290978  | The Office | 2001 | 14       |
| 386676  | The Office | 2005 | 188      |
| 1791001 | The Office | 2010 | 30       |
| 2186395 | The Office | 2012 | 8        |
| 8305218 | The Office | 2019 | 28       |
+---------+------------+------+----------+
Run Time: real 0.000 user 0.000104 sys 0.000124
```

- It turns out that these data structures are generally **B-trees**, like binary trees we've seen in C but with more children, with nodes organized such that we can search faster than linearly:



  - Creating an index takes some time up front, perhaps by sorting the data, but afterwards we can search much more quickly.

- With our data spread among different tables, we can nest our queries to get useful data. For example, we can get all the titles of shows starring a particular person:

```
sqlite3> SELECT title FROM shows WHERE id IN (SELECT show_id FROM stars
+----------------------------------+
|              title               |
```

```
+-----------------------------------+
| The Dana Carvey Show              |
| Over the Top                      |
| Watching Ellie                    |
| Come to Papa                      |
| The Office                        |
...
```

    - We'll `SELECT` the `title` from the `shows` table for shows with an `id` that matches a list of `show_id`s from the `stars` table. Those `show_id`s, in turn, must have a `person_id` that matches the `id` of Steve Carell in the `people` table.
- Our query runs pretty quickly, but we can create a few more indexes:

```
sqlite> CREATE INDEX person_index ON stars (person_id);
Run Time: real 0.890 user 0.662294 sys 0.097505
sqlite> CREATE INDEX show_index ON stars (show_id);
Run Time: real 0.644 user 0.469162 sys 0.058866
sqlite> CREATE INDEX name_index ON people (name);
Run Time: real 0.840 user 0.609600 sys 0.088177
```

    - Each index takes almost a second to build, but afterwards, our same query takes very little time to run.
- It turns out that we can use `JOIN` commands to combine tables in our queries:

```
sqlite> SELECT title FROM people
   ...> JOIN stars ON people.id = stars.person_id
   ...> JOIN shows ON stars.show_id = shows.id
   ...> WHERE name = "Steve Carell";
```

    - With the `JOIN` syntax, we can virtually combine tables based on their foreign keys, and use their columns as though they were one table. Here, we're matching the `people` table with the `stars` table, and then with the `shows` table.
- We can format the same query a little better by listing the tables we want to use all at once:

```
sqlite> SELECT title FROM people, stars, shows
   ...> WHERE people.id = stars.person_id
   ...> AND stars.show_id = shows.id
   ...> AND name = "Steve Carell";
+-----------------------------------+
|               title               |
+-----------------------------------+
| The Dana Carvey Show              |
| Over the Top                      |
```

```
| Watching Ellie                    |
| Come to Papa                      |
| The Office                        |
```

- The downside to having lots of indexes is that each of them take up some amount of space, which might become significant with lots of data and lots of indexes.

Problems

- One problem in SQL is called a **SQL injection attack**, where an someone can inject, or place, their own commands into inputs that we then run on our database.
- We might encounter a login page for a website that asks for a username and password, and checks for those in a SQL database.
- Our query for searching for a user might be:

```
rows = db.execute("SELECT * FROM users WHERE username = ? AND password =

if len(rows) == 1:
    # Log user in
```

  - By using the `?` symbols as placeholders, our SQL library will escape the input, or prevent dangerous characters from being interpreted as part of the command.
- In contrast, we might have a SQL query that's a formatted string, such as:

```
rows = db.execute(f"SELECT * FROM users WHERE username = '{username}' AN

if len(rows) == 1:
    # Log user in
```

- If a user types in `malan@harvard.edu'--` as their input, then the query will end up being:

```
rows = db.execute(f"SELECT * FROM users WHERE username = 'malan@harvarc
```

  - This query will actually select the row where `username = 'malan@harvard.edu'`, without checking the password, since the single quotes end the input, and `--` turns the rest of the line into a comment in SQL.
- The user could even add a semicolon, `;`, and write a new command of their own, that our database will execute.
- Another set of problems with databases are **race conditions**, where shared data is unintentionally changed by code running on different devices or servers at the same time.
- One example is a popular post getting lots of likes. A server might try to increment the

number of likes, asking the database for the current number of likes, adding one, and updating the value in the database:

```
rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);
likes = rows[0]["likes"]
db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);
```

    - Two different servers, responding to two different users, might get the same starting number of likes since the first line of code runs at the same time on each server.
    - Then, both will use `UPDATE` to set the *same* new number of likes, even though there should have been two separate increments.
- Another example might be of two roommates and a shared fridge in their dorm. The first roommate comes home, and sees that there is no milk in the fridge. So the first roommate leaves to the store to buy milk. While they are at the store, the second roommate comes home, sees that there is no milk, and leaves for another store to get milk as well. Later, there will be two jugs of milk in the fridge.
- We can solve this problem by locking the fridge so that our roommate can't check whether there is milk until we've gotten back.
- To solve this problem, SQL supports **transactions**, where we can lock rows in a database, such that a particular set of actions are **atomic**, or guaranteed to happen together.
- For example, we can fix our problem above with:

```
db.execute("BEGIN TRANSACTION")
rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);
likes = rows[0]["likes"]
db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);
db.execute("COMMIT")
```

    - The database will ensure that all the queries in between are executed together.
- But the more transactions we have, the slower our applications might be, since each server has to wait for other servers' transactions to finish.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 8

The internet

- Today we'll take a look at web programming, using a set of new languages and technologies to build applications that are both **server-side**, running on servers or cloud services, and **client-side**, running on the user's own devices.
- The **internet** is the network of networks of computers, or servers, communicating with one another by sending and receiving data.

    - The original “internet” was established in 1969, called ARPANET (https://en.wikipedia.org/wiki/ARPANET), which connected computers between various institutions.
    - Today, many more cables and server hardware connects all the computers on the internet.
- **Routers** are specialized computers, with CPUs and memory, that routes, or relays, data from one point to another. At home or on campus, for example, we might have routers that accepts data and sends them out.
    - We take a look at a video where staff members “send” an envelope across the screen of a Zoom meeting.
    - A router might have multiple options for what direction to send some data, and there are algorithms that try to figure out that direction.
- **Protocols** are a set of rules or conventions, like a physical handshake for humans , that the world has agreed upon for computers to communicate with.
- **TCP/IP** are two protocols for sending data between two computers. In the real world, we might write an address on an envelope in order to send a letter to someone, along with our own address for a letter in return.
- **IP** stands for internet protocol, a protocol that includes a standard way for computers to address each other. **IP addresses** are unique addresses for computers connected to the internet, such that a packet sent from one computer to another will be passed along routers until it reaches its destination.
    - An IP address might have the format `#.#.#.#`, where each number can have a value from 0 to 255. Each number will be the size of one byte, so the entire address will be 4 bytes, or 32 bits. This means that this version of IP, version 4, can only support a maximum of 4 billion addresses. Another version of IP, version 6, uses 128 bits to support many more possible addresses.
- **TCP**, transmission control protocol, is a protocol for sending and receiving data. TCP allows for a single server, at the same IP address, to provide multiple services through the use of a **port number**, a small integer added to the IP address. For example, HTTP is sent to port number 80, and HTTPS uses port number 443.
    - TCP also allows for a large amount of data, like an image, to be sent in smaller chunks. Each of them might be labeled with a sequence number, as with “part 1 of 4” or “part 2 of 4”. And if one of the parts is lost, the recipient can ask for the missing part again.
    - UDP is another protocol for sending data that does not guarantee delivery like TCP, which might be useful for streaming real-time videos or calls, since we don’t want to wait for all the packets to be redelivered before we get new ones.
- **DNS**, domain name system, is another technology that translates domain names like

cs50.harvard.edu to IP addresses. DNS is generally provided by a server nearby, with a big table in its memory, of domain names and IP addresses.

The web

- The internet, with routers, IP, TCP, and DNS, is like the plumbing that allows us to send data from one computer to another. The web is one application that is built on top of the internet.
- **HTTP**, or Hypertext Transfer Protocol, standardizes how web browsers and web servers communicate within TCP/IP packets.
  - **HTTPS** is the secure version of HTTP, ensuring that the contents of packets between the browser and server are encrypted.
- A **URL**, or web address, might look like `https://www.example.com/`.
  - `https://` is the protocol being used.
  - The `/` at the end is a request for the default file. It might also end in something like `/file.html` for a specific file.
  - `example.com` is the domain name. `.com` is a top-level domain name, and others like `.edu` or `.io` indicate what type of website might be hosted there. Today, there are hundreds of top-level domain names, some with restrictions on how they can be used.
  - `www` is the hostname, or subdomain, that refers to one or more specific servers in the domain name. A domain name might include web servers for `www`, or email servers for `mail`, so each subdomain can point to them separately.
  - Together, `www.example.com` is a **fully qualified domain name**, or one that has a specific set of addresses.
- Two commands supported by HTTP include **GET** and **POST**. GET allows a browser to ask for a page or file in a URL, and POST allows a browser to send additional data to the server that is hidden from the URL. Both of these are **requests** we can make to a server, which will provide a **response** in return.
- A GET request will start with:

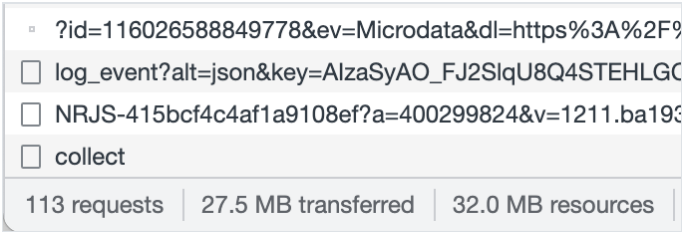
```
GET / HTTP/1.1
Host: www.example.com
...
```

  - The `GET` indicates that the request is for some file, and `/` indicates the default file.
  - There are different versions of the HTTP protocol, so `HTTP/1.1` indicates that the browser is using version 1.1.

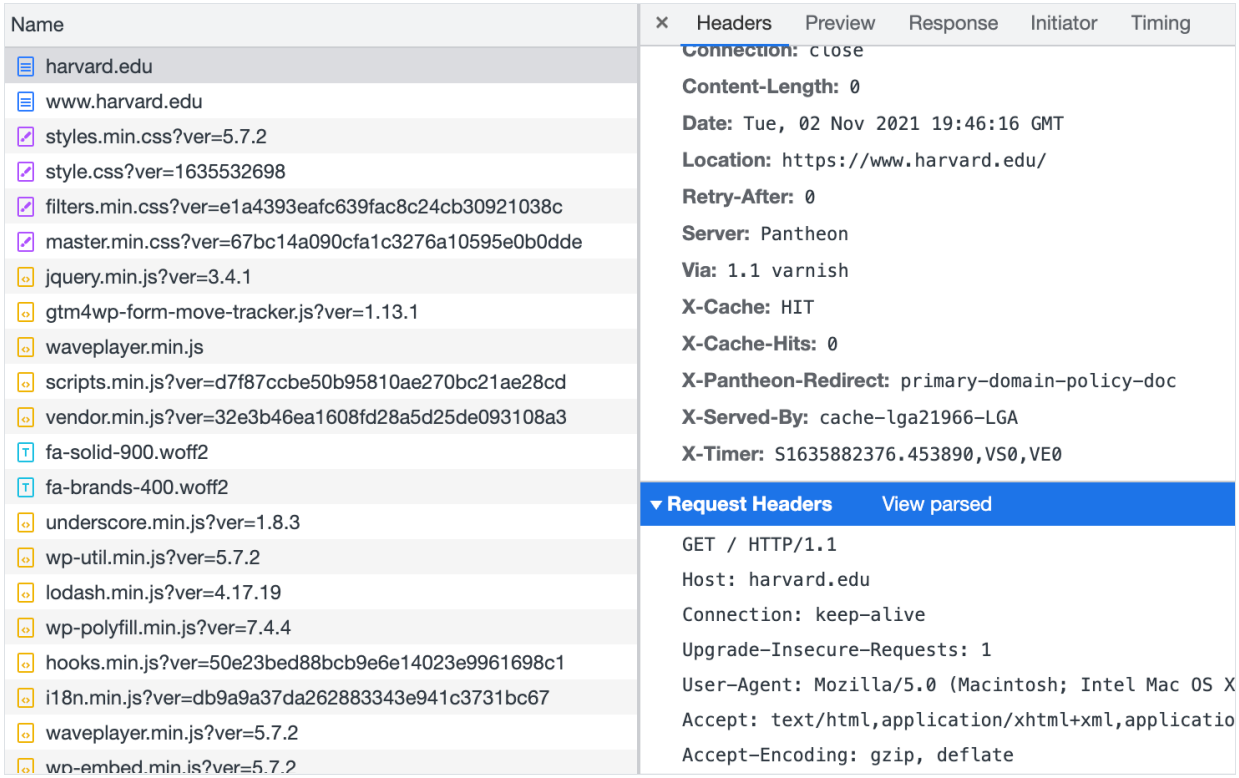
    - `Host: www.example.com` indicates that the request is for `www.example.com`, since the same web server might be hosting multiple websites and domains.
- A response for a successful request will start with:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

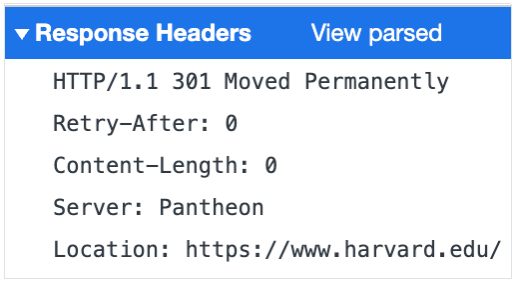
    - The web server will respond with the version of HTTP, followed by a status code, which is `200 OK` here, indicating that the request was valid.
    - Then, the web server indicates the type of content in its response, which might be text, image, or other format.
    - Finally, the rest of the packet or packets will include the content.
- The keys and values, like `Host: www.example.com` and `Content-Type: text/html`, are known as **HTTP headers**.
- We'll type in `http://harvard.edu` in our browser, and see that the address bar has changed to `https://www.harvard.edu` after the page has loaded. Browsers include developer tools, which allow us to see what's happening. In Chrome's menu, for example, we can go to View > Developer > Developer Tools, which will open a panel on the screen. We'll also use an Incognito window, so Chrome doesn't remember our previous requests.
- In the Network tab, we can see that there were over a hundred requests, for text, images, and other pieces of data that were downloaded separately for a single web page. It turns out that our browser made a single request, and the response from the server indicated that we needed to make all those other requests to download the other data on the page:



- If we scroll up in the lists of requests, we can see the request headers for the first request by clicking on the one for `harvard.edu`:



- And we can scroll to see that the server's response actually returned a status code of `301 Moved Permanently`, redirecting our browser from `http://...` to `https://...`:



  - Note that the response includes a `Location:` header for the browser to redirect us to.

- In VS Code's terminal, we can use a command-line tool, `curl`, to see the response headers for a request as well:

```
$ curl -I -X GET http://harvard.edu/
HTTP/1.1 301 Moved Permanently
Retry-After: 0
Content-Length: 0
Server: Pantheon
Location: https://www.harvard.edu/
...
```

- If we visit the new location with `curl`, we see a status code of `200`, as well as a new version of HTTP that we can use:

```
$ curl -I -X GET https://www.harvard.edu/
```

```
HTTP/2 200
cache-control: public, max-age=1200
content-type: text/html; charset=UTF-8
```

- And if we try to visit a URL that doesn't exist, we'll see an HTTP status code of `404`:

```
$ curl -I -X GET https://www.harvard.edu/thisfiledoesnotexist
HTTP/2 404
cache-control: no-cache, must-revalidate, max-age=0
content-type: text/html; charset=UTF-8
```

- Other **HTTP status codes** include:
  - `200 OK`
  - `301 Moved Permanently`
  - `302 Found`
  - `304 Not Modified`
  - `307 Temporary Redirect`
  - `401 Unauthorized`
  - `403 Forbidden`
  - `404 Not Found`
  - `418 I'm a Teapot`
    - An April Fool's joke years ago
  - `500 Internal Server Error`
    - Buggy code on a server might result in this status code, like segfaults we might have seen in C.
  - `503 Service Unavailable`
  - ...
- It turns out that `safetyschool.org` redirects to `yale.edu`! Someone must have purchased the domain name and set it to redirect:

```
$ curl -I -X GET http://safetyschool.org
HTTP/1.1 301 Moved Permanently
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 02 Nov 2021 19:59:18 GMT
Content-length: 122
Content-type: text/html
Location: http://www.yale.edu
Connection: close
```

- And `harvardsucks.org` used to be a website with a video of a prank on Harvard

(https://youtu.be/YuubOQFB9kk).

HTML

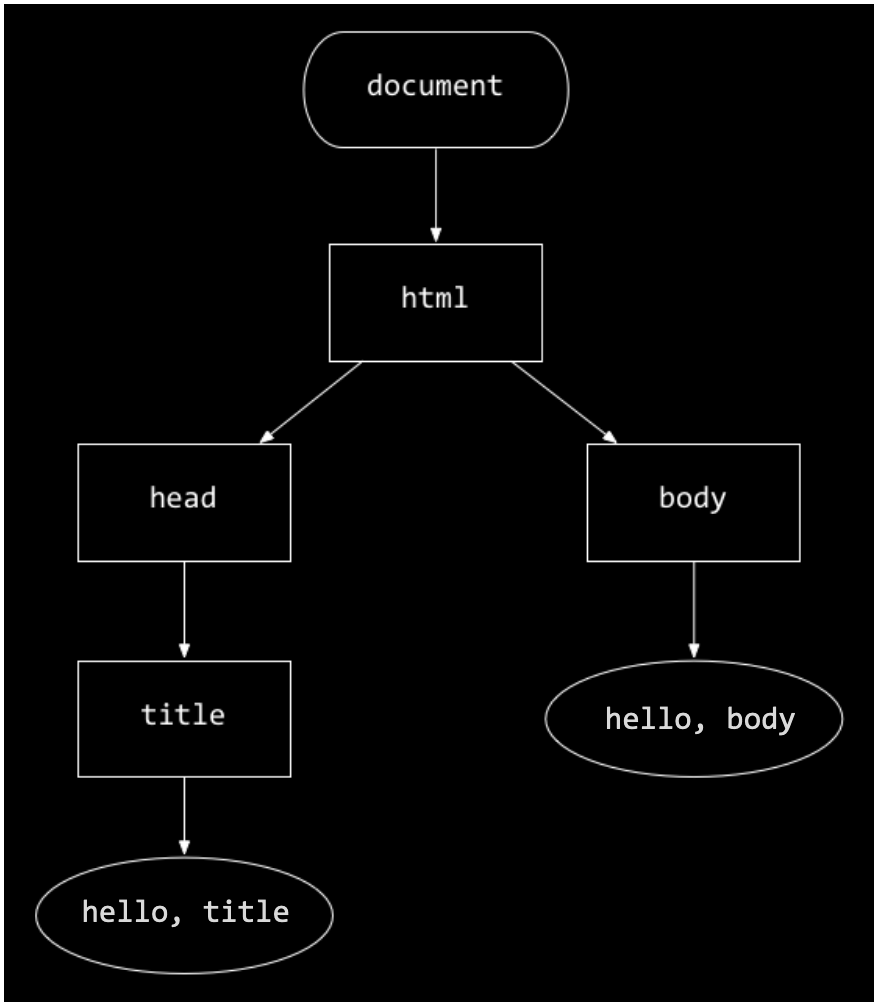
- Now that we can use the internet and HTTP to send and receive messages, it's time to see what's in the content for web pages. **HTML**, Hypertext Markup Language, is not a programming language, but rather used to format web pages and tell the browser how to display them.
- A simple page in HTML might look like this:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>
            hello, title
        </title>
    </head>
    <body>
        hello, body
    </body>
</html>
```

- Since this page is saved in our instance of VS Code, in the cloud, we can also run our own web server with the `http-server` command, and clicking "Open in Browser" in the notification that appears.
    - This web server will listen on port 8080 instead, since our instance of VS Code is using port 80 already.
- Then, we'll see the file we created, `hello.html`, and we can see our page's content, "hello, world", on the page, and title, "hello, title", in the tab bar.
- Let's look at the HTML again:
    - The first line, `<!DOCTYPE html>`, is a declaration that the page follows the HTML standard.
    - Next is a **tag**, a word in brackets like `<html>` and `</html>`. The first is a start or open tag, and the second is a close tag, which looks almost the same but with a `/` in front of the tag's name. In this case, the tags indicate the start and end of the HTML page. The start tag here has an **attribute** as well, `lang="en"` which specifies that the language of the page will be in English, to help the browser translate the page if needed. Notice that attributes are key-value pairs.
    - Nested within the `<html>` tag are two more tags, `<head>` and `<body>`, which are

both like children nodes in a tree. And within `<head>` is the `<title>` tag, the contents of which we see in a tab or window's title in a browser. Within `<body>` is the contents of the page itself, a text node, which we'll see in the main view of a browser as well.

- The page will be loaded into the browser's memory as a data structure, like this tree:



  - Note that there is a hierarchy mapping each tag and its children. Rectangular nodes are tags, while oval ones are text.
- HTML allows us to build the structure of our web pages, and we can look for reference materials online for all the tags and attributes that we can use as building blocks.
- We can use a validator (https://validator.w3.org/#validate_by_input) to check that our HTML is valid.
- We'll take a look at `paragraphs0.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/paragraphs0.html?highlight):

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>paragraphs</title>
```

```
    </head>
    <body>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Viv
        </p>
        <p>
            Mauris ut dui in eros semper hendrerit. Morbi vel elit mi. S
        </p>
        <p>
            Aenean venenatis convallis ante a rhoncus. Nullam in metus v
        </p>
    </body>
</html>
```

- With the `<p>` tag, we can indicate that each section of text should be a paragraph.
- After we save this file, we'll refresh the index of our web server, and then open `paragraphs.html`, to see that each paragraph of text is separated by some spacing.

- We can add headings with tags like `<h1>`, `<h2>`, and `<h3>` in `headings.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/headings.html?highlight):

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>headings</title>
    </head>

    <body>
        <h1>One</h1>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Viv
        </p>

        <h2>Two</h2>
        <p>
            Mauris ut dui in eros semper hendrerit. Morbi vel elit mi. S
        </p>

        <h3>Three</h3>
        <p>
            Aenean venenatis convallis ante a rhoncus. Nullam in metus v
        </p>
    </body>
```

```
</html>
```

    - Each level of heading has a different size, and we can use up to six levels of headings with `<h6>`.

- We take a look at `list0.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/list0.html?highlight), where we use the `<ul>` tag to create an unordered list, like bullet points:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>list</title>
    </head>
    <body>
        <ul>
            <li>foo</li>
            <li>bar</li>
            <li>baz</li>
        </ul>
    </body>
</html>
```

    - We can also use `<ol>` instead, for an ordered list with numbers.

- Tables start with a `<table>` tag and have `<tr>` tags as rows, and `<td>` tags for individual cells:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>table</title>
    </head>
    <body>
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Number</th>
                </tr>
            </thead>
            <tbody>
                <tr>
                    <td>Carter</td>
                    <td>+1-617-495-1000</td>
```

```
                </tr>
                <tr>
                    <td>David</td>
                    <td>+1-949-468-2750</td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

- In `image.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/image.html?highlight), we can upload an image to our instance of VS Code and include it in our page with an `<img>` tag. We can also use the `alt` attribute to add alternative text for accessibility:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>image</title>
    </head>
    <body>
        <img alt="Harvard University" src="harvard.jpg">
    </body>
</html>
```

    - It turns out the image is included at its full size, so we'll use CSS later to set its width and height.
- We can also include videos with `video.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/video.html?highlight):

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>video</title>
    </head>
    <body>
        <video autoplay loop muted width="1280">
            <source src="halloween.mp4" type="video/mp4">
        </video>
    </body>
</html>
```

    - We'll use HTML attributes to change how our video is displayed. Notice that some attributes are empty, where there is no value.

- We'll embed another page in ours with an inline frame, or iframe:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>iframe</title>
    </head>
    <body>
        <iframe allowfullscreen src="https://www.youtube.com/embed/xvFZ
    </body>
</html>
```

- We can create links in `link1.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/link1.html?highlight) with the `<a>`, or anchor, tag:

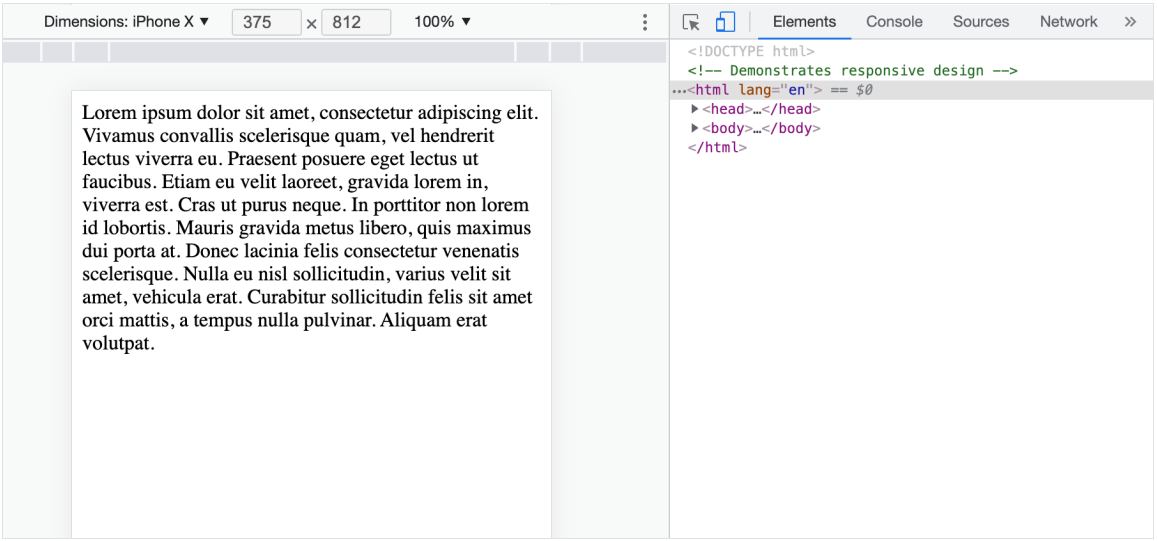
```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>link</title>
    </head>
    <body>
      Visit <a href="https://www.harvard.edu">Harvard</a>.
    </body>

</html>
```

    - The `href` attribute is for a hypertext reference, or simply where the link should take us, and within the tag is the text that should appear as the link.
    - When we visit this page, we can hover over the link, and our browser will show what the URL is.
    - But we could set the `href` to `https://www.yale.edu`, but leave `Harvard` within the tag, which might prank users or even trick them into visiting a fake version of some website. **Phishing** is an act of tricking users, a form of social engineering that includes misleading links.
    - We can link to other pages on our own server with just `image.html` or something similar.
- In `responsive.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/responsive.html?highlight), we can add attributes to make our page **responsive**, or automatically adapted for different screen sizes:

```
<!DOCTYPE html>
```

```
<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>responsive</title>
    </head>
    <body>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
    </body>
</html>
```

- We'll open Chrome's Developer Tools again, and in the top left of the panel, use the icon that looks like mobile devices to simulate a phone:



- It turns out that we can also provide inputs in a request as part of a URL like `https://wwww.example.com/path?key=value`. Here, the `?` indicates that we're adding inputs, which will include one or more key-value pairs.
- If we search for something on Google, we'll see that the URL changes to `https://www.google.com/search?q=cats&...`. Here, the `q` key, for "query", has a value of `cats`, along with other keys and values.
- These inputs are part of GET requests that look like:

```
GET /search?q=cats HTTP/1.1
Host: www.google.com
...
```

  - We can also use POST, to send inputs like usernames and passwords, that should be hidden from the URL.
- In `search0.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/search0.html?highlight), we can create a form that takes user input and sends it to Google's search engine:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>search</title>
    </head>
    <body>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

- First, we have a `<form>` tag that has an `action` of Google's search URL, with a method of GET.
- Inside the form, we have one `<input>`, with the name `q`, and another `<input>` with the type of `submit`. When the second input, a button, is clicked, the form will automatically add the input to the URL.
- So when we open `search.html` in our browser, we can use the form to search via Google.

CSS

- Let's make a home page:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>home</title>
    </head>
    <body>
        <p>
            John Harvard
        </p>
        <p>
            Welcome to my home page!
        </p>
        <p>
            Copyright (c) John Harvard
        </p>
    </body>
```

```
</html>
```

    - We have three paragraphs, and we could use `<div>` tags, or divisions, to indicate they are separate areas on our page.

- We can also use HTML tags that add more context to our page:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>home</title>
    </head>
    <body>
        <header>
            John Harvard
        </header>
        <main>
            Welcome to my home page!
        </main>
        <footer>
            Copyright (c) John Harvard
        </footer>
    </body>
</html>
```

- We'll stylize our page by adding a few aesthetics:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>home</title>
    </head>
    <body>
        <header style="font-size: large; text-align: center;">
            John Harvard
        </header>
        <main style="font-size: medium; text-align: center;">
            Welcome to my home page!
        </main>
        <footer style="font-size: small; text-align: center;">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

    - We'll also use an **HTML entity** to represent the copyright symbol, which will be displayed in our browser as ©.
- In our `<style>` tags, we're using **CSS**, Cascading Style Sheets, another language that tells our browser how to display tags on a page. CSS uses **properties**, or key-value pairs, like `font-size: large;`.
- We can align all the text at once, instead of repeating ourselves:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>home</title>
    </head>
    <body style="text-align: center;">
        <header style="font-size: large;">
            John Harvard
        </header>
        <main style="font-size: medium;">
            Welcome to my home page!
        </main>
        <footer style="font-size: small;">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

    - Here, the style applied to the `<body>` tag cascades, or applies, to its children, so all the sections inside will have centered text as well.
- To factor out, or separate our CSS from HTML, we can include styles in the `<head>` tag:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            body {
                text-align: center;
            }

            header
            {
                font-size: large;
            }
```

```
            main
            {
                font-size: medium;
            }

            footer
            {
                font-size: small;
            }

        </style>
        <title>home</title>
    </head>
    <body>
        <header>
            John Harvard
        </header>
        <main>
            Welcome to my home page!
        </main>
        <footer>
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

    - We can use a CSS **type selector** to style each type of tag.
- We can also use a more specific **class selector**:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            .centered
            {
                text-align: center;
            }

            .large
            {
                font-size: large;
            }
```

```
            .medium
            {
                font-size: medium;
            }

            .small
            {
                font-size: small;
            }

        </style>
        <title>css</title>
    </head>
    <body>
        <header class="centered large">
            John Harvard
        </header>
        <main class="centered medium">
            Welcome to my home page!
        </main>
        <footer class="centered small">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- We can define our own CSS **class** with a `.` followed by a keyword we choose, so here we've created `.centered`, `.large`, `.medium`, and `.small`, each with some property.
- Then, on any number of tags in our page's HTML, we can add one or more of these classes with the `class` attribute.

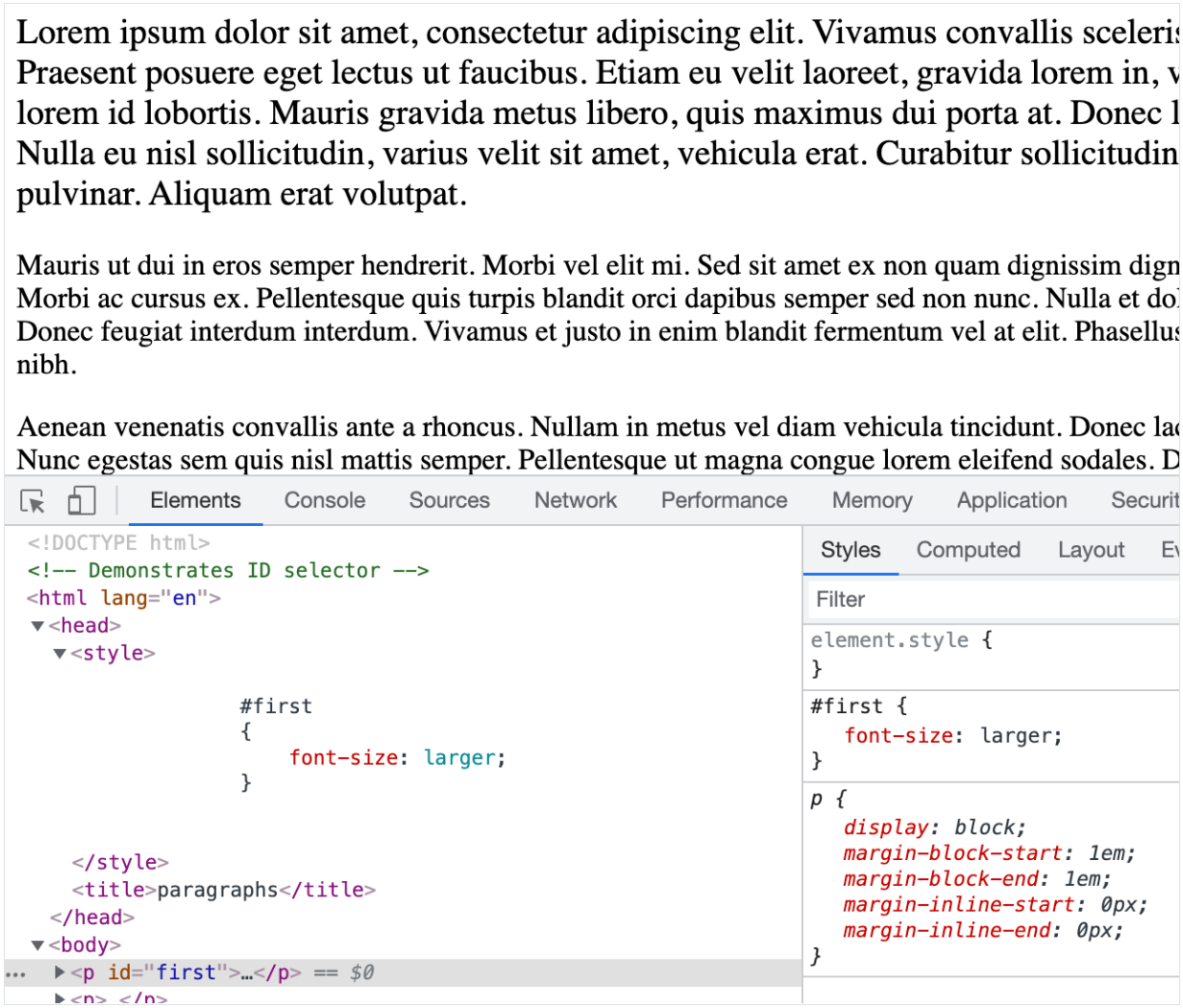
- Finally, we can take all of the CSS for the properties and move them to another file with the `<link>` tag:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <link href="home.css" rel="stylesheet">
        <title>home</title>
    </head>
    <body>
        <header class="centered large">
```

```
            John Harvard
        </header>
        <main class="centered medium">
            Welcome to my home page!
        </main>
        <footer class="centered small">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

```
.centered
{
    text-align: center;
}

.large
{
    font-size: large;
}

.medium
{
    font-size: medium;
}

.small
{
    font-size: small;
}
```

    - Now, we have a reusable CSS file.
- CSS also has **ID selectors**, like in `paragraphs1.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/paragraphs1.html). It turns out that we can use Chrome's Developer Tools here as well. We'll use the Elements tab to see that the `<head>` of this page includes properties for `#first`, an ID in CSS that we can use only once, as well as a HTML tag `<p id="first">` that has the styles applied:



- We can click on an element in the HTML in this panel, and change the style of our page within our browser. We can hover over CSS properties on the right side, and uncheck or change them. This won't change our original source code, but this will change our browser's copy so we can experiment.
- We can also right-click on anything displayed on the page, and click "Inspect Element" to see it highlighted in the panel for us, where we can make more changes quickly or learn how other pages implement features.

- With CSS, we'll also rely on references and other resources to look up how to use properties as we need them.
- We can use other types of selectors as well:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            p:first-child
```

```
            {
                font-size: larger;
            }

        </style>
        <title>paragraphs</title>
    </head>
    <body>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Viv
        </p>
        <p>
            Mauris ut dui in eros semper hendrerit. Morbi vel elit mi. S
        </p>
        <p>
            Aenean venenatis convallis ante a rhoncus. Nullam in metus v
        </p>
    </body>
</html>
```

- Here, we're using `p:first-child` to set properties on the first `<p>` tag.

- We'll look at the style of `link2.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/link2.html):

```
a {
    color: #ff0000;
    text-decoration: none;
}

a:hover {
    text-decoration: underline;
}
```

- In `link4.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/link4.html), we can select tags based on attributes:

```
a
{
    text-decoration: none;
}

a:hover
{
    text-decoration: underline;
}
```

```
a[href="https://www.harvard.edu/"]
{
    color: #ff0000;
}

a[href="https://www.yale.edu/"]
{
    color: #0000ff;
}
```

    - The **attribute selectors** will affect tags with those attributes, and we can use `a[href*="harvard.edu"]` to be less specific in our selection, affecting tags with `harvard.edu` anywhere in its `href`.
- A set of CSS conventions and shared styles is known as a **framework**, with classes and components we can quickly use.
- One popular framework is Bootstrap (https://getbootstrap.com/), with components like alerts that we can use with HTML like:

```
<div class="alert alert-warning">
    ...
</div>
```

    - The framework provides the CSS that sets the style for those classes.
- With the help of the documentation on Bootstrap's website, we'll include a `<link>` to its CSS for our page with a table:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/cs
        <title>table</title>
    </head>
    <body>
        <table class="table">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Number</th>
                </tr>
            </thead>
            <tbody>
                <tr>
```

```
                    <td>Carter</td>
                    <td>+1-617-495-1000</td>
                </tr>
                <tr>
                    <td>David</td>
                    <td>+1-949-468-2750</td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

- By adding the `table` class, per the Boostrap documentation, we see that our table is indeed stylized to be easier to read.

- We'll update our search page, too, with styles from Bootstrap:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/cs
    <title>search</title>
    </head>
    <body>
        <div class="container-fluid">

            <ul class="m-3 nav">
                <li class="nav-item">
                    <a class="nav-link text-dark" href="https://about.go
                </li>
                <li class="nav-item">
                    <a class="nav-link text-dark" href="https://store.go
                </li>
                <li class="nav-item ms-auto">
                    <a class="nav-link text-dark" href="https://www.goog
                </li>
                <li class="nav-item">
                    <a class="nav-link text-dark" href="https://www.goog
                </li>
                <li class="nav-item">
                    <a class="btn btn-primary" href="https://accounts.go
                </li>
            </ul>

            <div class="text-center">
```

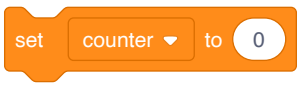
```
                <img alt="Happy Cat" class="img-fluid w-25" src="cat.gif">
                <form action="https://www.google.com/search" class="mt-4"
                    <input autocomplete="off" autofocus class="form-contro
                    <button class="btn btn-light" type="submit">Google Sea
                    <button class="btn btn-light" name="btnI" type="submit
                </form>

            </div>
        </div>

    </body>
</html>
```

- First, we'll put everything in a `<div>` that can grow to fit the screen.
- Then, we'll create a list with items and classes based on Bootstrap's documentation, to display links and buttons in the header.
- Finally, we'll add an image of a cat to the center of our page, as well as styles for our form.

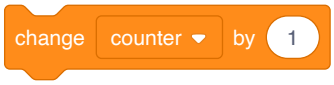
- Even with a framework, we can still write our own CSS styles to change any that we want.

JavaScript

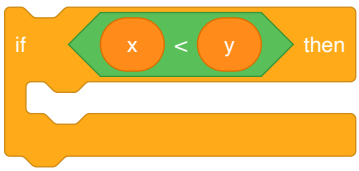
- To write code that can run in users' browsers, or on the client, we'll use a new language, **JavaScript**. The code will still come from our web server, but it will be executed by the user's browser.
- The syntax of JavaScript is similar to that of C and Python for basic constructs:



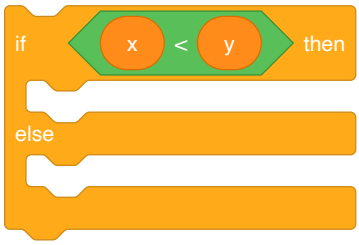

```
let counter = 0;
```



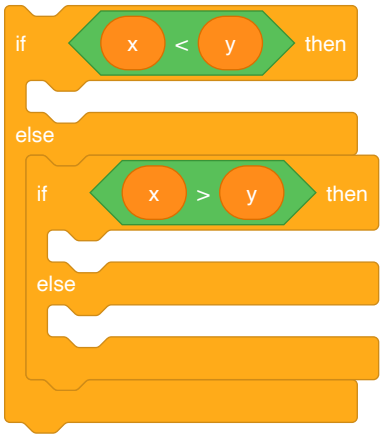

```
counter = counter + 1;
counter += 1;
counter++;
```



```
if (x < y)
{

}
```




```
if (x < y)
{

}
else
{

}
```




```
if (x < y)
{

}
else if (x > y)
{

}
else
{

}
```


```
while (true)
{

}
```


```
for (let i = 0; i < 3; i++)
{

}
```

- Notice that in JavaScript we use `let` to declare variables, without needing to indicate types.

- With JavaScript, we can change the HTML in the browser in real-time. We can use `<script>` tags to include our code directly, or from a `.js` file.
- We'll create another form:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

            function greet()
            {
                alert('hello, there');
            }

        </script>
        <title>hello</title>
    </head>
    <body>
        <form onsubmit="greet(); return false;">
            <input autocomplete="off" autofocus id="name" placeholder="N
            <input type="submit">
        </form>
    </body>
```

```
</html>
```

- Here, we won't add an `action` to our form, since this will stay on the same page. Instead, we'll have an `onsubmit` attribute that will call a function we've defined in JavaScript, and use `return false;` to prevent the form from actually being submitted anywhere.
- In the `<head>` tag, we'll have a `<script>` tag with a function that defines a function, `greet`, in JavaScript.
- Now, if we load that page, we'll see `hello, there` being shown when we submit the form.

- Since our input tag, or **element**, has an ID of `name`, we can use it in our code:

```
<script>

    function greet()
    {
        let name = document.querySelector('#name').value;
        alert('hello, ' + name);
    }

</script>
```

- `document` is a global variable that comes with JavaScript in the browser, and `querySelector` is a function we can use to select a node in the **DOM**, Document Object Model, or the tree structure of the HTML page. After we select the element with the ID `name`, we get the text `value` inside the input, and add it to our alert.

- We can move our function to the bottom of the `<body>` of the page, since we want the rest of the page to load first:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        <form>
            <input autocomplete="off" autofocus id="name" placeholder="N
            <input type="submit">
        </form>
        <script>

            function greet()
```

```
            {
                let name = document.querySelector('#name').value;
                alert('hello, ' + name);
            }

            document.querySelector('form').addEventListener('submit', gr

      </script>
    </body>
</html>
```

- Now, we can listen to **events** in JavaScript, which occur when something happens on the page. For example, we can listen to the `submit` event on our `form` element, and call the `greet` function when the event happens.

- We can also use **anonymous functions** in JavaScript:

```
<script>

    document.querySelector('form').addEventListener('submit', function(e
        let name = document.querySelector('#name').value;
        alert('hello, ' + name);
        e.preventDefault();
    });

</script>
```

- We can pass in a function with no name with the `function()` syntax, and it turns out that event handlers in JavaScript get an event variable, `e` by convention, that we can use inside our function. Here, we use `e.preventDefault();` to stop the default behavior of the form.

- We can programmatically change style, too:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>background</title>
    </head>
    <body>
        <button id="red">R</button>
        <button id="green">G</button>
        <button id="blue">B</button>
        <script>
```

```
            let body = document.querySelector('body');
            document.querySelector('#red').addEventListener('click', fur
            {
                body.style.backgroundColor = 'red';
            });

            document.querySelector('#green').addEventListener('click', f
                body.style.backgroundColor = 'green';
            });

            document.querySelector('#blue').addEventListener('click', fu
                body.style.backgroundColor = 'blue';
            });

        </script>
    </body>
</html>
```

- After selecting an element, we can use the `style` property to set values for CSS properties as well. Here, we have three buttons, each of which has an event listener for the `click` event, that changes the background color of the `<body>` element.

- We can also use JavaScript to make an element "blink", or appear and reappear at an interval:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

            // Toggles visibility of greeting
            function blink()
            {
                let body = document.querySelector('body');
                if (body.style.visibility == 'hidden')
                {
                    body.style.visibility = 'visible';
                }
                else
                {
                    body.style.visibility = 'hidden';
                }
            }

            // Blink every 500ms
```

```
            window.setInterval(blink, 500);

        </script>
        <title>blink</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- We can implement a form with autocomplete, using a dictionary of words and an event listener for the `keyup` event:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <title>autocomplete</title>
    </head>

    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="te

        <ul></ul>

        <script src="large.js"></script>
        <script>

            let input = document.querySelector('input');
            input.addEventListener('keyup', function(event) {
                let html = '';
                if (input.value) {
                    for (word of WORDS) {
                        if (word.startsWith(input.value)) {
                            html += `<li>${word}</li>`;
                        }
                    }
                }
                document.querySelector('ul').innerHTML = html;
            });

        </script>
```

```
    </body>
</html>
```

    - If we visit `autocomplete.html` (https://cdn.cs50.net/2021/fall/lectures/8/src8/autocomplete.html) and start typing in the input box, we'll see matching words appear below.

- With `geolocation.html` (https://cdn.cs50.net/2020/fall/lectures/8/src8/geolocation.html), we can ask the browser for a user's GPS coordinates:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>geolocation</title>
    </head>
    <body>
        <script>

            navigator.geolocation.getCurrentPosition(function(position)
                document.write(position.coords.latitude + ", " + positic
            });

        </script>
    </body>
</html>
```

    - Now, we can use those coordinates to see our location on a map.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 9

- Web programming
- Flask
- Forms
- Layouts
- POST
- MVC
- Frosh IMs
- Storing data
- Emails
- Sessions
- store, shows
- Searching

Web programming

- Today, we'll use all the tools and technologies we've learned to build applications for the web.
- Last week, we used `http-server` as a web server in VS Code. This program listens for connections and requests, and responds with static content, like HTML files and images. But `http-server` can't process other types of requests, like form inputs from the user.
- Recall that a URL might look like `http://www.example.com/`, for the site's default page, or `http://www.example.com/file.html` for a specific file. A file might be in a folder, like `folder/file.html`, and that reference is known as a **path**. A path can also be called a **route**, which does not need to refer to an actual file.
- A URL can also include form inputs, like:

  ```
  http://www.example.com/route?key=value
  ```

- An HTTP request for a route with inputs might look like:

  ```
  GET /search?q=cats HTTP/1.1
  Host: www.google.com
  ...
  ```

  - Now, we need a web server that can **parse**, or analyze, HTTP request headers and return different pages based on the route.

Flask

- We'll use Python and a library called **Flask** to write our own web server.
- Flask is also a **framework**, where the library of code also comes with a set of conventions for how it should be used. For example, like other libraries, Flask includes functions we can use to parse requests individually, but as a framework, also requires our program's code to be organized in a certain way:

  ```
  app.py
  requirements.txt
  static/
  templates/
  ```

  - `app.py` will have the Python code for our web server.
  - `requirements.py` includes a list of required libraries for our application.
  - `static/` is a directory of static files, like images and CSS and JavaScript files.

    - `templates/` is a directory for HTML files that will form our pages.
- Other web server frameworks will have a different set of conventions and requirements.
- Let's write a simple web server by creating an `app.py` in VS Code:

```
from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")
```

    - First, we'll import `Flask` from the `flask` library, along with some other functions.
    - Then, we'll create an `app` variable by giving our Python file's name to the `Flask` variable.
    - Next, we'll label a function for the `/` route with `@app.route` from Flask. The `@` symbol in Python is called a **decorator**, which modifies a function.
    - Finally, our `index()` function will just render a template, or return HTML code, from the file `index.html`.
- Now, we'll need to create a `templates/` directory, and create an `index.html` file with some content inside it:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>hello</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- Now, typing `flask run` will return that HTML file when we visit our server's URL: ``` $ flask run
    - Environment: development
    - Debug mode: off
    - Running on https://student50-code50-5000.githubpreview.dev/ (Press CTRL+C to quit)

    - Restarting with stat ```
- We can change the URL by adding `/?name=David`, but our page stays the same. We'll need to change our code in `app.py`:

```
from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    name = request.args.get("name")
    return render_template("index.html", name=name)
```

    - We can use the `request` variable from the Flask library to get the arguments from the request. Then, we can pass in the `name` variable as an argument to the `render_template` function.
- In our HTML file, we can include that variable with two curly braces:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>hello</title>
    </head>
    <body>
        hello, {{ name }}
    </body>
</html>
```

- Now, if we visit our URL with our input, we'll see that the page's content now includes it.
- To make sure our templates are reloaded, we can press `control` and `C` in the terminal window to exit Flask, and restart our server with `flask run` again.

Forms

- In `index.html`, we'll create a form:

```
<!DOCTYPE html>

<html lang="en">
    <head>
```

```
        <meta name="viewport" content="initial-scale=1, width=device-wid
        <title>hello</title>
    </head>
    <body>
        <form action="/greet" method="get">
            <input autocomplete="off" autofocus name="name" placeholder=
            <input type="submit">
        </form>
    </body>
</html>
```

- We'll send the form to the `/greet` route, and have an input for the `name` parameter as well as a submit button.

- Now, when we refresh our page and submit the form, we see that the URL has changed to something like `/greet?name=David`, and a page that says "Not Found".
- In `app.py`, we'll need to add a function for the `/greet` route with what we had in the `index()` function before:

```
@app.route("/")
def index():
    return render_template("index.html")


@app.route("/greet")
def greet():
    name = request.args.get("name")
    return render_template("greet.html", name=name)
```

- Meanwhile, our `index()` function will just return the `index.html` page with the form.

- When we submit our form this time, we see an "Internal Server Error" on the page. We can go back to our terminal window where `flask run` will show more details:

```
[2021-11-10 00:17:18,559] ERROR in app: Exception on /greet [GET]
Traceback (most recent call last):
  ...
  File "/usr/local/lib/python3.10/site-packages/flask/templating.py", li
    raise TemplateNotFound(template)
jinja2.exceptions.TemplateNotFound: greet.html
127.0.0.1 - - [8/Nov/2021 19:05:28] "GET /greet?name=David HTTP/1.0" 500
```

- We see that the request was indeed for the correct route, but it appears that Flask couldn't find the template called `greet.html`.

- We'll have to create a new file in `templates` called `greet.html` and use the `name`

variable as before:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>hello</title>
    </head>
    <body>
        hello, {{ name }}
    </body>
</html>
```

- Now, if we restart our server, we can see our form at the default page, and use the input from the user to generate another page from a template.
- It turns out that the `get` function allows for a default value, so we can write:

```
@app.route("/greet")
def greet():
    name = request.args.get("name", "world")
    return render_template("greet.html", name=name)
```

  - Now, if someone visits our URL with just `/greet`, the `name` variable will be set to `world` by default.
- But, if someone uses our form and provides no input, the URL becomes `/greet?name=`, and `name` will actually have a value of a blank string. We can add the `required` attribute to our form in `index.html`, but we can't rely on that safety check, since, as we saw last week, anyone can change our page on the client-side with Developer Tools in the browser.

Layouts

- In `index.html` and `greet.html`, we have some repeated HTML code. With just HTML, we aren't able to share code between files, but with Flask templates (and other web frameworks), we can factor out such common content.
- We'll create another template, `layout.html`:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
```

```
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

- With the `{% %}` syntax, we can include placeholder blocks, or other chunks of code. Here we've named our block `body` since it contains the HTML that should go in the `<body>` element.

- In `index.html`, we'll use the `layout.html` blueprint and only define the `body` block with:

```
{% extends "layout.html" %}

{% block body %}

    <form action="/greet" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Nam
        <input type="submit">
    </form>

{% endblock %}
```

  - Now, we can have just our `<form>` element in `index.html`. The template code, in `{% %}`, indicates to Flask that we want `index.html` to use another template, `layout.html`, and substitute this code in the `{% block body %}` placeholder.

- Similarly, in `greet.html`, we define the `body` block with just the greeting:

```
{% extends "layout.html" %}

{% block body %}

    hello, {{ name }}

{% endblock %}
```

  - The templating language is actually called Jinja, and understood by Flask.

- Now, if we restart our server, and view the source of our HTML after opening our server's URL, we see a complete page with our form inside our HTML file, generated by Flask:

```
<!DOCTYPE html>

<html lang="en">
    <head>
```

```
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>hello</title>
    </head>
    <body>


    <form action="/greet" method="get">
        <input autocomplete="off" autofocus name="name" placeholder="Na
        <input type="submit">
    </form>


    </body>
</html>
```

- Our indentation is no longer perfect, but that's acceptable since now our source code has the proper indentation, and Flask is using that to generate the page.

- Since our HTML pages are now generated by logic in code, we've built a **web application**.

POST

- Our form above used the GET method, which includes our form's data in the URL.
- We'll just need to change the `method` in our HTML form: `<form action="/greet" method="post">`.
- Now, when we visit our form and submit it, we see another error, "Method Not Allowed".
- Our controller will also need to be changed to accept the POST method, and look for the input from the form:

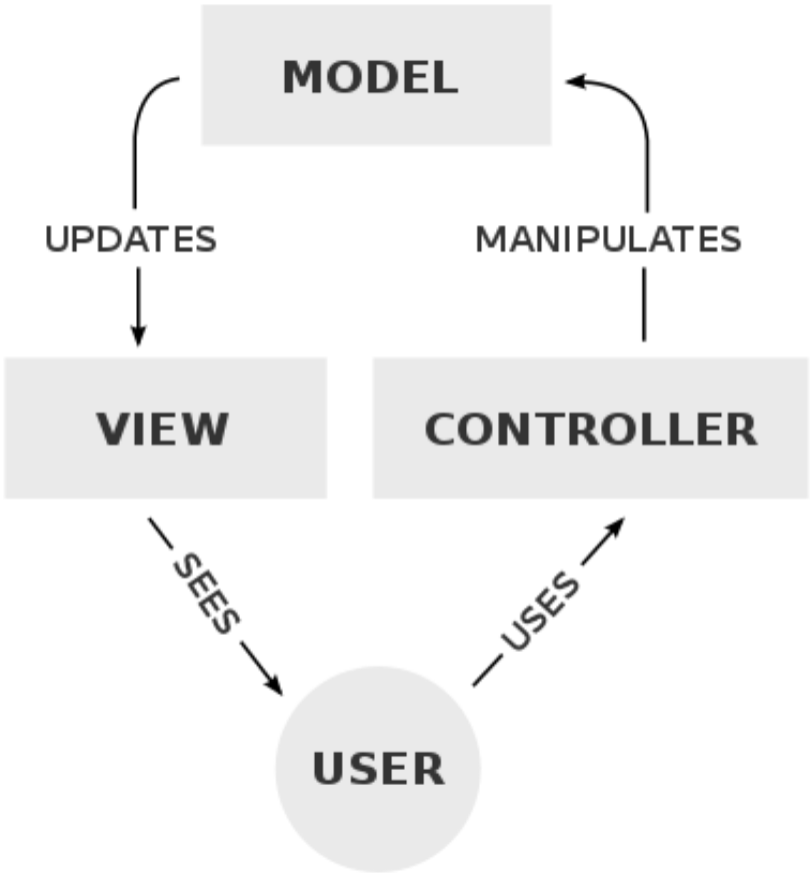
```
@app.route("/greet", methods=["POST"])
def greet():
    return render_template("greet.html", name=request.form.get("name", "
```

  - While `request.args` is for inputs from a GET request, we have to use `request.form` in Flask for inputs from a POST request.

- Now, when we restart our application after making these changes, we can see that the form takes us to `/greet`, but the contents aren't included in the URL anymore.
- Note that when we reload the `/greet` page, the browser asks us to confirm the form submission, since it's temporarily remembering the inputs.
- GET requests are useful since they allow the browser to save the contents of the form in history, and allow links that include information as well, like https://www.google.com/search?q=what+time+is+it (https://www.google.com/search?q=what+time+is+it).

MVC

- The Flask framework implements a particular **paradigm**, or way of thinking and programming. This paradigm, also implemented by other frameworks, is known as **MVC**, or Model–view–controller:



  - The controller contains our "business logic", code that manages our application overall, given user input. In Flask, this will be our Python code in `app.py`.
  - The view includes templates and visuals for the user interface, like the HTML and CSS that the user will see and interact with.
  - The model is our application's data, such as a SQL database or CSV file, which we haven't yet used.

Frosh IMs

- One of David's first web applications was for students on campus to register for "frosh IMs", intramural sports:

- We'll use a `layout.html` similar to what we had before:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wid
        <title>froshims</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

- In our `app.py`, we'll return our `index.html` template for the default `/` route:

```
from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")
```

- Our `index.html` template will look like this:

```
{% extends "layout.html" %}

{% block body %}

    TODO

{% endblock %}
```

- We'll run our server and load the page just to see that everything is working so far.
- Let's add a form to the `index.html` template:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Nam
        <select>
            <option disabled selected>Sport</option>
            <option value="Basketball">Basketball</option>
            <option value="Soccer">Soccer</option>
            <option value="Ultimate Frisbee">Ultimate Frisbee</option>
        </select>
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

  - We'll plan to have a `/register` route, and have a `<select>` menu, which looks like a dropdown menu with options for each sport.
- In `app.py`, we'll allow POST for our `/register` route:

```
@app.route("/register", methods=["POST"])
def register():

  # Validate submission
  if not request.form.get("name") or request.form.get("sport") not in ['
      return render_template("failure.html")

  # Confirm registration
  return render_template("success.html")
```

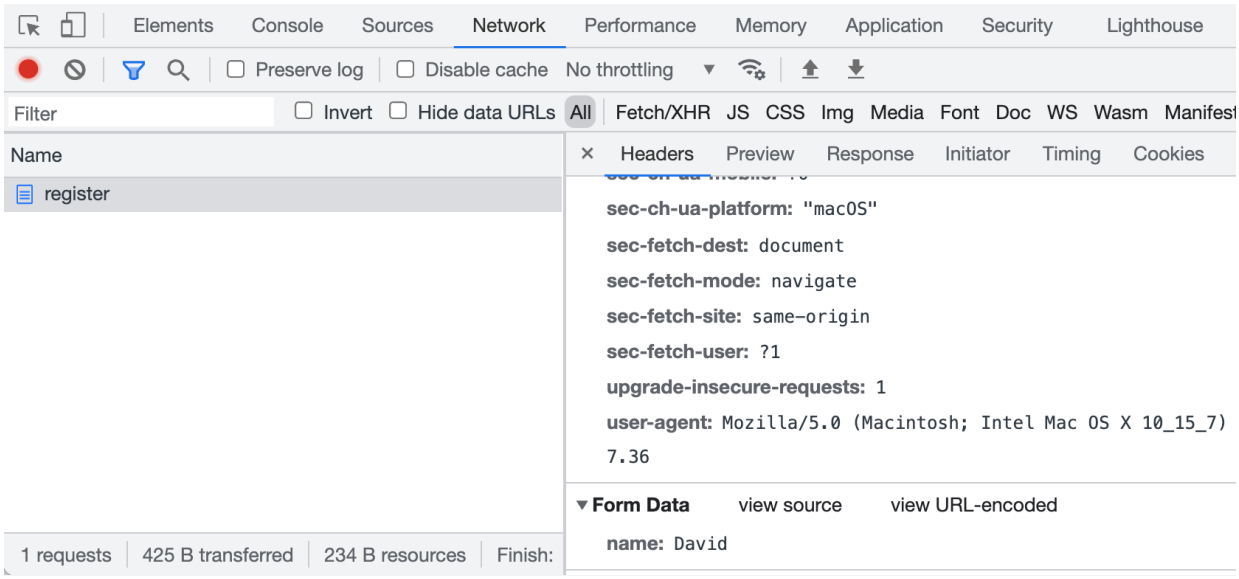
  - We'll check that our form's values are valid, and then return a template for either failure or success depending on the results.
- `failure.html` will have:

```
{% extends "layout.html" %}

{% block body %}
    You are not registered!
{% endblock %}
```

- And `success.html` will have:

```
{% extends "layout.html" %}

{% block body %}
    You are registered!
{% endblock %}
```

- Notice that our application doesn't actually save the data anywhere yet.
- When we run this application, though, we see that we are not registered, even if we fill out the form.
- We can open the Developer Tools in Chrome, using the Network tab like we did last week, and submit our form again to see the request our browser actually made:



  - At the very bottom, we see that the form data only has `name: David`.

- It turns out that our `<select>` input needs a name of `sport` to be sent to the server:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Nam
        <select name="sport">
            <option disabled selected>Sport</option>
```

```
            <option value="Basketball">Basketball</option>
            <option value="Soccer">Soccer</option>
            <option value="Ultimate Frisbee">Ultimate Frisbee</option>
        </select>
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

- Now our server will be able to get the input with `request.form.get("sport")`.

- We can also link to CSS in `layout.html`:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <link href="/static/styles.css" rel="stylesheet">
        <title>froshims</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

- By Flask's convention, we'll have a directory called `static` for files like images and CSS.

- We can improve the design of our application by having a single list of sports:

```
from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Basketball"
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)

...
```

    - Then, we'll pass that list into the `index.html` template.
- In our template, we can use a loop to generate a list of options from the list of strings passed in as `sports`:

```
...
<select name="sport">
    <option disabled selected value="">Sport</option>
    {% for sport in sports %}
        <option value="{{ sport }}">{{ sport }}</option>
    {% endfor %}
</select>
...
```

    - The `for` and `endfor` syntax creates a loop, and now we can programmatically create an `<option>` element for each sport in our list.
- Finally, we can check that the `sport` sent in the POST request is in the list `SPORTS` in `app.py`:

```
...
@app.route("/register", methods=["POST"])
def register():

    if not request.form.get("name") or request.form.get("sport") not in
        return render_template("failure.html")

    return render_template("success.html")
```

    - Now, to add another sport, we only need to change the `SPORTS` list.
- We can change the select menu in our form to be checkboxes, to allow for multiple sports:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Register</h1>

    <form action="/register" method="post">

        <input autocomplete="off" autofocus name="name" placeholder="Nam
        {% for sport in sports %}
            <input name="sport" type="checkbox" value="{{ sport }}"> {{
        {% endfor %}
        <input type="submit" value="Register">
```

```
    </form>
{% endblock %}
```

- We'll make sure that each `input` has the value of the sport, so it can be sent to the server, and also the sport printed next to it, so the user can see it.
- Back in our `register` function in `app.py`, we can use another function to get the list of checked options.

- We can also use radio buttons, which will allow only one option to be chosen at a time with `<input name="sport" type="radio" value="{{ sport }}"> {{ sport }}`.

Storing data

- Let's look at how we might store our registered students in a dictionary in the memory of our web server with `froshims3` (https://cdn.cs50.net/2021/fall/lectures/9/src9/froshims3/):

```
# Implements a registration form, storing registrants in a dictionary, v

from flask import Flask, redirect, render_template, request

app = Flask(__name__)

REGISTRANTS = {}

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/register", methods=["POST"])
def register():

    # Validate name
    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")
```

```
    # Validate sport
    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    # Remember registrant
    REGISTRANTS[name] = sport

    # Confirm registration
    return redirect("/registrants")


@app.route("/registrants")
def registrants():
    return render_template("registrants.html", registrants=REGISTRANTS)
```

  - We'll create a dictionary called `REGISTRANTS`, and in `register` we'll first check the `name` and `sport`, returning a different error message in each case with `error.html`. Then, we can store the name and sport in our `REGISTRANTS` dictionary, and redirect to another route that will display registered students.

- The error message template, meanwhile, will display the error message along with a fun image of a grumpy cat:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Error</h1>
    <p>{{ message }}</p>
    <img alt="Grumpy Cat" src="/static/cat.jpg">
{% endblock %}
```

- Our `registrants.html` template will print a table with the dictionary passed in as input:

```
  {% extends "layout.html" %}

  {% block body %}
      <h1>Registrants</h1>
      <table>
          <thead>
              <tr>
                  <th>Name</th>
```

```
                <th>Sport</th>
            </tr>
        </thead>
        <tbody>
            {% for name in registrants %}
                <tr>
                    <td>{{ name }}</td>
                    <td>{{ registrants[name] }}</td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

- Our table has a header row, and then a row for each key and value stored in `registrants`.

- If our web server stops running, we'll lose the data stored in memory, so we'll use a SQLite database with the SQL library from `cs50` in `froshims4` (https://cdn.cs50.net/2021/fall/lectures/9/src9/froshims4/):

```
# Implements a registration form, storing registrants in a SQLite databa

from cs50 import SQL
from flask import Flask, redirect, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///froshims.db")

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)

...
```

- In our terminal, we can run `sqlite3 froshims.db` to open the database, and use the `.schema` command to see the table with columns of `id`, `name`, and `sport`, which was created in advance:

```
src9/froshims4/ $ sqlite3 froshims.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE registrants (id INTEGER, name TEXT NOT NULL, sport TEX
```

- Now, in our `/register` route, we're using SQL to store our data:

```
@app.route("/register", methods=["POST"])
def register():

    # Validate submission
    name = request.form.get("name")
    sport = request.form.get("sport")
    if not name or sport not in SPORTS:
        return render_template("failure.html")

    # Remember registrant
    db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", nam

    # Confirm registration
    return redirect("/registrants")
```

  - Once we've validated the request, we can use `INSERT INTO` to add a row.
  - Flask also includes a function, `redirect`, that we can use to redirect to another route.

- Similarly, for the `/registrants` route, we can `SELECT` all rows and pass them to the template as a list of rows:

```
@app.route("/registrants")
def registrants():
    registrants = db.execute("SELECT * FROM registrants")
    return render_template("registrants.html", registrants=registrants)
```

- Our `registrants.html` template will use `registrant["name"]` and `registrant["sport"]` to access the value of each key in each row:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Registrants</h1>
    <table>
        <thead>
            <tr>
                <th>Name</th>
```

```
                <th>Sport</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            {% for registrant in registrants %}
                <tr>
                    <td>{{ registrant["name"] }}</td>
                    <td>{{ registrant["sport"] }}</td>
                    <td>
                        <form action="/deregister" method="post">
                            <input name="id" type="hidden" value="{{ reg
                            <input type="submit" value="Deregister">
                        </form>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

- Our page will include another form for deregistering a person by an `id`. When we click that button, we'll see a request that just sends the `id` to our `/deregister` route.

- We'll register a few people, and go back into our terminal to see that our database now has a few rows:

```
$ sqlite3 froshims.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> SELECT * FROM registrants;
+----+--------+-----------------+
| id |  name  |      sport      |
+----+--------+-----------------+
| 1  | David  | Ultimate Frisbee |
| 2  | Carter | Basketball      |
| 3  | Emma   | Soccer          |
+----+--------+-----------------+
```

- Our `/deregister` route will take an `id` and delete that row from our database:

```
@app.route("/deregister", methods=["POST"])
def deregister():

    # Forget registrant
```

```
    id = request.form.get("id")
    if id:
        db.execute("DELETE FROM registrants WHERE id = ?", id)
    return redirect("/registrants")
```

- A URL that uses GET can also be used to trick people, since they might click on them while logged in to a website, and perform some action unintentionally, known as a cross-site request forgery (https://en.wikipedia.org/wiki/Cross-site_request_forgery).

Emails

- We can even email users with another library, `flask_mail`, in `froshims5` (https://cdn.cs50.net/2021/fall/lectures/9/src9/froshims5/):

```
# Implements a registration form, confirming registration via email

import os
import re

from flask import Flask, render_template, request
from flask_mail import Mail, Message

app = Flask(__name__)

# Requires that "Less secure app access" be on
# https://support.google.com/accounts/answer/6010255
app.config["MAIL_DEFAULT_SENDER"] = os.environ["MAIL_DEFAULT_SENDER"]
app.config["MAIL_PASSWORD"] = os.environ["MAIL_PASSWORD"]
app.config["MAIL_PORT"] = 587
app.config["MAIL_SERVER"] = "smtp.gmail.com"
app.config["MAIL_USE_TLS"] = True
app.config["MAIL_USERNAME"] = os.environ["MAIL_USERNAME"]
mail = Mail(app)

...
```

  - It turns out that we can provide configuration details like a username and password and mail server, in this case Gmail's, to the `Mail` variable, which will send mail for us.
  - We set the sensitive variables outside of our source code, in VS Code's environment, so we can avoid including them in our code.
- In our `register` route, we send an email to the user with the `mail.send()` function from the `flask_mail` library:

```
@app.route("/register", methods=["POST"])
def register():

    # Validate submission
    name = request.form.get("name")
    email = request.form.get("email")
    sport = request.form.get("sport")
    if not name or not email or sport not in SPORTS:
        return render_template("failure.html")

    # Send email
    message = Message("You are registered!", recipients=[email])
    mail.send(message)

    # Confirm registration
    return render_template("success.html")
```

- To include the libraries we need, we'll write a `requirements.txt` file with:

```
Flask
Flask-Mail
```

- Now, if we restart our server and use the form to provide an email, we'll see that we indeed get one sent to us (though it may end up in the Spam folder if we send too many to the same address)!

Sessions

- **Sessions** are how web servers remembers information about each user, which enables features like allowing users to stay logged in, and saving items to a shopping cart. These features require our server to be **stateful**, or having access to additional state, or information. HTTP on its own is **stateless**, since after we make a request and get a response, the interaction is completed.
- It turns out that servers can send another header in a response, called `Set-Cookie`:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=value
...
```

    - **Cookies** are small pieces of data from a web server that the browser saves for us. In many cases, they are large random numbers or strings used to uniquely identify and track a user between visits.

    - In this case, the server is asking our browser to set a cookie for that server, called `session` to a value of `value`.
- Then, when the browser makes another request to the same server, it'll send back the same cookie that the same server has set before:

```
GET / HTTP/1.1
Host: gmail.com
Cookie: session=value
```

- In the real world, amusement parks might give you a hand stamp so you can come back inside after leaving. Similarly, our browser is presenting our cookies back to the web server, so it can remember who we are.
- In Flask, we can use the `flask_session` library to help manage this for us, in a new app called `login` (https://cdn.cs50.net/2021/fall/lectures/9/src9/login/):

```
from flask import Flask, redirect, render_template, request, session
from flask_session import Session

# Configure app
app = Flask(__name__)

# Configure session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

    - We'll configure the session library to use the server's filesystem, and use `session` like a dictionary to store a user's name. It turns out that Flask will use HTTP cookies for us, to maintain this `session` variable for each user visiting our web server. Every user will be logged in with a different session, and we can see them in the `flask_session` directory:

```
src9/login/flask_session/ $ ls
2029240f6d1128be89ddc32729463129
```

- We'll visit our server's URL, and get redirected to the `/login` route automatically. We see a form that we can fill in, and then redirected back to the index route, `/`, with "You are logged in as David." Then, we can reload the page, or open it again in a new tab, and see the same message. We can also use the "Log out" link to log out.
- For our default `/` route, we'll redirect to `/login` if there's no name set in `session` for the user yet, and otherwise show a default `index.html` template.

```
@app.route("/")
```

```
def index():
    if not session.get("name"):
        return redirect("/login")
    return render_template("index.html")
```

- In our `index.html`, we can check if `session["name"]` exists, and show different content if so:

```
{% extends "layout.html" %}

{% block body %}

    {% if session["name"] %}
        You are logged in as {{ session["name"] }}. <a href="/logout">Lo
    {% else %}
        You are not logged in. <a href="/login">Log in</a>.
    {% endif %}

{% endblock %}
```

- For our `/login` route, we'll store `name` in `session` to the form's value sent via POST, and then redirect to the default route. If we visited the route via GET, we'll render the login form at `login.html`:

```
@app.route("/login", methods=["GET", "POST"])
  def login():
      if request.method == "POST":
          session["name"] = request.form.get("name")
          return redirect("/")
      return render_template("login.html")
```

  - Then, in our `login.html`, we can have a form that can submit to itself:

```
{% extends "layout.html" %}

{% block body %}

    <form action="/login" method="post">
        <input autocomplete="off" autofocus name="name" placeholder
        <input type="submit" value="Log In">
    </form>

{% endblock %}
```

- For the `/logout` route, we can clear the value for `name` in `session` by setting it to `None`, and redirect to `/` again:

```
@app.route("/logout")
def logout():
    session["name"] = None
    return redirect("/")
```

store, shows

- We'll look through another example, `store` (https://cdn.cs50.net/2021/fall/lectures/9/src9/store/). When we run our server and visit our site's URL, we see a list of books we can add to a virtual shopping cart.
- In the template `books.html`, we see another loop for each book:

```
{% extends "layout.html" %}

{% block body %}

    <h1>Books</h1>
    {% for book in books %}
        <h2>{{ book["title"] }}</h2>
        <form action="/cart" method="post">
            <input name="id" type="hidden" value="{{ book['id'] }}">
            <input type="submit" value="Add to Cart">
        </form>
    {% endfor %}

{% endblock %}
```

  - Notice that each `<form>` includes a hidden value of `book['id']`. When the button "Add to Cart" is clicked, that ID is sent back to the server.
- In `app.py`, we see that there's a database used to get the list of books:

```
...
# Connect to database
db = SQL("sqlite:///store.db")
...
@app.route("/")
def index():
    books = db.execute("SELECT * FROM books")
    return render_template("books.html", books=books)
...
```

- As before, we can open the database to see its schema and data:

```
src9/store/ $ sqlite3 store.db
.SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE books (id INTEGER, title TEXT NOT NULL, PRIMARY KEY(id));
sqlite> SELECT * FROM books;
+----+-----------------------------------------+
| id |                  title                  |
+----+-----------------------------------------+
| 1  | Harry Potter and the Sorcerer's Stone   |
| 2  | Harry Potter and the Chamber of Secrets |
| 3  | Harry Potter and the Prisoner of Azkaban|
| 4  | Harry Potter and the Goblet of Fire     |
| 5  | Harry Potter and the Order of the Phoenix|
| 6  | Harry Potter and the Half-Blood Prince  |
| 7  | Harry Potter and the Deathly Hallows    |
+----+-----------------------------------------+
```

- In the `/cart` route, we'll create a cart for the session automatically, and add values to it:

```
@app.route("/cart", methods=["GET", "POST"])
def cart():

    # Ensure cart exists
    if "cart" not in session:
        session["cart"] = []

    # POST
    if request.method == "POST":
        id = request.form.get("id")
        if id:
            session["cart"].append(id)
        return redirect("/cart")

    # GET
    books = db.execute("SELECT * FROM books WHERE id IN (?)", session["c
    return render_template("cart.html", books=books)
```

  - Our route can accept either a form input via POST, or a request via POST to shows the contents of the cart. We can add this check with `if request.method == "POST":`.
  - When we receive a form input, we can add it to our list of IDs in memory with `session["cart"].append(id)`.
  - When we receive a GET request, we can pass that list of IDs to our database to

retrieve the names of the books.

Searching

- Let's look at another example, `shows0` (https://cdn.cs50.net/2021/fall/lectures/9/src9/shows0/).
- When we run our server and visit the page, we see a search form that allows us to search for titles of TV shows. Our URL changes to `/search?q=cats`.
- In `app.py`, we start by opening a database, `shows.db`:

```
# Searches for shows

from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + r
    return render_template("search.html", shows=shows)
```

  - The default `/` route will show a form, where we can type in some search term.
  - The form will use the GET method to send the search query to `/search`, which in turn will use SQL to find a list of shows that match.
  - Finally, the `search.html` template will use a loop to print the list of show titles:

```
{% extends "layout.html" %}

{% block body %}

    <ul>
        {% for show in shows %}
            <li>{{ show["title"] }}</li>
        {% endfor %}
```

```
    </ul>

{% endblock %}
```

- With JavaScript in shows1 (https://cdn.cs50.net/2021/fall/lectures/9/src9/shows1/), we can show a partial list of results as we type.
- If we look at the source code of our page, we see that there's JavaScript that sets the HTML of an empty `<ul>`.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wi
        <title>shows</title>
    </head>
    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="se

        <ul></ul>

        <script>

            let input = document.querySelector('input');
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
                let shows = await response.text();
                document.querySelector('ul').innerHTML = shows;
            });

        </script>

    </body>
</html>
```

  - In our JavaScript code, we start by selecting the input box. Then, every time the input changes, we use a function called `fetch` to get more data from the server without changing the URL of the current page. Here, we're using the `/search` route.
  - Then, we store the text of the response in a variable called `shows`, and then using that in our HTML.
  - We can use the Network tab in Developer Tools again, to see that the response of the `/search` route is a list of `<li>` elements.

- We can even visit `/search?q=c` manually to see the same response in our browser:

```
<li>Catweazle</li>

<li>Ace of Wands</li>

<li>The Adventures of Don Quick</li>

<li>Albert and Victoria</li>

<li>All My Children</li>

...
```

- It turns out that we can use another format for our data, JSON, JavaScript Object Notation in `shows2` (https://cdn.cs50.net/2021/fall/lectures/9/src9/shows2/), instead of returning a long list of `<li>` elements that are already created. Now, when we run this version and visit the same URL, we see:

```
[{"id":63881,"title":"Catweazle"},{"id":65269,"title":"Ace of Wands"},{"
```

  - We see a more efficient notation, that includes the raw data of an `id` and `title` for each show.
- Then, in the source code of our `index.html`, we'll use JavaScript to add those shows to our page programmatically:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-wid
        <title>shows</title>
    </head>
    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="te

        <ul></ul>

        <script>

            let input = document.querySelector('input');
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
```

```
                let shows = await response.json();
                let html = '';
                for (let id in shows) {
                    let title = shows[id].title.replace('<', '&lt;').rep
                    html += '<li>' + title + '</li>';
                }
                document.querySelector('ul').innerHTML = html;
            });

        </script>

    </body>
</html>
```

- With `response.json()`, we turn the raw response into a list of dictionaries, and then we can build a string of HTML with the `for` loop.

- So, in this web application, HTML is used to for the view itself, Python is used to write the code on our server that sends back data, and JavaScript is used to make the page interactive and dynamic.

This is CS50x

CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 10

The End

- We'd like to thank everyone who helped make CS50 possible:
  - Office for the Arts at Harvard, Memorial Hall / Lowell Hall Complex
  - Harvard FAS's Education Support Services
  - CS50's own production team behind the cameras
  - CS50's staff and course assistants

- And you may have noticed a few puppets around the theater, to add some interest to some shots in our lecture videos.
- Don't forget that, ultimately:

  > what ultimately matters in this course is not so much where you end up relative to your classmates but where you end up relative to yourself when you began

- Recall that we started by printing out characters in the shape of a pyramid from the Super Mario Bros. games, then progressed to analyzing the readability of sentences, implementing algorithms for elections, and applying filters to images.
- Then, we learned about data structures, building hash tables and tries to search for data quickly. Finally, we learned several new languages, Python, SQL, HTML, CSS, and JavaScript to build our own web application.
- We've learned some first principles:
  - computational thinking, where we think more logically and methodically
  - using algorithms to solve problems, by taking some input and producing some output
- And now, with all the tools and skills we've learned, we hope that in the future we'll consider whether we, and others, *should* be building (and using) programs and applications, just because we *can*.
- We've learned to evalute our work on the axes of correctness, design, and style.
- We can also use the concept of abstraction to break down a complex problem into layers and solve them one at a time.
- We aspire to be precise, too, in our instructions, whether in our programs or other humans.
  - We ask a volunteer to come on stage. They are able to see a drawing of a cube that the audience can't, and give a series of verbal instructions to the audience for how to draw it themselves. Since everyone interpreted each instruction slightly differently, the final drawings all ended up very different.
  - We could have used abstraction to just say, "draw a cube," but that wouldn't include details about its size or angle. We could have been more precise by saying, "draw a line at 45 degrees," but that would be tricky to follow as well.
- We have another volunteer come on the stage, and the audience now tells the volunteer to draw a stick figure saying "Hi."

Tools

- After CS50, we might want to use more advanced tools for developing software:
  - https://developer.apple.com/xcode/ (https://developer.apple.com/xcode/)

    - https://docs.microsoft.com/en-us/windows/wsl/about (https://docs.microsoft.com/en-us/windows/wsl/about)
    - …
- We can watch a workshop (https://www.youtube.com/watch?v=MJUJ4wbFm_A) from a former TF on Git, a version-control software used to manage different versions of code and enable collaboration with others.
- VS Code (https://code.visualstudio.com/) is also available for download, to our own computers, rather than running in the cloud.
- We can host a web site with services like:
    - https://pages.github.com/ (https://pages.github.com/)
    - https://www.netlify.com/ (https://www.netlify.com/)
    - …
- We can run web apps with hosts like:
    - https://www.heroku.com/platform (https://www.heroku.com/platform)
    - https://aws.amazon.com/education/awseducate/ (https://aws.amazon.com/education/awseducate/)
    - https://azure.microsoft.com/en-us/free/students/ (https://azure.microsoft.com/en-us/free/students/)
    - https://cloud.google.com/edu/students (https://cloud.google.com/edu/students)
    - https://education.github.com/pack (https://education.github.com/pack)
    - …
- And sources to read more on technology and programming include:
    - https://www.reddit.com/r/learnprogramming/ (https://www.reddit.com/r/learnprogramming/)
    - https://www.reddit.com/r/programming/ (https://www.reddit.com/r/programming/)
    - https://stackoverflow.com/ (https://stackoverflow.com/)
    - https://serverfault.com/ (https://serverfault.com/)
    - https://techcrunch.com/ (https://techcrunch.com/)
    - https://news.ycombinator.com/ (https://news.ycombinator.com/)
    - …
- CS50 has many communities (https://cs50.harvard.edu/x/communities) as well.

Quiz Show

- We host a quiz show with the audience, with the following questions:

- What does CSS stand for?
    - Cascading Style Sheet
- Which best describes the role of a compiler?
    - Turn source code into machine code
- What is the type of `argc`?
    - `int`
- What is the searching efficiency of a Balanced Binary Search Tree?
    - $O(\log n)$
- What was the CS50 Duck's Halloween costume?
    - Vampire
- In C, how can we unify several variables of different types into a single, new type?
    - Structs
- In Python, which of the following statements is false?
    - Arrays in Python are of a fixed size
- What does `strcmp` return in C?
    - An integer
- What does David Malan's phone number [949-468-2750] play when you call it?
    - Never Gonna Give You Up
- From which of the following places does malloc get free memory for a memory to use?
    - heap
- Suppose I have an unsorted list of items (store receipts, perhaps). Should I sort the items before searching for an element?
    - If you will be searching the list many times, then yes, you should sort first
- When you run the "CREATE INDEX" command in SQL, what type of data structure do you create?
    - B-trees
- What HTTP status code means "I'm a teapot"?
    - 418
- What is an example of a SQL injection attack?
    - When someone submits malicious SQL commands via a web form
- How are the elements of an array stored in memory?
    - Contiguously
- Which SQL query would allow you to select the ID of a specific movie star (Zendaya) in a table of movie stars?

        - `SELECT id FROM moviestars WHERE name = 'Zendaya'`
    - Why is a hash table faster to search than a linked list, even though the runtime for both is $O(n)$?
        - The hash table creates shorter linked lists to search rather than one long linked list
    - Game of thrones is a...
        - All of the above (Comedy, Drama, Historical, Fantasy, Documentary, Romance, Sci-Fi)
    - Which of the following is a golden rule when allocating memory?
        - All of the above (Every block of memory that you malloc must be freed, Only memory that you malloc should be freed, Do not free a block of memory more than once)
    - What do the binary bulbs on stage spell today?
        - ❤️

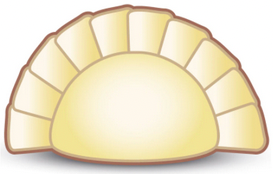
Emoji

- In week 0, we talked about representing numbers and letters with ASCII. Emoji, too, might be represented with four bytes, like `11110000 10011111 10011000 10110111`.
- And you may have noticed that there are occasionally new emoji, governed by the Unicode Consortium (https://unicode.org/consortium/consort.html).
- Today we're joined by Jennifer 8. Lee, '99, one of David's former classmates, who has been involved in many endeavors (https://www.jennifer8lee.com/), but particularly for advocating for emojis across cultures and groups of people:

HIJAB emoji
with Rayouf Alhumedhi

SAUNA emoji
with the Finnish government

RED ENVELOPE emoji
with Baidu



DUMPLING emoji
through Kickstarter

BROCCOLI emoji
with vegetarians

DNA emoji
with GE and the American Chemical Society

- Jennifer took CS50 in 1994, and one of her classmates was the first intern for Netscape (https://en.wikipedia.org/wiki/Netscape). In fact, Google did not exist when she was an undergrad.
- A few years ago, Jennifer was texting her friend, Yiying Lu, when they both realized that a dumpling emoji doesn't exist. So her friend, who is a designer, came up with her own image of a dumpling, and Jennifer was inspired to investigate who controls emoji.
- She discovered the Unicode Consortium, a non-profit organization based in California, with technology companies and other organizations as voting members.
- It turns out that, for $75 a year, individuals can become non-voting members who can attend meetings. Jennifer signed up and attended a meeting which was run by just a few people around a table.
- So with her friend, Jennifer founded Emojination (http://www.emojination.org/), a group with the motto "Emoji By The People, For The People."
- After running campaigns to gain popular support, they created a proposal for the emoji subcommittee in the Unicode Consortium. Once the proposal is approved, it is voted on.
- There are also objective factors for whether an emoji is included:
    - Popular demand, or frequent requests
    - Multiple usages or meanings
    - Visual distinctiveness, or easily recognizable at small sizes
    - Whether they complete some gap, such as an orange heart among red, yellow, green, and blue hearts

    - Existing vendor compatibility, such as if one company already unofficially supports it
- Factors that are considered against an emoji's inclusion:
    - Too specific, or narrow
    - Redundant, or too similar to an existing emoji
    - Not visually discernible, like a cave
    - No logos, brands, deities, celebrities
    - No more flags
- Once the proposals for new emoji are voted on, which happens once a year, then the companies that build operating systems and software add them to their devices and apps.
- Emoji came about as little symbols popular in Japan many years ago, and as smartphones became popular as well, there needed to be a standard for all the software and devices that people use.
- And the Unicode Consortium's mission is to "enable people around the world to use computers in any language".
- So the Unicode Consortium has three main projects:
    - Encoding characters (including, but not limited to emoji), with over 100,000 now
    - Localization resources, like a repository of currencies or date and time formats for different countries
    - Programming libraries, so developers can localize their software more easily
- In 2010, the Unicode 6.0 standard included the first emoji.
- A Unicode code point is a unique number assigned to each Unicode character.
- And some emoji might still be ambiguous, like 🙃.
- It turns out that anyone can propose a emoji, as well, and many individuals and organizations have created proposals that were ultimately accepted, with the help of Emojination.
- Jennifer personally cares about emoji because she grew up speaking Chinese, and many emoji and Chinese characters have similarities in representing objects or being combined to represent another concept.
    - The Chinese character for a forest is the character for a tree, right next to another character for a tree.
    - The character for "house" or "home" is a roof over the character for pig, as though a home is where your pigs are.
    - The character for "good" is the character for a woman next to a character for a child, which might be problematic by today's standards.
- These examples, and more, will be in The Hanmoji Handbook (https://hanmoji.org/),

coming out next year.

- It turns out that the Unicode Standard also allows for a special character, the zero-width joiner, that combines multiple emoji. An emoji for an occupation, such as a farmer or chef, might be a man or woman emoji combined with a tractor or frying pan.
- Skin tones, too, are combinations of existing emoji and certain colors.
- It turns out that the face with tears of joy emoji, 😂 is almost 10% of all emoji usage. Almost all of the rest are used relatively infrequently, since there are so many.
- Next year, there will be more new emoji, including hearts with more colors, birds, flowers, animals, food, objects, and even a shaking face.
- Perhaps you, too, will come up with a new emoji proposal that can impact billions of people!