

Harvard CS50x Course Notes

Lecture 0-Scratch

- [Welcome!](#)
- [What's Ahead](#)
- [Community!](#)
- [Computer Science](#)
- [ASCII](#)
- [Unicode](#)
- [Representation](#)
- [Algorithms](#)
- [Pseudocode](#)
- [Artificial Intelligence](#)
- [Scratch](#)
- [Hello World](#)
- [Hello, You](#)
- [Meow and Abstraction](#)
- [Conditionals](#)
- [Oscartime](#)
- [Ivy's Hardest Game](#)
- [Summing Up](#)

Welcome!

- This class is about more than computer programming!
- Indeed, this class is about problem-solving in a way that is exceedingly empowering! You will likely take the problem solving that you learn here will likely be instantly applicable to your work beyond this course and even your career as a whole!
- However, it will not be easy! You will be “drinking from the firehose” of knowledge during this course. You’ll be amazed at what you will be able to accomplish in the coming weeks.
- This course is far more about you advancing “you” from “where you are today” than hitting some imagined standard.
- The most important opening consideration in this course: Give the time you need to learn through this course. Everyone learns differently. If something does not work out well at the start, know that with time you will grow and grow in your skill.

- Don't be scared if this is your first computer science class! For most of your peers, this is their first computer science class too!

What's Ahead

- You will be learning this week about Scratch, a visual programming language.
- Then, in future weeks, you will learn about C. That will look something like this:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

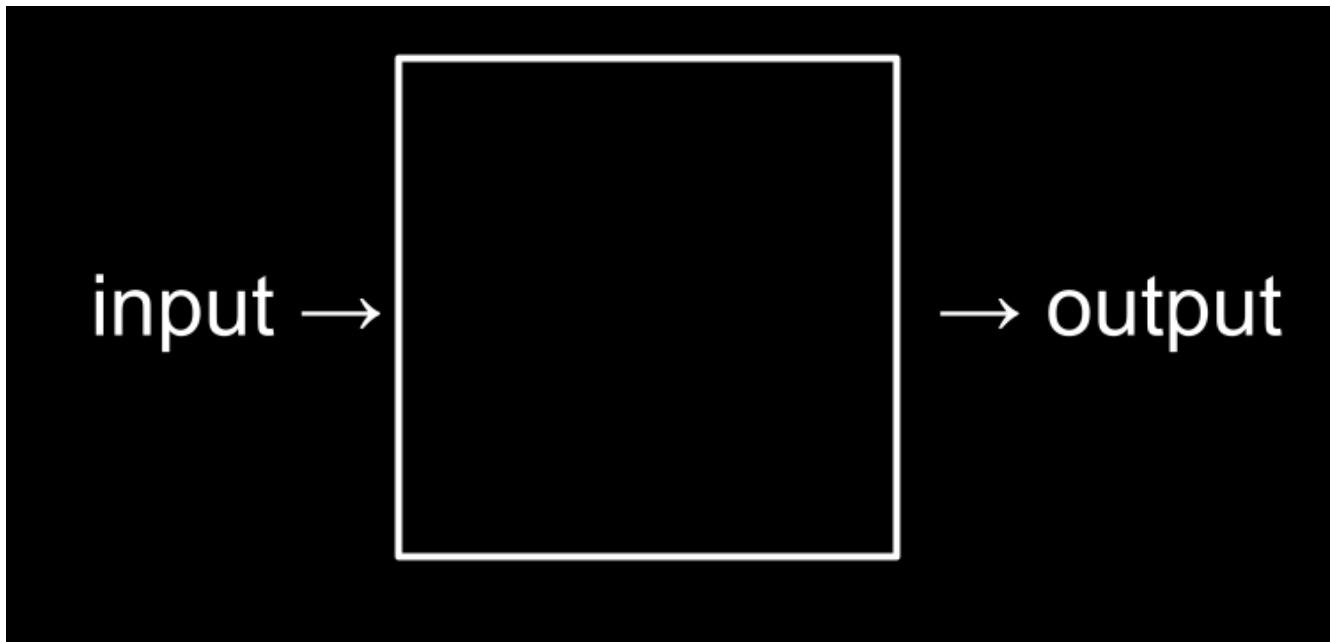
- Further, as the weeks progress, you will learn about algorithms.
- You will learn about memory.
- You will learn about buggy code and what causes computer crashes.
- You will learn about data structures such as a hash table.
- Then, we will transition to a new, higher-level language called *Python*. Your code will look something like this:
- This class will give you a strong understanding of how recent programming languages developed from the earlier ones.
- You will learn SQL, JavaScript, HTML, and CSS.
- We will also be looking at how we can use databases and third-party frameworks to build web applications.

Community!

- You are part of a community of those taking this course at Harvard College, Harvard Extension School, and via edX.org.
- Puzzle Day and the CS50 Fair
- You can attend CS50 Lunches and CS50 Hackathon, if you are student on Harvard's campus.

Computer Science

- Essentially, computer programming is about taking some input and creating some output - thus solving a problem. What happens in between the input and output, what we could call a *black box*, is the focus of this course.



- For example, we may need to take attendance for a class. We could use a system called *unary* to count, one finger at a time.
- Computers today count using a system called *binary*. It's from the term *binary digit* that we get a familiar term called *bit*. A *bit* is a zero or one: on or off.
- Computers only speak in terms of zeros and ones. Zeros represent *off*. Ones represent *on*. Computers are millions, and perhaps billions, of transistors that are being turned on and off.
- If you imagine using a light bulb, a single bulb can only count from zero to one.
- However, if you were to have three light bulbs, there are more options open to you!
- Using three light bulbs, the following could represent zero:
- Similarly, the following would represent one:
- By this logic, we could propose that the following equals two:
- Extending this logic further, the following represents three:
- Four would appear as:
- We could, in fact, using only three light bulbs count as high as seven!
- As a heuristic, we could imagine that the following values represent each possible place in our *binary digit*:
- Computers use 'base-2' to count. This can be pictured as follows:
- Therefore, you could say that it would require three bits (the four's place, the two's place, and the one's place) to represent a number as high as seven.
- Computers generally use eight bits (also known as a *byte*) to represent a number. For example, `00000101` is the number 5 in *binary*. `11111111` represents the number 255.

ASCII

- Just as numbers are binary patterns of ones and zeros, letters are represented using ones and zeros too!

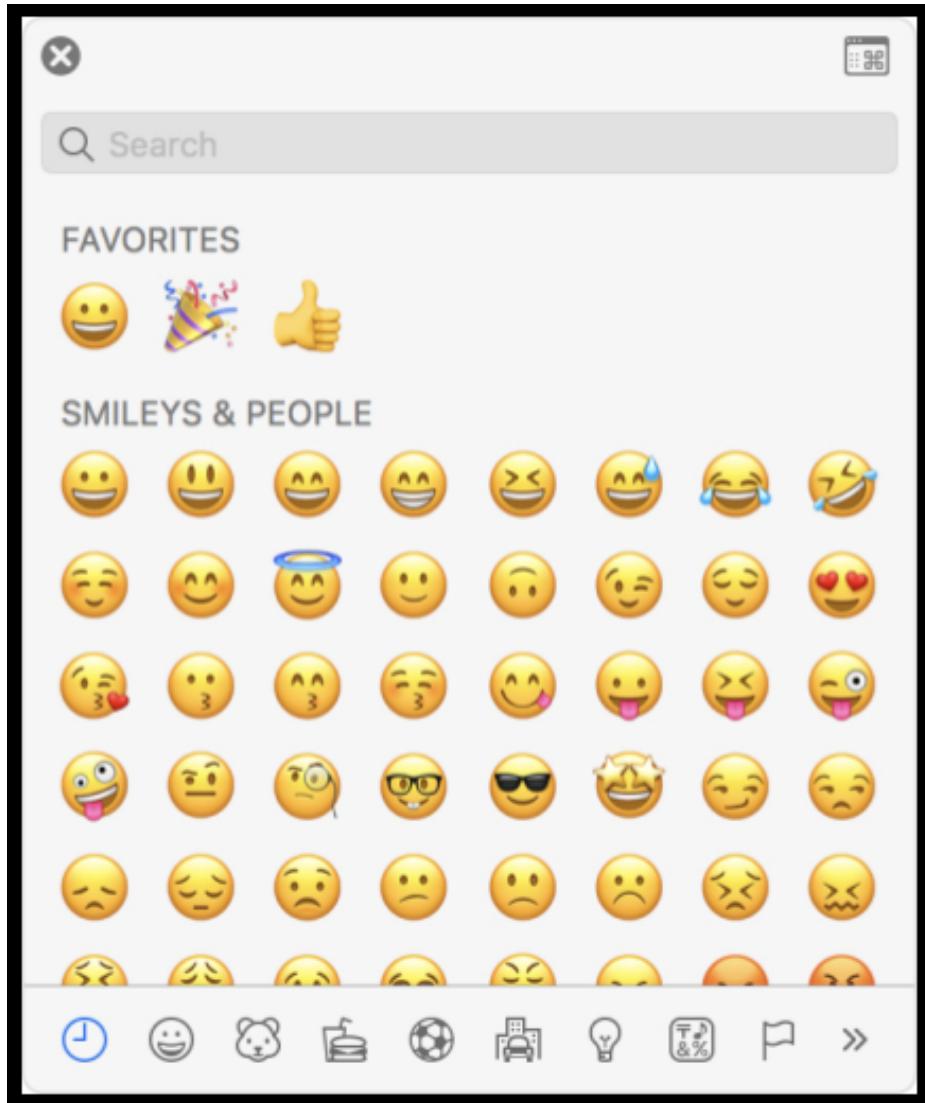
- Since there is an overlap between the ones and zeros that represent numbers and letters, the ASCII standard was created to map specific letters to specific numbers.
- For example, the letter A was decided to map to the number 65. 01000001 represents the number 65 in binary.
- If you received a text message, the binary under that message might represent the numbers 72, 73, and 33. Mapping these out to ASCII, your message would look as follows:
- Thank goodness for standards like ASCII that allow us to agree upon these values!
- Here is an expanded map of ASCII values:

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	'	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	\$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	
13	CR	29	GS	45	-	61	=	77	M	93]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	-
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

- If you wish, you can learn more about [ASCII](#).
- Since binary can only count up to 255 we are limited to the number of characters represented by ASCII.

Unicode

- As time has rolled on, there are more and more ways to communicate via text.
- Since there were not enough digits in binary to represent all the various characters that could be represented by humans, the *Unicode* standard expanded the number of bits that can be transmitted and understood by computers. Unicode includes not only special characters, but emoji as well.
- There are emoji that you probably use every day. The following may look familiar to you:



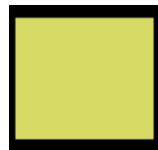
- Computer scientists faced a challenge when wanting to assign various skin tones to each emoji to allow the communication to be further personalized. In this case, the creators and contributors of emoji decided that the initial bits would be the structure of the emoji itself, followed by skin tone.
- For example, the unicode for a generic thumbs up is `U+1F44D`. However, the following represents the same thumbs up with a different skin tone: `U+1F44D U+1F3FD`.
- More and more features are being added to the Unicode standard to represent further characters and emoji.
- If you wish, you can learn more about [Unicode](#).
- If you wish, you can learn more about [emoji](#).

Representation

- Zeros and ones can be used to represent color.
- Red, green, and blue (called `RGB`) is a combination of three numbers.



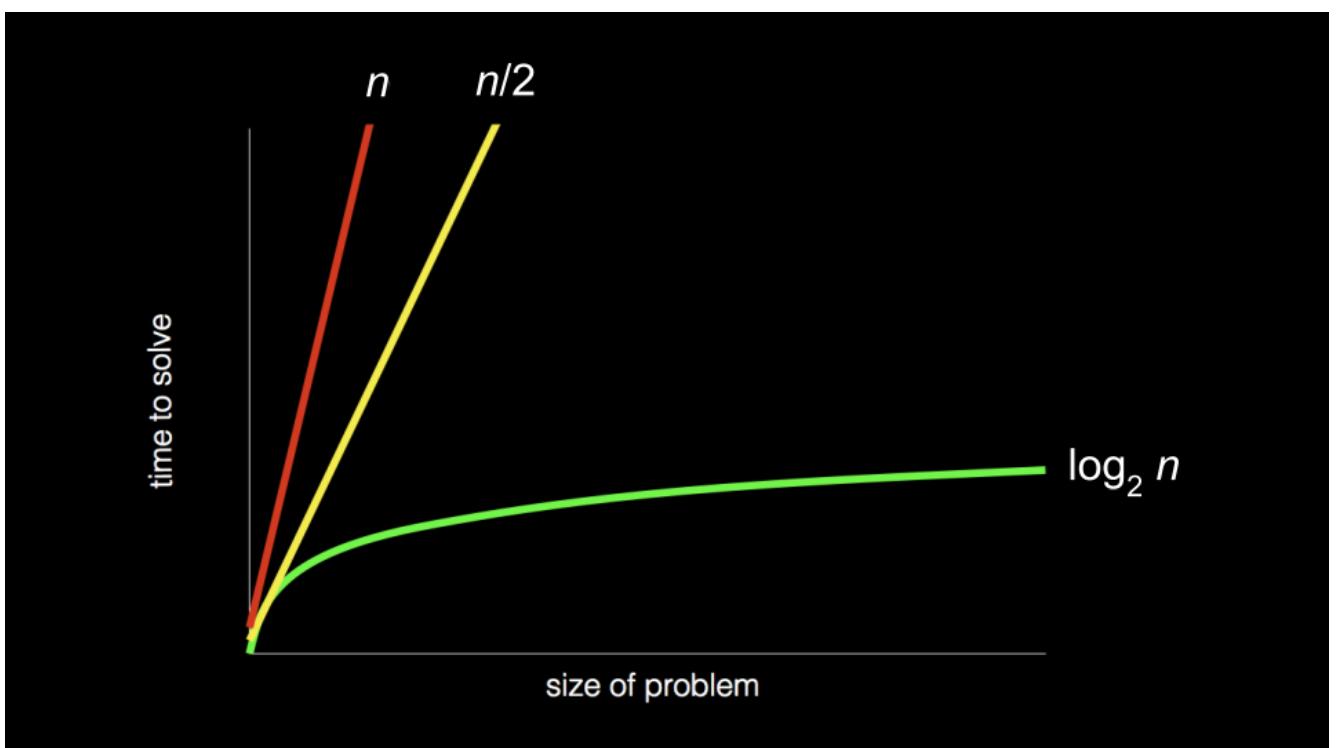
- Taking our previously used 72, 73, and 33, which said `HI!` via text, would be interpreted by image readers as a light shade of yellow. The red value would be 72, the green value would be 73, and the blue would be 33.



- Further, zeros and ones can be used to represent images, videos, and music!
- Images are simply collections of RGB values.
- Videos are sequences of many images that are stored together, just like a flipbook.
- Music can be represented through MIDI data.

Algorithms

- Problem-solving is central to computer science and computer programming.
- Imagine the basic problem of trying to locate a single name in a phone book.
- How might you go about this?
- One approach could be to simply read from page one to the next to the next until reaching the last page.
- Another approach could be to search two pages at a time.
- A final and perhaps better approach could be to go to the middle of the phone book and ask, “Is the name I am looking for to the left or to the right?” Then, repeat this process, cutting the problem in half and half and half.
- Each of these approaches could be called algorithms. The speed of each of these algorithms can be pictured as follows in what is called *big-O notation*:



Notice that the first algorithm, highlighted in red, has a big-O of n because if there are 100 names in the phone book, it could take up to 100 tries to find the correct name. The second algorithm, where two pages were searched at a time, has a big-O of ' $n/2$ ' because we searched twice as fast through the pages. The final algorithm has a big-O of $\log_2 n$ as doubling the problem would only result in one more step to solve the problem.

Pseudocode

- The ability to create *pseudocode* is central to one's success in both this class and in computer programming.
- Pseudocode is a human-readable version of your code. For example, considering the third algorithm above, we could compose pseudocode as follows:

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

- Pseudocoding is such an important skill for at least two reasons. First, when you pseudocode before you create formal code, it allows you to think through the logic of your problem in advance. Second, when you pseudocode, you can later provide this information to others that are seeking to understand your coding decisions and how your code works.
- Notice that the language within our pseudocode has some unique features. First, some of these lines begin with verbs like *pick up*, *open*, *look at*. Later, we will call these *functions*.
- Second, notice that some lines include statements like `if` or `else if`. These are called *conditionals*.
- Third, notice how there are expressions that can be stated as *true* or *false*, such as "person is earlier in the book." We call these *boolean expressions*.
- Finally, notice how these statements like "go back to line 3." We call these *loops*.
- These building blocks are the fundamentals of programming.
- In the context of *Scratch*, which is discussed below, we will use each of the above basic building blocks of programming.

Artificial Intelligence

- Consider how we can utilize the building blocks above to start creating our own artificial intelligence. Look at the following pseudocode:

```

1 If student says hello
2     Say hello back
3 Else if student says goodbye
4     Say goodbye back
5 Else if student asks how you are
6     Say you're well
7 Else if student asks why 111 in binary is 7 in decimal
8 ...

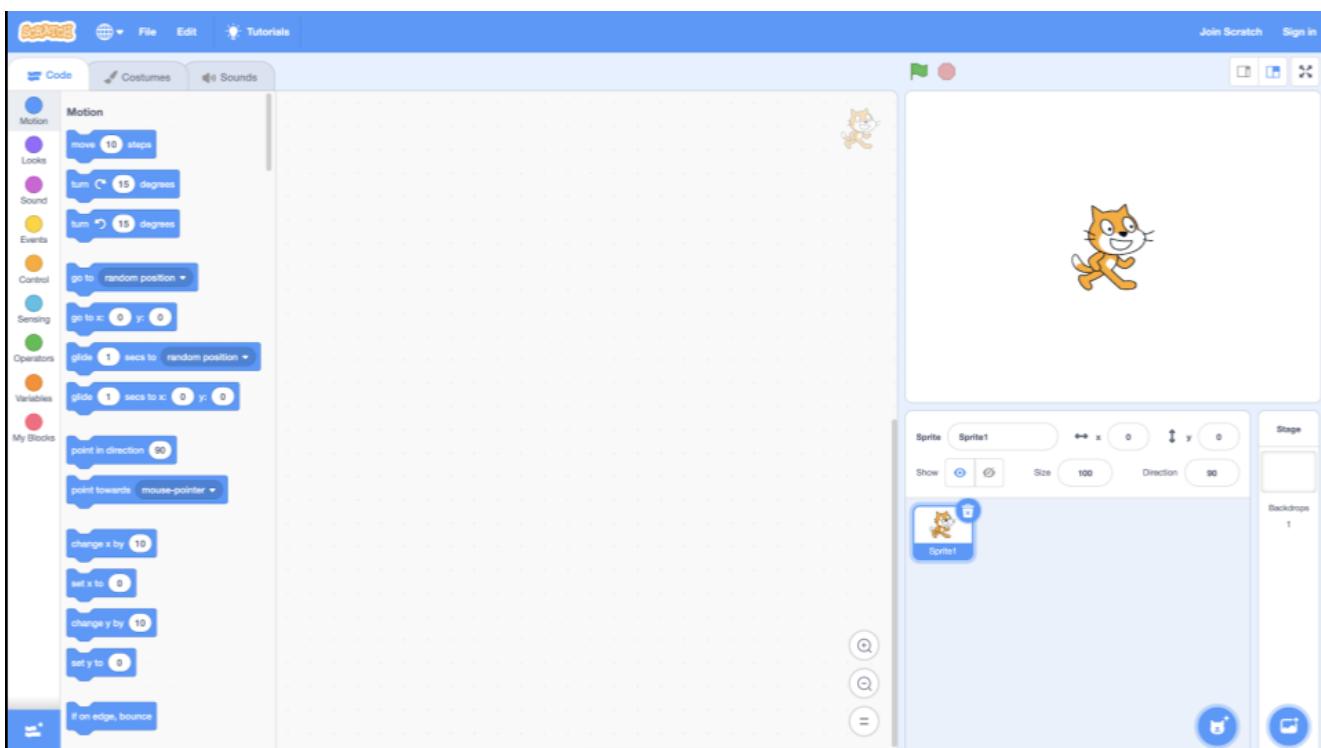
```

Notice how just to program a handful of interactions, many lines of code would be required. How many lines of code would be required for thousands or tens of thousands of possible interactions?

- `large language models` look at patterns in large blocks of language. Such language models attempt to create a best guess of what words come after one another or alongside one another.
- As very useful in many avenues of life and work, we stipulate that the utilization of AI-based software other than CS50's own is *not reasonable*.
- CS50's own AI-based software tool called [CS50 Duck](#) is an AI helper that you can use during this course. It will help you, but not give away the entire answers to the course's problems.

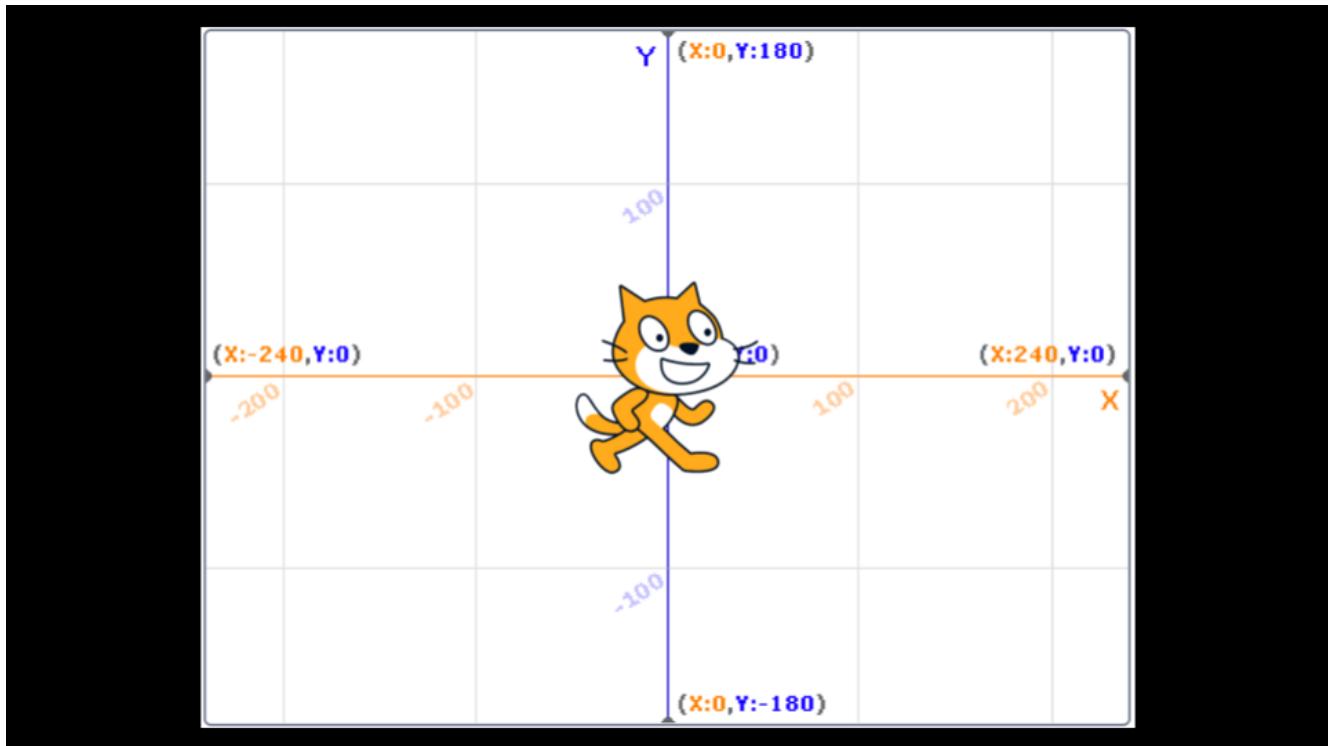
Scratch

- *Scratch* is a visual programming language developed by MIT.
- Scratch utilizes the same essential coding building blocks that we covered earlier in this lecture.
- Scratch is a great way to get into computer programming because it allows you to play with these building blocks in a visual manner, not having to be concerned about the syntax of curly braces, semicolons, parentheses, and the like.
- Scratch `IDE` (integrated development environment) looks like the following:



Notice that on the left, there are *building blocks* that you can use in your programming. To the immediate right of the building blocks, there is the area to which you can drag blocks to build a program. To the right of that, you see the *stage* where a cat stands. The stage is where your programming comes to life.

- Scratch operates on a coordinate system as follows:



Notice that the center of the stage is at coordinate (0,0). Right now, the cat's position is at that same position.

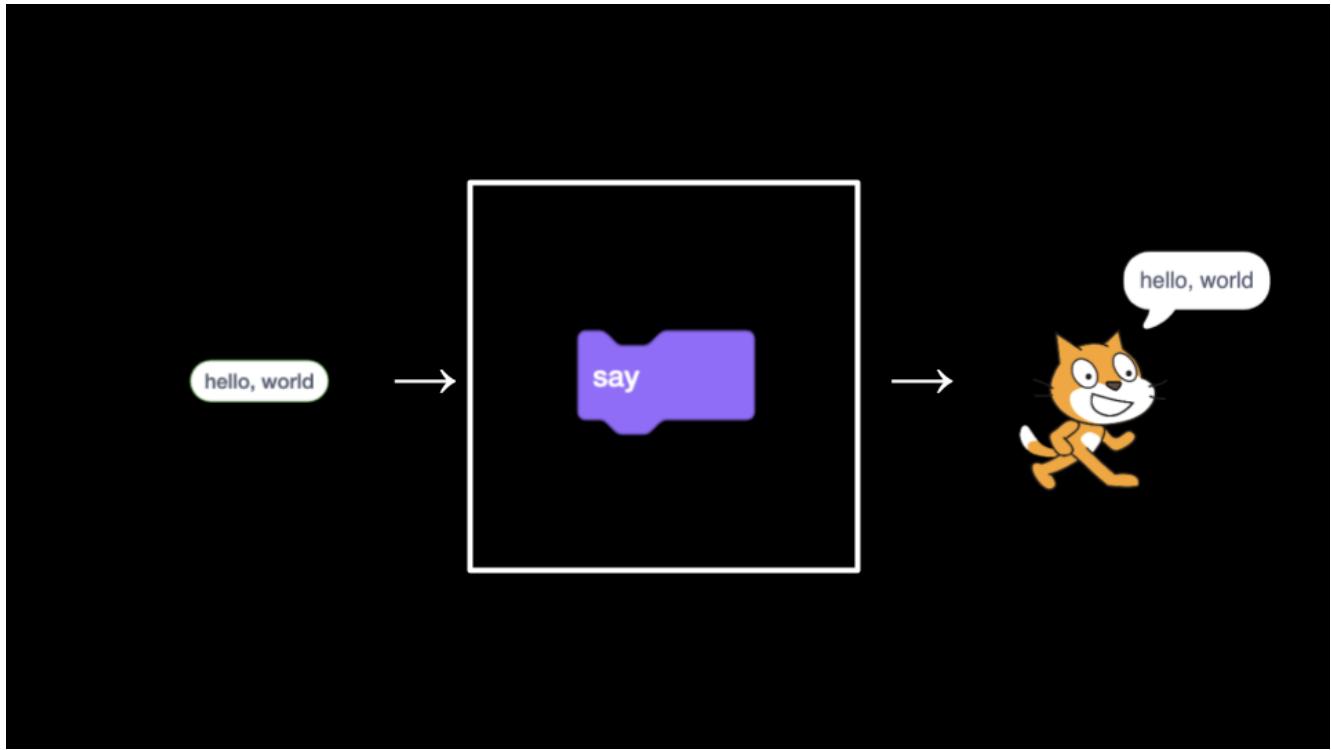
Hello World

- To begin, drag the “when green flag clicked” building block to the programming area. Then, drag the `say` building block to the programming area and attach it to the previous block.

```
1 when green flag clicked  
2 say [hello, world]
```

Notice that when you click the green flag now, on the stage, the cat says, “hello world.”

- This illustrates quite well what we were discussing earlier regarding programming:



Notice that the input `hello world` is passed to the function `say`, and the *side effect* of that function running is the cat saying `hello world`.

Hello, You

- We can make your program more interactive by having the cat say `hello` to someone specific. Modify your program as below:

```

1 when green flag clicked
2 ask [What's your name?] and wait
3 say (join [hello,] (answer))

```

Notice that when the green flag is clicked, the function `ask` is run. The program prompts you, the user, `What's your name?` It then stores that name in the *variable* called `answer`. The program then passes `answer` to a special function called `join`, which combines two strings of text `hello`, and whatever name was provided. These collectively are passed to the `say` function. The cat says, `Hello,` and a name. Your program is now interactive.

- Quite similarly, we can modify our program as follows:

```

1 when green flag clicked
2 ask [What's your name?] and wait
3 speak (join [hello,] (answer))

```

Notice that this program, when the green flag is clicked, passes the same variable, joined with `hello`, to a function called `speak`.

Meow and Abstraction

- Along with pseudocoding, *abstraction* is an essential skill and concept within computer programming.

- Abstraction is the act of simplifying a problem into smaller and smaller problems.
- For example, if you were hosting a huge dinner for your friends, the *problem* of having to cook the entire meal could be quite overwhelming! However, if you break down the task of cooking the meal into smaller and smaller tasks (or problems), the big task of creating this delicious meal might feel less challenging.
- In programming, and even within Scratch, we can see abstraction in action. In your programming area, program as follows:

```

1 when green flag clicked
2 play sound (Meow v) until done
3 wait (1) seconds
4 play sound (Meow v) until done
5 wait (1) seconds
6 play sound (Meow v) until done

```

Notice that you are doing the same thing over and over again. Indeed, if you see yourself repeatedly coding the same statements, it's likely the case that you could program more artfully – abstracting away this repetitive code.

- You can modify your code as follows:

```

1 when green flag clicked
2 repeat (3)
3 play sound (Meow v) until done
4 wait (1) seconds

```

Notice that the loop does exactly as the previous program did. However, the problem is simplified by abstracting away the repetition to a block that *repeats* the code for us.

- We can even advance this further by using the `define` block, where you can create your own block (your own function)! Write code as follows:

```

1 define meow
2 play sound (Meow v) until done
3 wait (1) seconds
4
5 when green flag clicked
6 repeat (3)
7 meow

```

Notice that we are defining our own block called `meow`. The function plays the sound `meow`, then waits one second. Below that, you can see that when the green flag is clicked, our `meow` function is repeated three times.

- We can even provide a way by which the function can take an input `n` and repeat a number of times:

```
1 define meow n times
2 repeat (n)
3 play sound [meow v] until done
4 wait (1) seconds
```

Notice how `n` is taken from “meow n times.” `n` is passed to the meow function through the `define` block.

- The cat, by the way, we can call a `sprite` – a general term used in game programming for an object or character on the screen with which the player will interact.

Conditionals

- *conditionals* are an essential building block of programming, where the program looks to see if a specific condition has been met. If a condition is met, the program does something.
- To illustrate a conditional, write code as follows:

```
1 when green flag clicked
2 forever
3 if <touching (mouse-pointer v)?> then
4 play sound (Meow v) until done
```

Notice that the `forever` block is utilized such that the `if` block is triggered over and over again, such that it can check continuously if the cat is touching the mouse pointer.

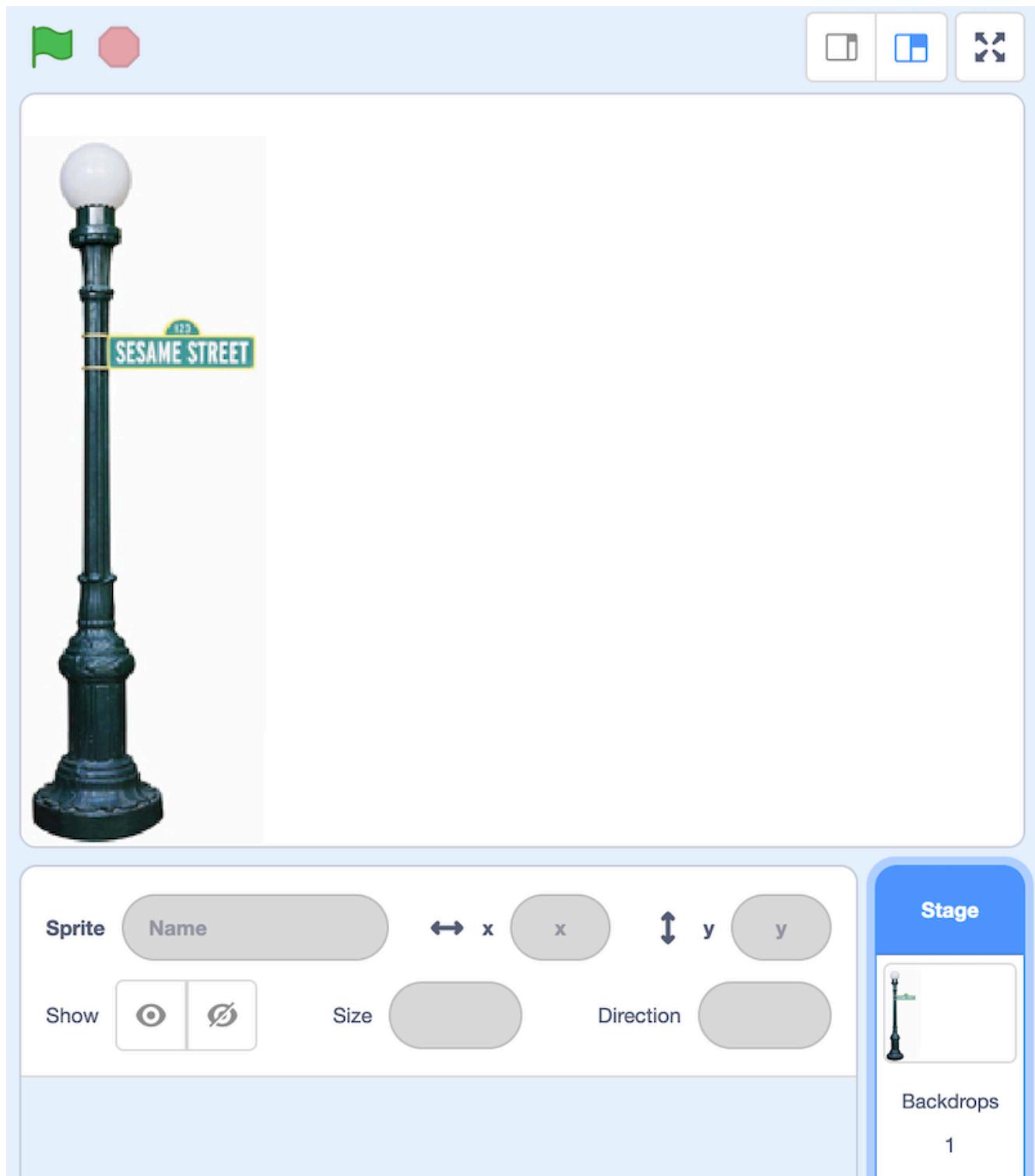
- We can modify our program as follows to integrate video sensing:

```
1 when video motion > (50)
2 play sound (Meow v) until done
```

- Remember, programming is often a process of trial and error. If you get frustrated, take time to talk yourself through the problem at hand. What is the specific problem that you are working on right now? What is working? What is not working?

Oscartime

- We showed you in this lecture a number of Scratch programs to stoke your imagination.
- *Oscartime* is one of David’s own Scratch programs – though the music may haunt him because of the number of hours he listened to it while creating this program. Take a few moments to play through the game yourself.
- Building Oscartime ourselves, we first add the lamp post.



- Then, write code as follows:

```
1 when green flag clicked
2 switch costume to (oscar1 v)
3 forever
4 if <touching (mouse-pointer v)?> then
5 switch costume to (oscar2 v)
6 else
7 switch costume to (oscar1 v)
```

Notice that moving your mouse over Oscar changes his costume. You can learn more by [exploring these code blocks](#).

- Then, modify your code as follow to create a falling piece of trash:

```
1 when green flag clicked
2 go to x: (pick random (-240) to (240)) y: (180)
3 forever
4 if <(distance to (floor v)) > (0)> then
5 change y by (-3)
```

Notice that the trash's position on the y-axis always begins at 180. The x position is randomized. While the trash is above the floor, it goes down 3 pixels at a time. You can learn more by [exploring these code blocks](#).

- Next, modify your code as follows to allow for the possibility of dragging trash.

```
1 when green flag clicked
2 forever
3 if <<mouse down?> and <touching (mouse-pointer v) ?>> then
4 go to (mouse-pointer v)
```

You can learn more by [exploring these code blocks](#).

- Next, we can implement the scoring variables as follows:

```
1 when green flag clicked
2 forever
3 if <touching (Oscar v) ?> then
4 change (score) by (1)
5 go to x: (pick random (-240) to (240)) y: (180)
```

You can learn more by [exploring these code blocks](#).

- Go try the full game [Oscartime](#).

Ivy's Hardest Game

- Moving away from Oscartime to Ivy's Hardest Game, we can now imagine how to implement movement within our program.
- Our program has three main components.
- First, write code as follows:

```
1 when green flag clicked
2 go to x: (0) y: (0)
3 forever
4 listen for keyboard
5 feel for walls
```

Notice that when the green flag is clicked, our sprite moves to the center of the stage at coordinates (0,0) and then listens for the keyboard and checks for walls forever.

- Second, add this second group of code blocks:

```
1 define listen for keyboard
2 if <key (up arrow v) pressed?> then
3 change y by (1)
4 end
5 if <key (down arrow v) pressed?> then
6 change y by (-1)
7 end
8 if <key (right arrow v) pressed?> then
9 change x by (1)
10 end
11 if <key (left arrow v) pressed?> then
12 change x by (-1)
13 end
```

Notice how we have created a custom `listen for keyboard` script. For each of our arrow keys on the keyboard, it will move the sprite around the screen.

- Finally, add this group of code blocks:

```
1 define feel for walls
2 if <touching (left wall v) ?> then
3 change x by (1)
4 end
5 if <touching (right wall v) ?> then
6 change x by (-1)
7 end
```

Notice how we also have a custom `feel for walls` script. When a sprite touches a wall, it moves it back to a safe position – preventing it from walking off the screen.

- You can learn more by [exploring these code blocks](#).
- Go try the full game [Ivy's Hardest Game](#).
- Scratch allows for many sprites to be on the screen at once.
- Adding another sprite, add the following code blocks to your program:

```
1 when green flag clicked
2 go to x: (0) y: (0)
3 point in direction (90)
4 forever
5 if <<touching (left wall v)?> or <touching (right wall v)?>> then
6 turn right (180) degrees
7 end
8 move (1) steps
9 end
```

Notice how the Yale sprite seems to get in the way of the Harvard sprite by moving back and forth. When it bumps into a wall, it turns around until it bumps the wall again. You can learn more by [exploring these code blocks](#).

- You can even make a sprite follow another sprite. Adding another sprite, add the following code blocks to your program:

```
1 when green flag clicked
2 go to (random position v)
3 forever
4 point towards (Harvard v)
5 move (1) steps
```

Notice how the MIT logo now seems to follow around the Harvard one. You can learn more by [exploring these code blocks](#).

- Go try the full game [Ivy's Hardest Game](#).

Summing Up

In this lesson, you learned how this course sits in the wide world of computer science and programming. You learned...

- Few students come to this class with prior programming experience!
- You are not alone! You are part of a community.
- Problem solving is the essence of the work of computer scientists.
- This course is not simply about programming – this course will introduce you to a new way of learning that you can apply to almost every area of life.
- How numbers, text, images, music, and video are understood by computers.
- The fundamental programming skill of pseudocoding.
- Reasonable and unreasonable ways to utilize AI in this course.
- How abstraction will play a role in your future work in this course.
- The basic building blocks of programming, including functions, conditionals, loops, and variables.
- How to build a project in Scratch.

See you next time!

Lecture 1-C programming

- [Welcome!](#)
- [Hello World](#)
- [Functions](#)
- [Variables](#)
- [Conditionals](#)

- [Loops](#)
- [Operators and Abstraction](#)
- [Linux and the Command Line](#)
- [Mario](#)
- [Comments](#)
- [Types](#)
- [Summing Up](#)

Welcome!

- In our previous session, we learned about Scratch, a visual programming language.
- Indeed, all the essential programming concepts presented in Scratch will be utilized as you learn how to program any programming language.
- Recall that machines only understand binary. Where humans write *source code*, a list of instructions for the computer that is human readable, machines only understand what we can now call *machine code*. This machine code is a pattern of ones and zeros that produces a desired effect.
- It turns out that we can convert *source code* into `machine code` using a very special piece of software called a *compiler*. Today, we will be introducing you to a compiler that will allow you to convert source code in the programming language *C* into machine code.
- Today, in addition to learning about how to code, you will be learning about how to write good code.
- Code can be evaluated upon three axes. First, *correctness* refers to “does the code run as intended?” Second, *design* refers to “how well is the code designed?” Finally, *style* refers to “how aesthetically pleasing and consistent is the code?”

Hello World

- The integrated development environment (IDE) that is utilized for this course is *Visual Studio Code*, affectionately referred to as `VS Code`, which can be accessed via that same url, or simply as `*VS Code.*`
- One of the most important reasons we utilize VS Code is that it has all the software required for the course already pre-loaded on it. This course and the instructions herein were designed with VS Code in mind.
- Manually installing the necessary software for the course on your own computer is a cumbersome headache. Best always to utilize VS Code for assignments in this course.
- You can open VS Code at cs50.dev.
- The compiler can be divided into a number of regions:

The screenshot shows the Microsoft CodeSpace interface. On the left is the Explorer sidebar with icons for files, search, and other tools. The main area has three tabs: 'hello.c' (selected), 'hello.h', and 'hello.cpp'. The 'hello.c' tab contains the following C code:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

Below the code editor is the 'TERMINAL' section, which currently displays the command '\$ make hello'.

Notice that there is a *file explorer* on the left side where you can find your files. Further, notice that there is a region in the middle called a *text editor* where you can edit your program. Finally, there is a `command line interface`, known as a *CLI*, *command line*, or *terminal window* where we can send commands to the computer in the cloud.

- We will be using three commands to write, compile, and run our first program:

```
1 code hello.c
2
3 make hello
4
5 ./hello
6
```

The first command, `code hello.c` creates a file and allows us to type instructions for this program. The second command, `make hello`, compiles the file from our instructions in C and creates an executable file called `hello`. The last command, `./hello`, runs the program called `hello`.

- We can build your first program in C by typing `code hello.c` into the terminal window. Notice that we deliberately lowercased the entire filename and included the `.c` extension. Then, in the text editor that appears, write code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

Note that every single character above serves a purpose. If you type it incorrectly, the program will not run. `printf` is a function that can output a line of text. Notice the placement of the quotes and the semicolon. Further, notice that the `\n` creates a new line after the words `hello, world`.

- Clicking back in the terminal window, you can compile your code by executing `make hello`. Notice that we are omitting `.c`. `make` is a compiler that will look for our `hello.c` file and turn it into a program called `hello`. If executing this command results in no errors, you can proceed. If not, double-check your code to ensure it matches the above.
- Now, type `./hello` and your program will execute saying `hello, world`.
- Now, open the *file explorer* on the left. You will notice that there is now both a file called `hello.c` and another file called `hello`. `hello.c` is able to be read by the compiler: It's where your code is stored. `hello` is an executable file that you can run, but cannot be read by the compiler.

Functions

- In Scratch, we utilized the `say` block to display any text on the screen. Indeed, in C, we have a function called `printf` that does exactly this.
- Notice our code already invokes this function:

```
1 | printf("hello, world\n");
```

Notice that the `printf` function is called. The argument passed to `printf` is 'hello, world\n'. The statement of code is closed with a `;`.

- A common error in C programming is the omission of a semicolon. Modify your code as follows:

```
1 | #include <stdio.h>
2 |
3 | int main(void)
4 | {
5 |     printf("hello, world\n")
6 | }
```

Notice the semicolon is now gone.

- In your terminal window, run `make hello`. You will now be met with numerous errors! Placing the semicolon back in the correct position and running `make hello` again, the errors go away.
- Notice also the special symbol `\n` in your code. Try removing those characters and *making* your program again by executing `make hello`. Typing `./hello` in the terminal window, how did your program change? This `\` character is called an *escape character* that tells the compiler that `\n` is a special instruction.
- Restore your program to the following:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

Notice the semicolon and `\n` have been restored.

- The statement at the start of the code `#include <stdio.h>` is a very special command that tells the compiler that you want to use the capabilities of a *library* called `stdio.h`, a *header file*. This allows you, among many other things, to utilize the `printf` function. You can read about all the capabilities of this library on the [Manual Pages](#). The Manual Pages provide a means by which to better understand what various commands do and how they function.
- Libraries are collections of pre-written functions that others have written in the past that we can utilize in our code.
- It turns out that CS50 has its own library called `cs50.h`. Let's use this library in your program.

Variables

- Recall that in Scratch, we had the ability to ask the user "What's your name?" and say "hello" with that name appended to it.
- In C, we can do the same. Modify your code as follows:

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

The `get_string` function is used to get a string from the user. Then, the variable `answer` is passed to the `printf` function. `%s` tells the `printf` function to prepare itself to receive a `string`.

- `answer` is a special holding place we call a *variable*. `answer` is of type `string` and can hold any string within it. There are many *data types*, such as `int`, `bool`, `char`, and many others.
- `%s` is a placeholder called a *format code* that tells the `printf` function to prepare to receive a `string`. `answer` is the `string` being passed to `%s`.
- Running `make hello` again in the terminal window, notice that numerous errors appear.
- Looking at the errors `string` and `get_string` are not recognized by the compiler. We have to teach the compiler these features by adding a library called `cs50.h`:

```
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 }
```

Notice that `#include <cs50.h>` has been added to the top of your code.

- Now running `make hello` again in the terminal window, you can run your program by typing `./hello`. The program now asks for your name and then says hello with your name attached, as intended.
- `printf` allows for many format codes. Here is a noncomprehensive list of ones you may utilize in this course:
`%s` is used for `string` variables. `%i` is used for `int` or integer variables. You can find out more about this on the [Manual Pages](#)

Conditionals

- Another building block you utilized within Scratch was that of *conditionals*. For example, you might want to do one thing if `x` is greater than `y`. Further, you might want to do something else if that condition is not met.
- We look at a few examples from Scratch.

- In C, you can assign a value to an `int` or integer as follows:

Notice how a variable called `counter` of type `int` is assigned the value `0`.

- C can also be programmed to add one to `counter` as follows:

Notice how `1` is added to the value of `counter`.

- This can be represented also as:

Notice how `1` is added to the value of `counter`. However the `++` is used instead of `counter + 1`.

- You can also subtract one from `counter` as follows:

Notice how `1` is removed to the value of `counter`.

- Using this new knowledge about how to assign values to variables, you can program your first conditional statement.

- In the terminal window, type `code compare.c` and write code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
{
5     int x = get_int("What's x? ");
6     int y = get_int("What's y? ");
7
8     if (x < y)
9     {
10         printf("x is less than y\n");
11     }
12 }
13 }
```

Notice that we create two variables, an `int` or integer called `x` and another called `y`. The values of these are populated using the `get_int` function.

- You can run your code by executing `make compare` in the terminal window, followed by `./compare`. If you get any error messages, check your code for errors.
- *Flow charts* are a way by which you can examine how a computer program functions. Such charts can be used to examine the efficiency of our code.
- Looking at a flow chart of the above code, we can notice numerous shortcomings.
- We can improve your program by coding as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
{
5     int x = get_int("What's x? ");
6     int y = get_int("What's y? ");
7
8     if (x < y)
9     {
10         printf("x is less than y\n");
11     }
12     else if (x > y)
13     {
14         printf("x is greater than y\n");
15     }
16     else
17     {
18         printf("x is equal to y\n");
19     }
20 }
21 }
```

Notice that all potential outcomes are now accounted for.

- You can re-make and re-run your program and test it out.

- Examining this program on a flow chart, you can see the efficiency of our code design decisions.
- Considering another data type called a `char` we can start a new program by typing `code agree.c` into the terminal window.
- Where a `string` is a series of characters, a `char` is a single character.
- In the text editor, write code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Prompt user to agree
7     char c = get_char("Do you agree? ");
8
9     // Check whether agreed
10    if (c == 'Y' || c == 'y')
11    {
12        printf("Agreed.\n");
13    }
14    else if (c == 'N' || c == 'n')
15    {
16        printf("Not agreed.\n");
17    }
18 }
```

Notice that single quotes are utilized for single characters. Further, notice that `==` ensure that something *is equal* to something else, where a single equal sign would have a very different function in C. Finally, notice that `||` effectively means *or*.

- You can test your code by typing `make agree` into the terminal window, followed by `./agree`.

Loops

- We can also utilize the loops building block from Scratch in our C programs.
- We look at a few examples from Scratch. Consider the following code:

```

1 int counter = 3;
2 while (counter > 0)
3 {
4     printf("meow\n");
5     counter = counter - 1;
6 }
```

Notice that his code assigns the value of `3` to the `counter` variable. Then, the `while` loop says `meow` and removes one from the counter for each iteration. Once the counter is not greater than zero, the loop ends.

- In your terminal window, type `code meow.c` and write code as follows:

```

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

Notice this does as intended but has an opportunity for better design.

- We can improve our program by modifying your code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i = 3;
6     while (i > 0)
7     {
8         printf("meow\n");
9         i--;
10    }
11 }
```

Notice that we create an `int` called `i` and assign it the value `3`. Then, we create a `while` loop that will run as long as `i > 0`. Then, the loop runs. Every time `1` is subtracted to `i` using the `i--` statement.

- Similarly, we can implement a count-up of sorts by modifying our code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i = 1;
6     while (i <= 3)
7     {
8         printf("meow\n");
9         i++;
10    }
11 }
```

Notice how our counter `i` is started at `1`. Each time the loop runs, it will increment the counter by `1`. Once the counter is greater than `3`, it will stop the loop.

- Generally, in computer science we count from zero. Best to revise your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i = 0;
6     while (i < 3)
7     {
8         printf("meow\n");
9         i++;
10    }
11 }
```

Notice we now count from zero.

- Another tool in our toolbox for looping is a `for` loop.
- You can further improve the design of our `meow.c` program using a `for` loop. Modify your code as follows:

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

Notice that the `for` loop includes three arguments. The first argument `int i = 0` starts our counter at zero. The second argument `i < 3` is the condition that is being checked. Finally, the argument `i++` tells the loop to increment by one each time the loop runs.

- We can even loop forever using the following code:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     while (true)
7     {
8         printf("meow\n");
9     }
10 }
```

Notice that `true` will always be the case. Therefore, the code will always run. You will lose control of your terminal window by running this code. You can break from an infinite by hitting `control-C` on your keyboard.

- While we will provide much more guidance later, you can create your own function within C as follows:

```
1 void meow(void)
2 {
3     printf("meow\n");
4 }
```

The initial `void` means that the function does not return any values. The `(void)` means that no values are being provided to the function.

- This function can be used in the main function as follows:

```
1 #include <stdio.h>
2
3 void meow(void);
4
5 int main(void)
6 {
7     for (int i = 0; i < 3; i++)
8     {
9         meow();
10    }
11 }
12
13 void meow(void)
14 {
15     printf("meow\n");
16 }
```

Notice how the `meow` function is called with the `meow()` instruction. This is possible because the `meow` function is defined at the bottom of the code and the *prototype* of the function is provided at the top of the code as `void meow(void)`.

- Your `meow` function can be further modified to accept input:

```
1 #include <stdio.h>
2
3 void meow(int n);
4
5 int main(void)
6 {
7     meow(3);
8 }
9
10 // Meow some number of times
11 void meow(int n)
12 {
13     for (int i = 0; i < n; i++)
14     {
15         printf("meow\n");
16     }
17 }
```

Notice that the prototype has changed to `void meow(int n)` to show that `meow` accepts an `int` as its input.

Operators and Abstraction

- You can implement a calculator in C. In your terminal, type `code calculator.c` and write code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Prompt user for x
7     int x = get_int("x: ");
8
9     // Prompt user for y
10    int y = get_int("y: ");
11
12    // Perform addition
13    printf("%i\n", x + y);
14 }
```

Notice how the `get_int` function is utilized to obtain an integer from the user twice. One integer is stored in the `int` variable called `x`. Another is stored in the `int` variable called `y`. Then, the `printf` function prints the value of `x + y`, designated by the `%i` symbol.

- *Operators* refer to the mathematical operations that are supported by your compiler. In C, these mathematical operators include:
 - `+` for addition
 - `-` for subtraction
 - `*` for multiplication
 - `/` for division
 - `%` for remainder
- *Abstraction* is the art of simplifying our code such that it deals with smaller and smaller problems.
- Expanding on our previously acquired knowledge about functions, we could *abstract away* the addition into a function. Modify your code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int add(int a, int b);
5
6 int main(void)
7 {
8     // Prompt user for x
9     int x = get_int("x: ");
```

```

10
11     // Prompt user for y
12     int y = get_int("y: ");
13
14     // Perform addition
15     int z = add(x, y);
16     printf("%i\n", z);
17 }
18
19 int add(int a, int b)
20 {
21     int c = a + b;
22     return c;
23 }
```

Notice that the `add` function takes two variables as its input. These values are assigned to `a` and `b` and performs a calculation, returning the value of `c`. Further, notice that the *scope* (or context in which variables exist) of `x` is the `main` function. The variable `c` is only within the scope of the `add` function.

- The design of this program can be further improved as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int add(int a, int b);
5
6 int main(void)
7 {
8     // Prompt user for x
9     int x = get_int("x: ");
10
11    // Prompt user for y
12    int y = get_int("y: ");
13
14    // Perform addition
15    printf("%i\n", add(x, y));
16 }
17
18 int add(int a, int b)
19 {
20     return a + b;
21 }
```

Notice that `c` in the `add` function is removed entirely.

- While very useful to be able to abstract away to an `add` function, you can also perform addition through *truncation* as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
```

```

4 int main(void)
5 {
6     // Prompt user for x
7     long x = get_long("x: ");
8
9     // Prompt user for y
10    long y = get_long("y: ");
11
12    // Perform addition
13    printf("%li\n", x + y);
14 }
```

Notice that the addition is performed within the `printf` function.

- Similarly, division can be performed as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Prompt user for x
7     int x = get_int("x: ");
8
9     // Prompt user for y
10    int y = get_int("y: ");
11
12    // Divide x by y
13    printf("%i\n", x / y);
14 }
```

Notice that division is performed within the `printf` function.

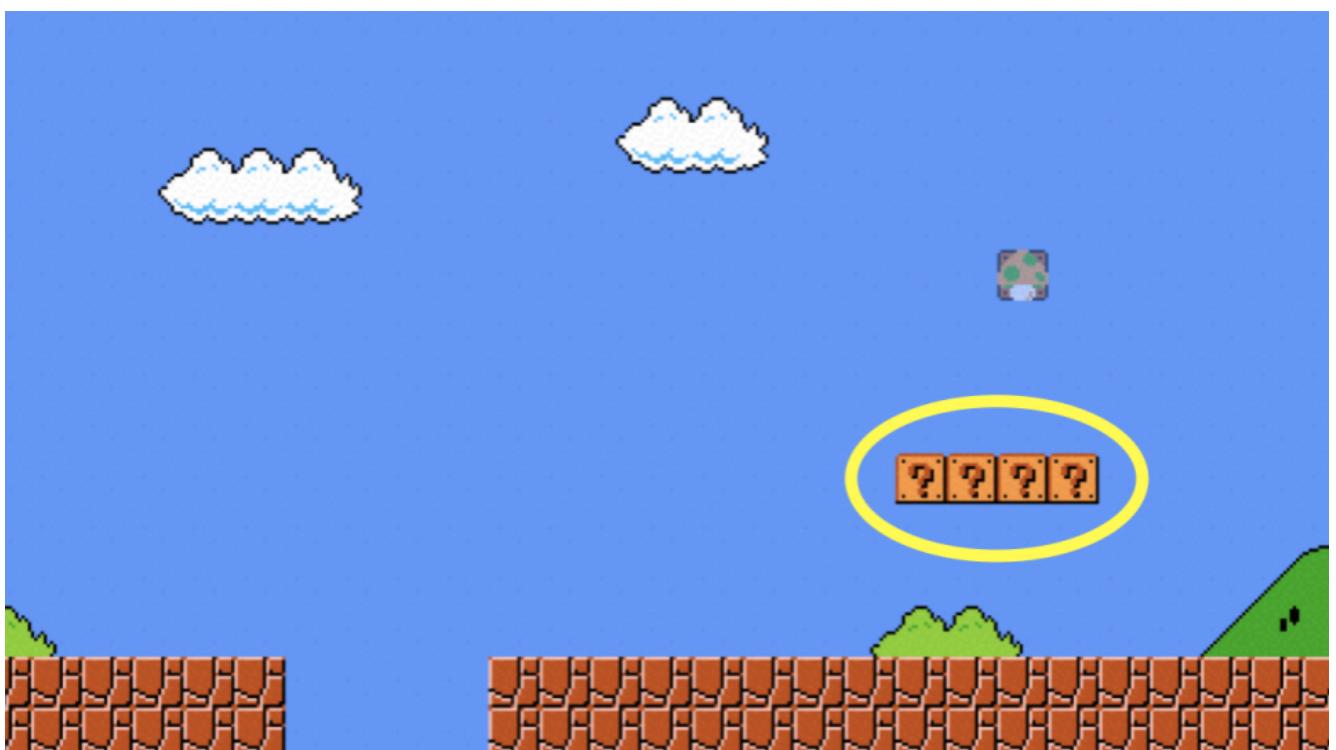
Linux and the Command Line

- Linux* is an operating system that is accessible via the command line in the terminal window in VS Code.
- Some common command-line arguments we may use include:
 - `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
- The most commonly used is `ls` which will list all the files in the current directory or directory. Go ahead and type `ls` into the terminal window and hit `enter`. You'll see all the files in the current folder.

- Another useful command is `mv`, where you can move a file from one file to another. For example, you could use this command to rename `Hello.c` (notice the uppercase `H`) to `hello.c` by typing `mv Hello.c hello.c`.
- You can also create folders. You can type `mkdir pset1` to create a directory called `pset1`.
- You can then use `cd pset1` to change your current directory to `pset1`.

Mario

- Everything we've discussed today has focused on various building-blocks of your work as an emerging computer scientist.
- The following will help you orient toward working on a problem set for this class in general: How does one approach a computer science related problem?
- Imagine we wanted to emulate the visual of the game Super Mario Bros. Considering the four question-blocks pictured, how could we create code that roughly represents these four horizontal blocks?



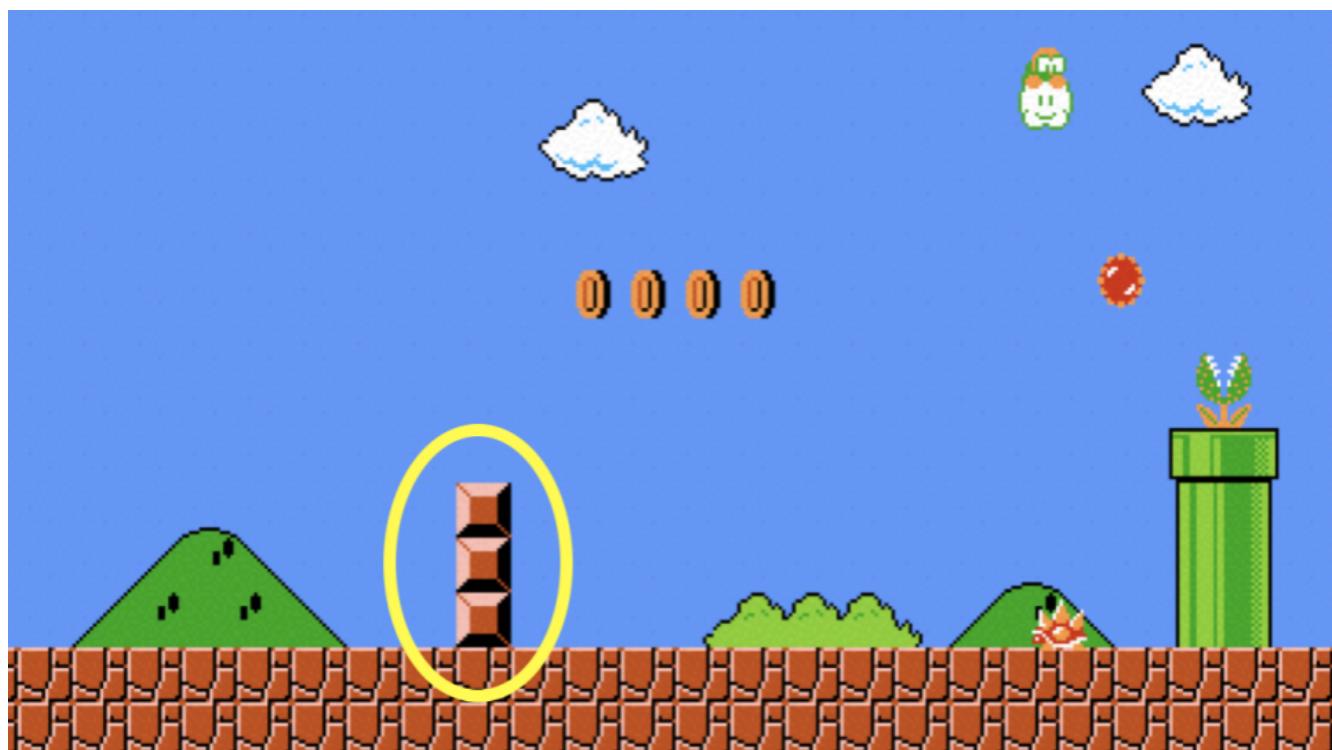
- In the terminal window, type `code mario.c` and code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     for (int i = 0; i < 4; i++)
6     {
7         printf "?";
8     }
9     printf "\n";
10 }
```

Notice how four question marks are printed here using a loop.

- Similarly, we can apply this same logic to be able to create three vertical blocks.

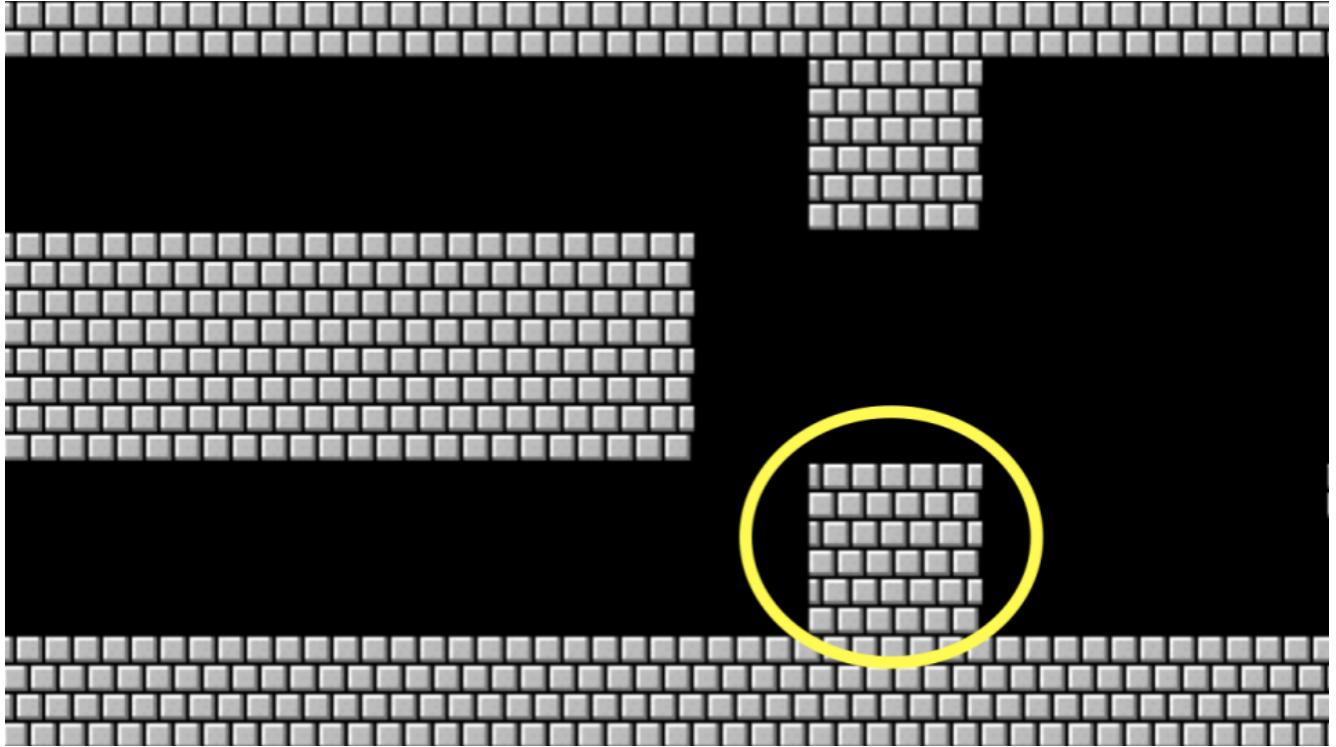


- To accomplish this, modify your code as follows:

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

Notice how three vertical bricks are printed using a loop.

- What if we wanted to combine these ideas to create a three-by-three group of blocks?



- We can follow the logic above, combining the same ideas. Modify your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for (int i = 0; i < 3; i++)
6     {
7         for (int j = 0; j < 3; j++)
8         {
9             printf("#");
10        }
11        printf("\n");
12    }
13 }
```

Notice that one loop is inside another. The first loop defines what vertical row is being printed. For each row, three columns are printed. After each row, a new line is printed.

- What if we wanted to ensure that the number of blocks to be *constant*, that is, unchangeable? Modify your code as follows:

```

1 int main(void)
2 {
3     const int n = 3;
4     for (int i = 0; i < n; i++)
5     {
6         for (int j = 0; j < n; j++)
7         {
8             printf("#");
9         }
10        printf("\n");
11    }
12 }
```

Notice how `n` is now a constant. It can never be changed.

- As illustrated earlier in this lecture, we can make our code prompt the user for the size of the grid. Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int n = get_int("Size: ");
7
8     for (int i = 0; i < n; i++)
9     {
10         for (int j = 0; j < n; j++)
11         {
12             printf("#");
13         }
14         printf("\n");
15     }
16 }
```

Notice that `get_int` is used to prompt the user.

- A general piece of advice within programming is that you should never fully trust your user. They will likely misbehave, typing incorrect values where they should not. We can protect our program from bad behavior by checking to make sure the user's input satisfies our needs. Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int n;
7     do
8     {
9         n = get_int("Size: ");
10    }
```

```

11  while (n < 1);
12
13  for (int i = 0; i < n; i++)
14  {
15      for (int j = 0; j < n; j++)
16      {
17          printf("#");
18      }
19      printf("\n");
20  }
21 }
```

Notice how the user is continuously prompted for the size until the user's input is 1 or greater.

Comments

- Comments are fundamental parts of a computer program, where you leave explanatory remarks to yourself and others that may be collaborating with you regarding your code.
- All code you create for this course must include robust comments.
- Typically each comment is a few words or more, providing the reader an opportunity to understand what is happening in a specific block of code. Further, such comments serve as a reminder for you later when you need to revise your code.
- Comments involve placing `//` into your code, followed by a comment. Modify your code as follows to integrate comments:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Prompt user for positive integer
7     int n;
8     do
9     {
10         n = get_int("Size: ");
11     }
12     while (n < 1);
13
14     // Print an n-by-n grid of bricks
15     for (int i = 0; i < n; i++)
16     {
17         for (int j = 0; j < n; j++)
18         {
19             printf("#");
20         }
21         printf("\n");
22     }
23 }
```

Notice how each comment begins with a `//`.

Types

- One of C's shortcomings is the ease by which it managing memory. While C provides you immense control over how memory is utilized, programmers have to be very aware of potential pitfalls of memory management.
- Types refer to the possible data that can be stored within a variable. For example, a `char` is designed to accommodate a single character like `a` or `2`.
- Types are very important because each type has specific limits. For example, because of the limits in memory, the highest value of an `int` can be `4294967295`. If you attempt to count an `int` higher, *integer overflow* will result where an incorrect value will be stored in this variable.
- The number of bits limits how high and low we can count.
- Types with which you might interact during this course include:
 - `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
- As you are coding, pay special attention to the types of variables you are using to avoid problems within your code.
- We examined some examples of disasters that can occur through memory-related errors.

Summing Up

In this lesson, you learned how to apply the building blocks you learned in Scratch to the C programming language. You learned...

- How to create your first program in C.
- Predefined functions that come natively with C and how to implement your own functions.
- How to use variables, conditionals, and loops.
- How to approach abstraction to simplify and improve your code.
- How to approach problem-solving for a computer science problem.
- How to use the Linux command line.
- How to integrate comments into your code.
- How to utilize types and operators.

See you next time!

-

Lecture 2-Arrays

- [Welcome!](#)
- [Compiling](#)
- [Debugging](#)
- [Arrays](#)
- [Strings](#)
- [String Length](#)
- [Command-Line Arguments](#)
- [Exit Status](#)
- [Cryptography](#)
- [Summing Up](#)

Welcome!

- In our previous session, we learned about C, a text-based programming language.
- This week, we are going to take a deeper look at additional building-blocks that will support our goals of learning more about programming from the bottom up.
- Fundamentally, in addition to the essentials of programming, this course is about problem-solving. Accordingly, we will also focus further on how to approach computer science problems.

Compiling

- *Encryption* is the act of hiding plain text from prying eyes. *decrypting*, then, is the act of taking an encrypted piece of text and returning it to a human-readable form.
- An encrypted piece of text may look like the following:

U I J T X B T D T 5 0

- Recall that last week you learned about a *compiler*, a specialized computer program that converts *source code* into *machine code* that can be understood by a computer.
- For example, you might have a computer program that looks like this:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world\n");
6 }
```

- A compiler will take the above code and turn it into the following machine code:

```
01111111 01000101 01001100 01000110 00000010 00000001 00000001 00000000  
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
00000010 00000000 00111110 00000000 00000001 00000000 00000000 00000000  
10110000 00000101 01000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
11010000 00010011 00000000 00000000 00000000 00000000 00000000 00000000  
00000000 00000000 00000000 00000000 01000000 00000000 00111000 00000000  
00001001 00000000 01000000 00000000 00100100 00000000 00100001 00000000  
00000110 00000000 00000000 00000000 00000101 00000000 00000000 00000000  
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000  
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000  
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000  
00001000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
00000011 00000000 00000000 00000000 00000100 00000000 00000000 00000000  
00111000 00000010 00000000 00000000 00000000 00000000 00000000 00000000  
...
```

- VS Code, the programming environment provided to you as a CS50 student, utilizes a compiler called `clang` or `c language`.
- If you were to type `make hello`, it runs a command that executes clang to create an output file that you can run as a user.
- VS Code has been pre-programmed such that `make` will run numerous command line arguments along with clang for your convenience as a user.
- Consider the following code:

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

- You can attempt to enter into the terminal window: `clang -o hello hello.c`. You will be met by an error that indicates that clang does not know where to find the `cs50.h` library.
- Attempting again to compile this code, run the following command in the terminal window: `clang -o hello hello.c -lcs50`. This will enable the compiler to access the `cs50.h` library.
- Running in the terminal window `./hello`, your program will run as intended.
- While the above is offered as an illustration, such that you can understand more deeply the process and concept of compiling code, using `make` in CS50 is perfectly fine and the expectation!
- Compiling involves major steps, including the following:

- First, *preprocessing* is where the header files in your code, designated by a `#` (such as `#include <cs50.h>`) are effectively copied and pasted into your file. During this step, the code from `cs50.h` is copied into your program. Similarly, just as your code contains `#include <stdio.h>`, code contained within `stdio.h` somewhere on your computer is copied to your program. This step can be visualized as follows:

```

1 string get_string(string prompt);
2 int printf(string format, ...);
3
4 int main(void)
5 {
6     string name = get_string("What's your name? ");
7     printf("hello, %s\n", name);
8 }
```

- Second, *compiling* is where your program is converted into assembly code. This step can be visualized as follows:

```

...
main:                                # @main
    .cfi_startproc
# BB#0:
    pushq    %rbp
.Ltmp0:
    .cfi_offset %rbp, 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
...
```

- Third, *assembling* involves the compiler converting your assembly code into machine code. This step can be visualized as follows:

```

0111111010001010100110001000110
000000100000001000000010000000
00000000000000000000000000000000
00000000000000000000000000000000
000000010000000001111000000000
00000001000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10100000000001000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
00001010000000000000000000000000
0101010101001000100111100101
01001000100000111110110000010000
00110001110000001000100111000111
01001000101111000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
01110100000000000000000000000000
00000000001001000101111100000000
00000000000000000000000000000000
00000000000000000000000000000000
...

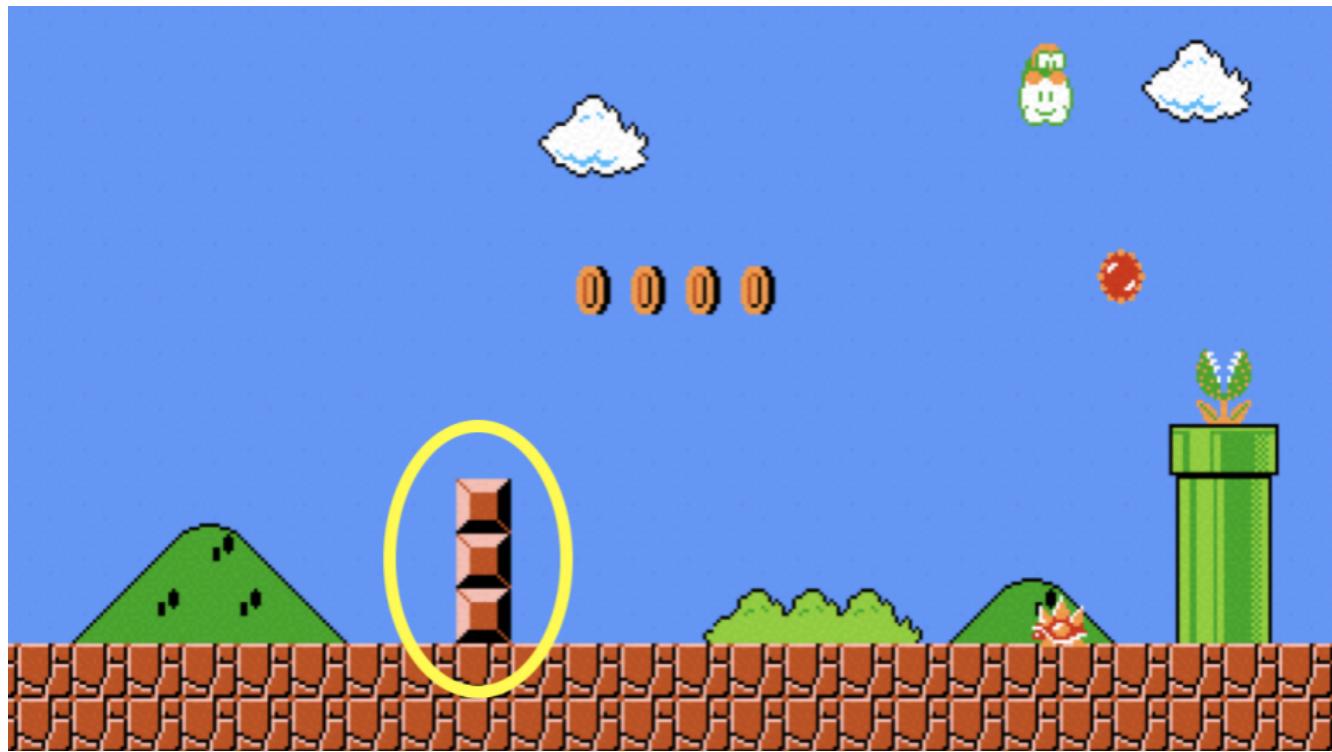
```

- Finally, during the *linking* step, code from your included libraries are converted also into machine code and combined with your code. The final executable file is then outputted.

0111111010001010100110001000110	0111111010001010100110001000110	00101111011011000110100101100010
000000100000001000000010000000	000000100000001000000010000000	0110001100101100111001101011111
00000000000000000000000000000000	00000000000000000000000000000000	00101110001101000100000001011111
00000000000000000000000000000000	00000000000000000000000000000000	01110101011100110111001000101111
00000001000000000111110000000000	00000001100000000111110000000000	0110110000110100010110001000101111
00000001000000000000000000000000	00000001000000000000000000000000	01111000001100000110100010111111
00000000000000000000000000000000	11000000000000000000000000000000	00110110001101000101101011011100
00000000000000000000000000000000	00000000000000000000000000000000	01101001011011001110101011111000
00000000000000000000000000000000	01000000000000000000000000000000	00101101011001100111010111001101
00000000000000000000000000000000	00000000000000000000000000000000	001011110110110001101000101100010
10100000000001000000000000000000	00101000001100100000000000000000	01100011010111101101110011011111
00000000000000000000000000000000	00000000000000000000000000000000	01101110011100110110100001100001
00000000000000000000000000000000	00000000000000000000000000000000	011100001001100101010001000010110
01000000000000000000000000000000	01000000000000000000000000000000	010100010010000001000000010000001
00000000000000000000000000000000	00000000000000000000000000000000	01010011010111101001110010001001
00000000000000000000000000000000	00000000000000000000000000000000	0100010100010001000100010001000100
00001010000000000000000000000000	00000000000000000000000000000000	010001010100010001000100010001000
0101010101001000100111100101	00000000000000000000000000000000	001000000010100000000000000000001
01001000100000111110110000010000	00000000000000000000000000000000	0110110000110100010110001000101111
00110001110000001000100111000111	00000000000000000000000000000000	01111000001100000110110010111111
01001000101111100000000000000000	00000000000000000000000000000000	00110110001101000101100010111100
00000000000000000000000000000000	00000000000000000000000000000000	01101000101111001101110011101010
11101000000000000000000000000000	00000000000000000000000000000000	0010111101101100011001000101101
00000000001001000101111100000000	00000000000000000000000000000000	011011000011010001011011001110101
00000000000000000000000000000000	01011100000000000000000000000000	01111000001011000101101100111000
00000000000000000000000000000000	00000000000000000000000000000000	001101100001011010001011011001100
00000000000000000000000000000000	00000000000000000000000000000000	001101100001011010001011011001100
...

Debugging

- Everyone will make mistakes while coding.
- Consider the following image from last week:



- Further, consider the following code that has a bug purposely inserted within it:

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

- Type `code buggy0.c` into the terminal window and write the above code.
- Running this code, four bricks appear instead of the intended three.
- `printf` is a very useful way of debugging your code. You could modify your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for (int i = 0; i <= 3; i++)
6     {
7         printf("i is %i\n", i);
8         printf("#\n");
9     }
10 }
```

- Running this code, you will see numerous statements, including `i is 0`, `i is 1`, `i is 2`, and `i is 3`. Seeing this, you might realize that further code needs to be corrected as follows:

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

Notice the `<=` has been replaced with `<`.

- This code can be further improved as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 void print_column(int height);
5
6 int main(void)
7 {
8     int h = get_int("Height: ");
9     print_column(h);
10 }
11
12 void print_column(int height)
13 {
14     for (int i = 0; i <= height; i++)
15     {
16         printf("#\n");
17     }
18 }
```

Notice that compiling and running this code still results in a bug.

- To address this bug, we will use a new tool at our disposal.
- A second tool in debugging is called a *debugger*, a software tool created by programmers to help track down bugs in code.
- In VS Code, a preconfigured debugger has been provided to you.
- To utilize this debugger, first set a *breakpoint* by clicking to the left of a line of your code, just to the left of the line number. When you click there, you will see a red dot appearing. Imagine this as a stop sign, asking the compiler to pause such that you can consider what's happening in this part of your code.

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 void print_column(int height);
5
● 6 int main(void)
  ↗ 7 {
8     int h = get_int("Height: ");
9     print_column(h);
10 }
11
12 void print_column(int height)
13 {
14     for (int i = 0; i <= height; i++)
```

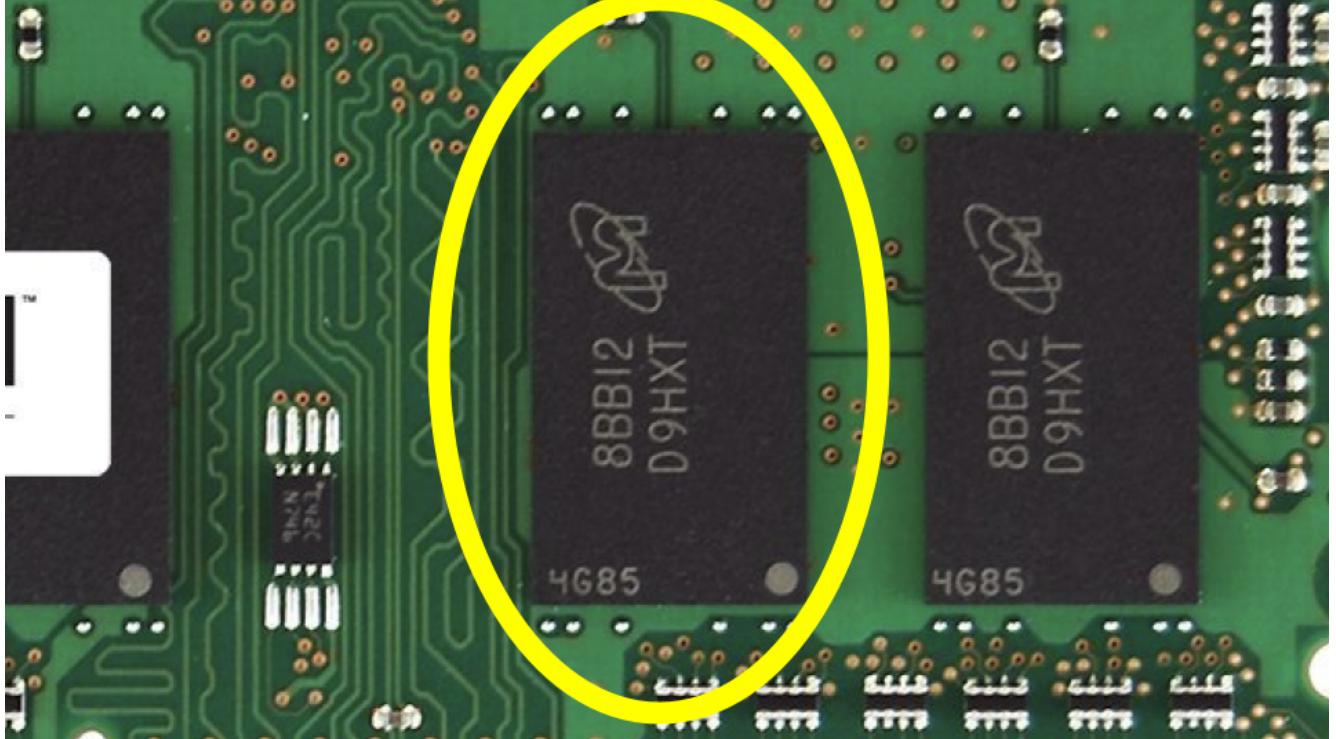
- Second, run `debug50 ./buggy0`. You will notice that after the debugger comes to life that a line of your code will illuminate in a gold-like color. Quite literally, the code has *paused* at this line of code. Notice in the top left corner how all local variables are being displayed, including `h`, which has a current value of `0`. At the top of your window, you can click the `step over` button and it will keep moving through your code. Notice how the value of `h` increases.
- While this tool will not show you where your bug is, it will help you slow down and see how your code is running step by step. You can use `step into` as a way to look further into the details of your buggy code.
- A final form of debugging is called *rubber duck debugging*. When you are having challenges with your code, consider how speaking out loud to, quite literally, a rubber duck about the code problem. If you'd rather not talk to a small plastic duck, you are welcome to speak to a human near you! They need not understand how to program: Speaking with them is an opportunity for you to speak about your code.

Arrays

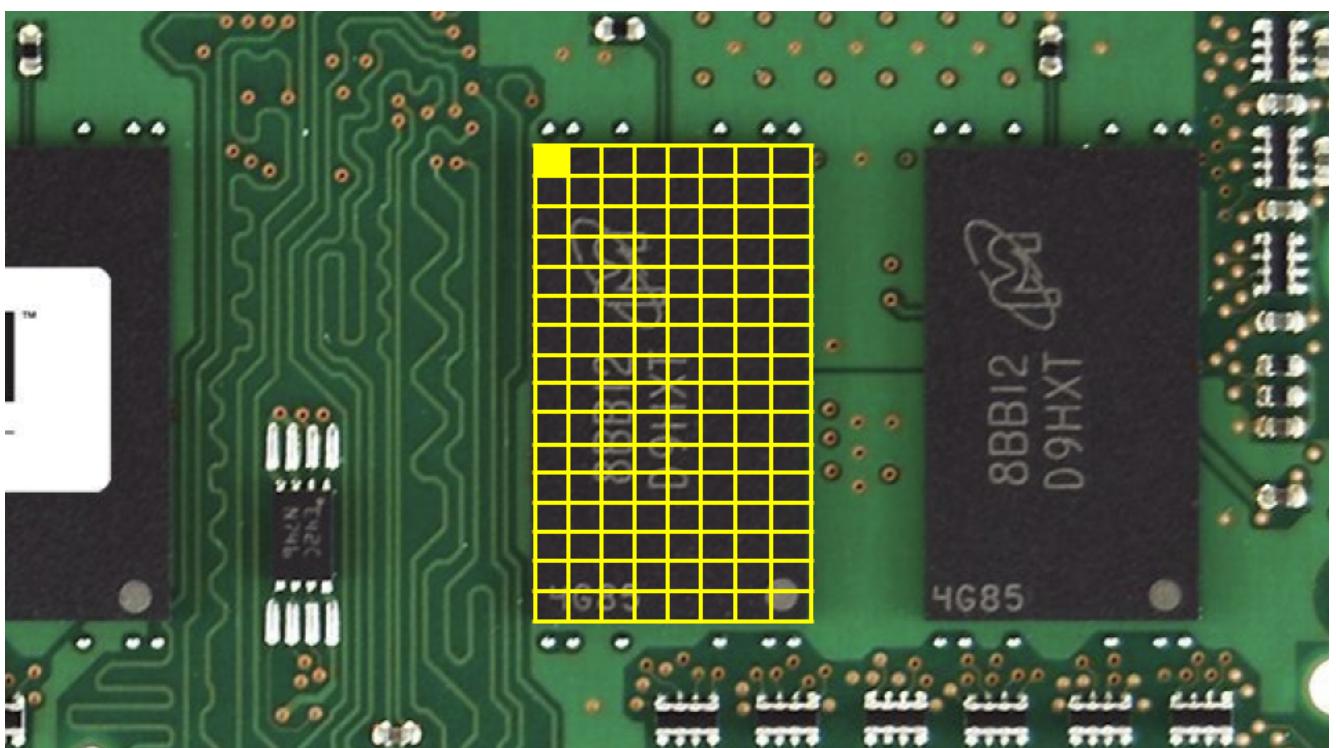
- In Week 0, we talked about *data types* such as `bool`, `int`, `char`, `string`, etc.
- Each data type requires a certain amount of system resources:
 - `bool` 1 byte
 - `int` 4 bytes

- `long` 8 bytes
- `float` 4 bytes
- `double` 8 bytes
- `char` 1 byte
- `string` ? bytes

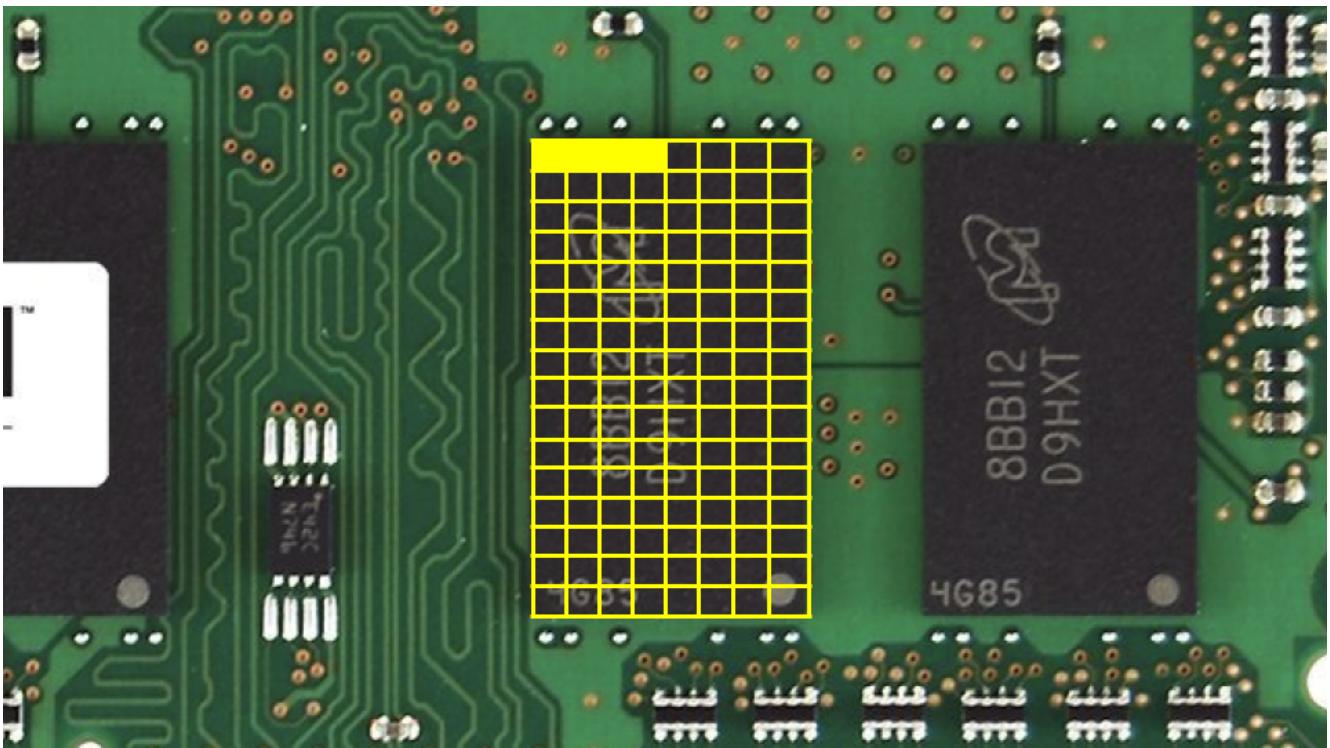
- Inside of your computer, you have a finite amount of memory available.



- Physically, on the memory of your computer, you can imagine how specific types of data are stored on your computer. You might imagine that a `char`, which only requires 1 byte of memory, may look as follows:



- Similarly, an `int`, which requires 4 bytes might look as follows:



- We can create a program that explores these concepts. Inside your terminal, type `code scores.c` and write code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     // Scores
6     int score1 = 72;
7     int score2 = 73;
8     int score3 = 33;
9
10    // Print average
11    printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
12 }
```

Notice that the number on the right is a floating point value of `3.0`, such that the calculation is rendered as a floating point value in the end.

- Running `make scores`, the program runs.
- You can imagine how these variables are stored in memory:

72 score1	73 score2
33 score3	

- *Arrays* are a way of storing data back-to-back in memory such that this data is easily accessible.
- `int scores[3]` is a way of telling the compiler to provide you three back-to-back places in memory of size `int` to store three `scores`. Considering our program, you can revise your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get scores
7     int scores[3];
8     scores[0] = get_int("Score: ");
9     scores[1] = get_int("Score: ");
10    scores[2] = get_int("Score: ");
11
12    // Print average
13    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
14 }
```

Notice that `score[0]` examines the value at this location of memory by `indexing into` the array called `scores` at location `0` to see what value is stored there.

- You can see how while the above code works, there is still an opportunity for improving our code. Revise your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get scores
7 }
```

```

7   int scores[3];
8   for (int i = 0; i < 3; i++)
9   {
10     scores[i] = get_int("Score: ");
11   }
12
13   // Print average
14   printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
15 }
```

Notice how we index into `scores` by using `scores[i]` where `i` is supplied by the `for` loop.

- We can simplify or *abstract away* the calculation of the average. Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 // Constant
5 const int N = 3;
6
7 // Prototype
8 float average(int length, int array[]);
9
10 int main(void)
11 {
12     // Get scores
13     int scores[N];
14     for (int i = 0; i < N; i++)
15     {
16         scores[i] = get_int("Score: ");
17     }
18
19     // Print average
20     printf("Average: %f\n", average(N, scores));
21 }
22
23 float average(int length, int array[])
24 {
25     // Calculate average
26     int sum = 0;
27     for (int i = 0; i < length; i++)
28     {
29         sum += array[i];
30     }
31     return sum / (float) length;
32 }
```

Notice that a new function called `average` is declared. Further, notice that a `const` or constant value of `N` is declared. Most importantly, notice how the `average` function takes `int array[]`, which means that the compiler passes an array to this function.

- Not only can arrays be containers: They can be passed between functions.

Strings

- A `string` is simply an array of variables of type `char`: an array of characters.
- Considering the following image, you can see how a string is an array of characters that begins with the first character and ends with a special character called a `NUL character`:

H	I	!	\0				

- Imagining this in decimal, your array would look like the following:

72	73	33	0				

- Implementing this in your own code, type `code hi.c` in the terminal window and write code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c1 = 'H';
6     char c2 = 'I';
7     char c3 = '!';
8
9     printf("%c%c%c\n", c1, c2, c3);
10 }
```

Notice that this will output a string of characters.

- Similarly, make the following modification to your code:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c1 = 'H';
6     char c2 = 'I';
7     char c3 = '!';
8
9     printf("%i %i %i\n", c1, c2, c3);
10 }
```

Notice that that ASCII codes are printed by replacing `%c` with `%i`.

- To further understand how a `string` works, revise your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     string s = "HI!";
7     printf("%c%c%c\n", s[0], s[1], s[2]);
8 }
```

Notice how the `printf` statement presents three values from our array called `s`.

- As before, we can replace `%c` with `%i` as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     string s = "HI!";
7     printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);
8 }
```

Notice that this prints the string's ASCII codes, including NUL.

- Let's imagine we want to say both `HI!` and `BYE!`. Modify your code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     string s = "HI!";
7     string t = "BYE!";
8
9     printf("%s\n", s);
10    printf("%s\n", t);
11 }
```

Notice that two strings are declared and used in this example.

- You can visualize this as follow:

H s[0]	I s[1]	! s[2]	\0 s[3]	B t[0]	Y t[1]	E t[2]	! t[3]
\0 t[4]							

- We can further improve this code. Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     string words[2];
7
8     words[0] = "HI!";
9     words[1] = "BYE!";
10
11    printf("%s\n", words[0]);
12    printf("%s\n", words[1]);
13 }
```

Notice that both strings are stored within a single array of type `string`.

String Length

- A common problem within programming, and perhaps C more specifically, is to discover the length of an array. How could we implement this in code? Type `code_length.c` in the terminal window and code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Prompt for user's name
7     string name = get_string("Name: ");
8
9     // Count number of characters up until '\0' (aka NUL)
10    int n = 0;
11    while (name[n] != '\0')
12    {
13        n++;
14    }
15    printf("%i\n", n);
16 }
```

Notice that this code loops until the `NUL` character is found.

- This code can be improved by abstracting away the counting as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int string_length(string s);
5
6 int main(void)
7 {
```

```

8 // Prompt for user's name
9 string name = get_string("Name: ");
10 int length = string_length(name);
11 printf("%i\n", length);
12 }
13
14 int string_length(string s)
15 {
16     // Count number of characters up until '\0' (aka NUL)
17     int n = 0;
18     while (s[n] != '\0')
19     {
20         n++;
21     }
22     return n;
23 }
```

- Since this is such a common problem within programming, other programmers have created code in the `string.h` library to find the length of a string. You can find the length of a string by modifying your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     // Prompt for user's name
8     string name = get_string("Name: ");
9     int length = strlen(name);
10    printf("%i\n", length);
11 }
```

Notice that this code uses the `string.h` library, declared at the top of the file. Further, it uses a function from that library called `strlen`, which calculates the length of the string passed to it.

- `ctype.h` is another library that is quite useful. Imagine we wanted to create a program that converted all lowercase characters to uppercase ones. In the terminal window type `code uppercase.c` and write code as follows:

```

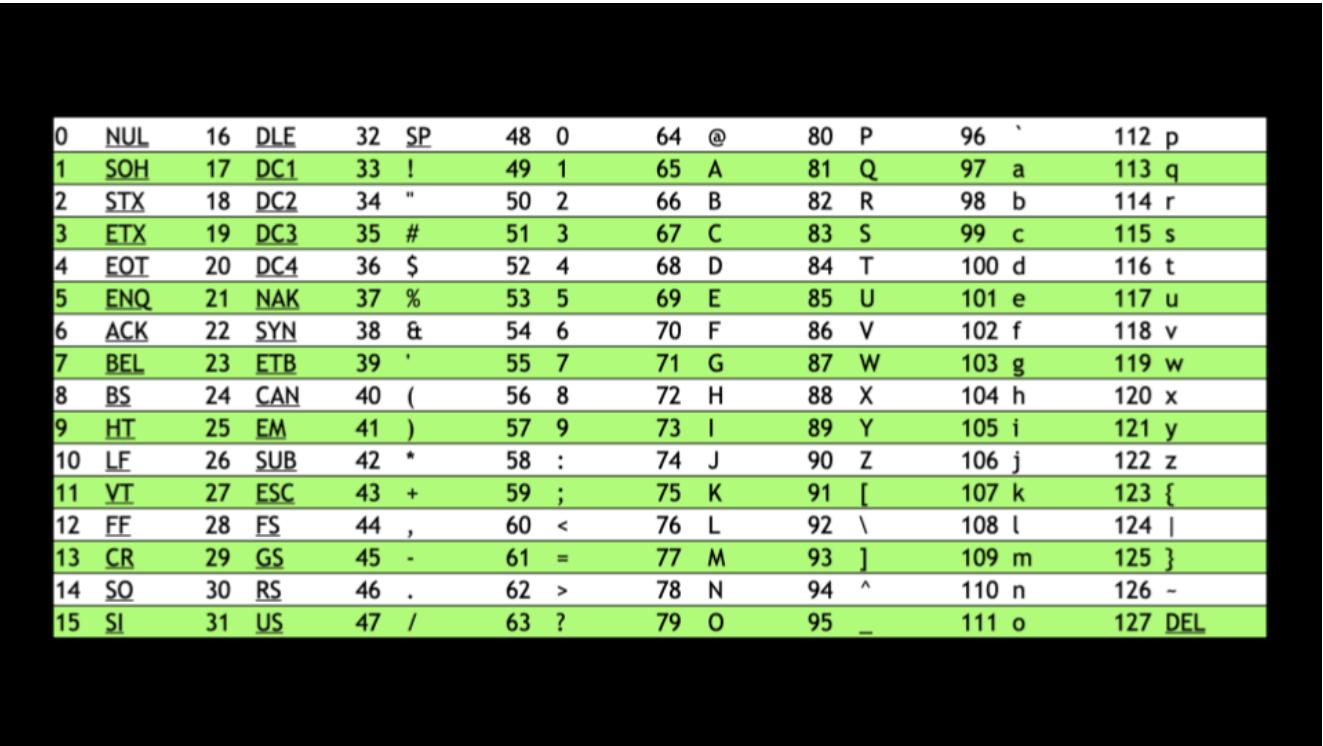
1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     string s = get_string("Before: ");
8     printf("After:  ");
9     for (int i = 0, n = strlen(s); i < n; i++)
10    {
11        if (s[i] >= 'a' && s[i] <= 'z')
```

```

12     {
13         printf("%c", s[i] - 32);
14     }
15     else
16     {
17         printf("%c", s[i]);
18     }
19 }
20 printf("\n");
21 }
```

Notice that this code *iterates* through each value in the string. The program looks at each character. If the character is lowercase, it subtracts the value 32 from it to convert it to uppercase.

- Recalling our previous work from last week, you might remember this ASCII values chart:



0	<u>NUL</u>	16	<u>DLE</u>	32	<u>SP</u>	48	0	64	@	80	P	96	'	112	p
1	<u>SOH</u>	17	<u>DC1</u>	33	!	49	1	65	A	81	Q	97	a	113	q
2	<u>STX</u>	18	<u>DC2</u>	34	"	50	2	66	B	82	R	98	b	114	r
3	<u>ETX</u>	19	<u>DC3</u>	35	#	51	3	67	C	83	S	99	c	115	s
4	<u>EOT</u>	20	<u>DC4</u>	36	\$	52	4	68	D	84	T	100	d	116	t
5	<u>ENQ</u>	21	<u>NAK</u>	37	%	53	5	69	E	85	U	101	e	117	u
6	<u>ACK</u>	22	<u>SYN</u>	38	&	54	6	70	F	86	V	102	f	118	v
7	<u>BEL</u>	23	<u>ETB</u>	39	'	55	7	71	G	87	W	103	g	119	w
8	<u>BS</u>	24	<u>CAN</u>	40	(56	8	72	H	88	X	104	h	120	x
9	<u>HT</u>	25	<u>EM</u>	41)	57	9	73	I	89	Y	105	i	121	y
10	<u>LF</u>	26	<u>SUB</u>	42	*	58	:	74	J	90	Z	106	j	122	z
11	<u>VT</u>	27	<u>ESC</u>	43	+	59	;	75	K	91	[107	k	123	{
12	<u>FF</u>	28	<u>FS</u>	44	,	60	<	76	L	92	\	108	l	124	
13	<u>CR</u>	29	<u>GS</u>	45	-	61	=	77	M	93]	109	m	125	}
14	<u>SO</u>	30	<u>RS</u>	46	.	62	>	78	N	94	^	110	n	126	-
15	<u>SI</u>	31	<u>US</u>	47	/	63	?	79	O	95	_	111	o	127	DEL

- When a lowercase character has `32` subtracted from it, it results in an uppercase version of that same character.
- While the program does what we want, there is an easier way using the `ctype.h` library. Modify your program as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void)
7 {
8     string s = get_string("Before: ");
9     printf("After: ");
10    for (int i = 0, n = strlen(s); i < n; i++)
11    {
```

```

12     if (islower(s[i]))
13     {
14         printf("%c", toupper(s[i]));
15     }
16     else
17     {
18         printf("%c", s[i]);
19     }
20 }
21 printf("\n");
22 }
```

Notice that the program iterates through each character of the string. The `toupper` function is passed `s[i]`. Each character (if lowercase) is converted to uppercase.

- It's worth mentioning that `toupper` automatically knows to uppercase only lowercase characters. Hence, your code can be simplified as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void)
7 {
8     string s = get_string("Before: ");
9     printf("After: ");
10    for (int i = 0, n = strlen(s); i < n; i++)
11    {
12        printf("%c", toupper(s[i]));
13    }
14    printf("\n");
15 }
```

Notice that this code uppercases a string using the `ctype` library.

- You can read about all the capabilities of the `ctype` library on the [Manual Pages](#).

Command-Line Arguments

- `Command-line arguments` are those arguments that are passed to your program at the command line. For example, all those statements you typed after `clang` are considered command line arguments. You can use these arguments in your own programs!
- In your terminal window, type `code greet.c` and write code as follows:

```
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 }
```

Notice that this says `hello` to the user.

- Still, would it not be nice to be able to take arguments before the program even runs? Modify your code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(int argc, string argv[])
5 {
6     if (argc == 2)
7     {
8         printf("hello, %s\n", argv[1]);
9     }
10    else
11    {
12        printf("hello, world\n");
13    }
14 }
```

Notice that this program knows both `argc`, the number of command line arguments, and `argv` which is an array of the characters passed as arguments at the command line.

- Therefore, using the syntax of this program, executing `./greet David` would result in the program saying `hello, David`.

Exit Status

- When a program ends, a special exit code is provided to the computer.
- When a program exits without error, a status code of `0` is provided to the computer. Often, when an error occurs that results in the program ending, a status of `1` is provided by the computer.
- You could write a program as follows that illustrates this by typing `code status.c` and writing code as follows:

```

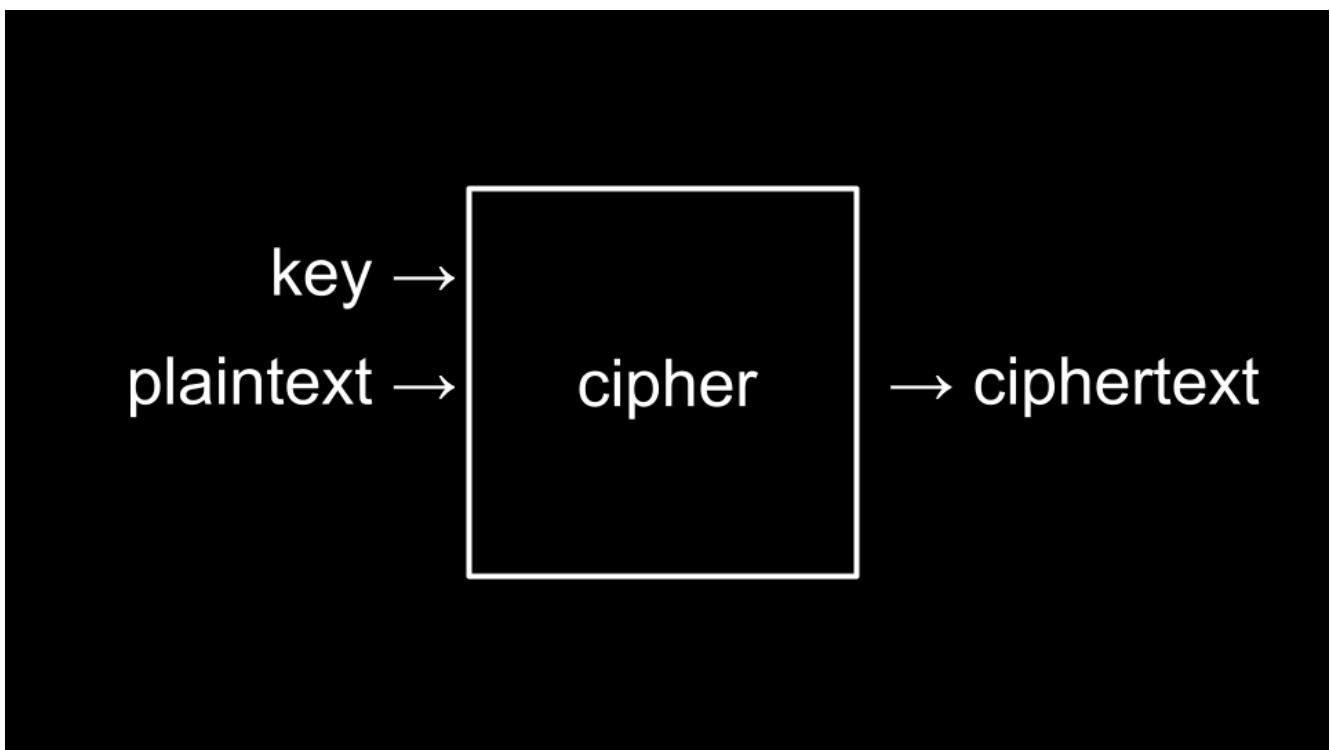
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(int argc, string argv[])
5 {
6     if (argc != 2)
7     {
8         printf("Missing command-line argument\n");
9         return 1;
10    }
11    printf("hello, %s\n", argv[1]);
12    return 0;
13 }
```

Notice that if you fail to provide `./status David`, you will get an exit status of `1`. However, if you do provide `./status David`, you will get an exit status of `0`.

- You can imagine how you might use portions of the above program to check if a user provided the correct number of command-line arguments.

Cryptography

- Cryptography is the art of ciphering and deciphering a message.
- `plaintext` and a `key` are provided to a `cipher`, resulting in `ciphertext`.



- The key is a special argument passed to the cipher along with the plaintext. The cipher uses the key to make decisions about how to implement its cipher algorithm.

Summing Up

In this lesson, you learned more details about compiling and how data is stored within a computer. Specifically, you learned...

- Generally, how a compiler works.
- How to debug your code using four methods.
- How to utilize arrays within your code.
- How arrays store data in back to back portions of memory.
- How strings are simply arrays of characters.
- How to interact with arrays in your code.
- How command-line arguments can be passed to your programs.
- The basic building-blocks of cryptography.

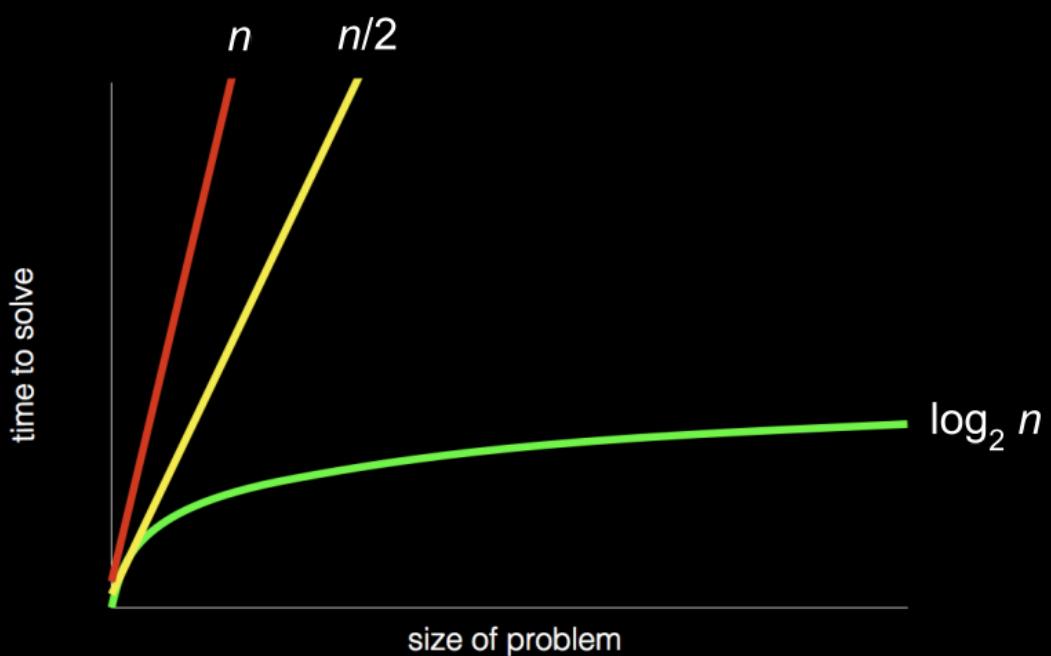
See you next time!

Lecture 3-Algorithms

- [Welcome!](#)
- [Linear Search](#)
- [Binary Search](#)
- [Running Time](#)
- [search.c](#)
- [Data Structures](#)
- [Sorting](#)
- [Bubble Sort](#)
- [Recursion](#)
- [Merge Sort](#)
- [Summing Up](#)

Welcome!

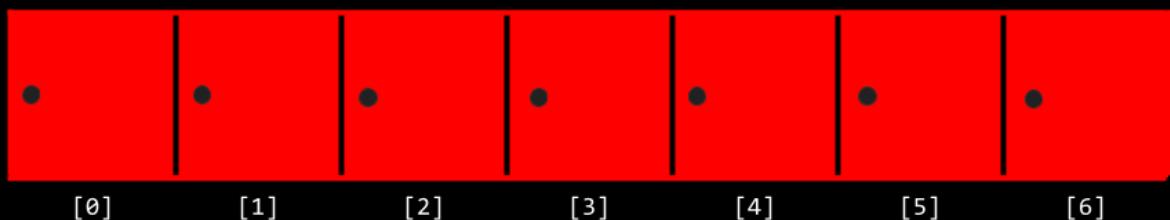
- In week zero, we introduced the idea of an *algorithm*: a black box that may take an input and creates an output.
- This week, we are going to expand upon our understanding of algorithms through pseudocode and into code itself.
- Also, we are going to consider the efficiency of these algorithms. Indeed, we are going to be building upon our understanding of how to use some of the *lower-level* concepts we discussed last week in building algorithms.
- Recall back to earlier in the course when we introduced the following graph:



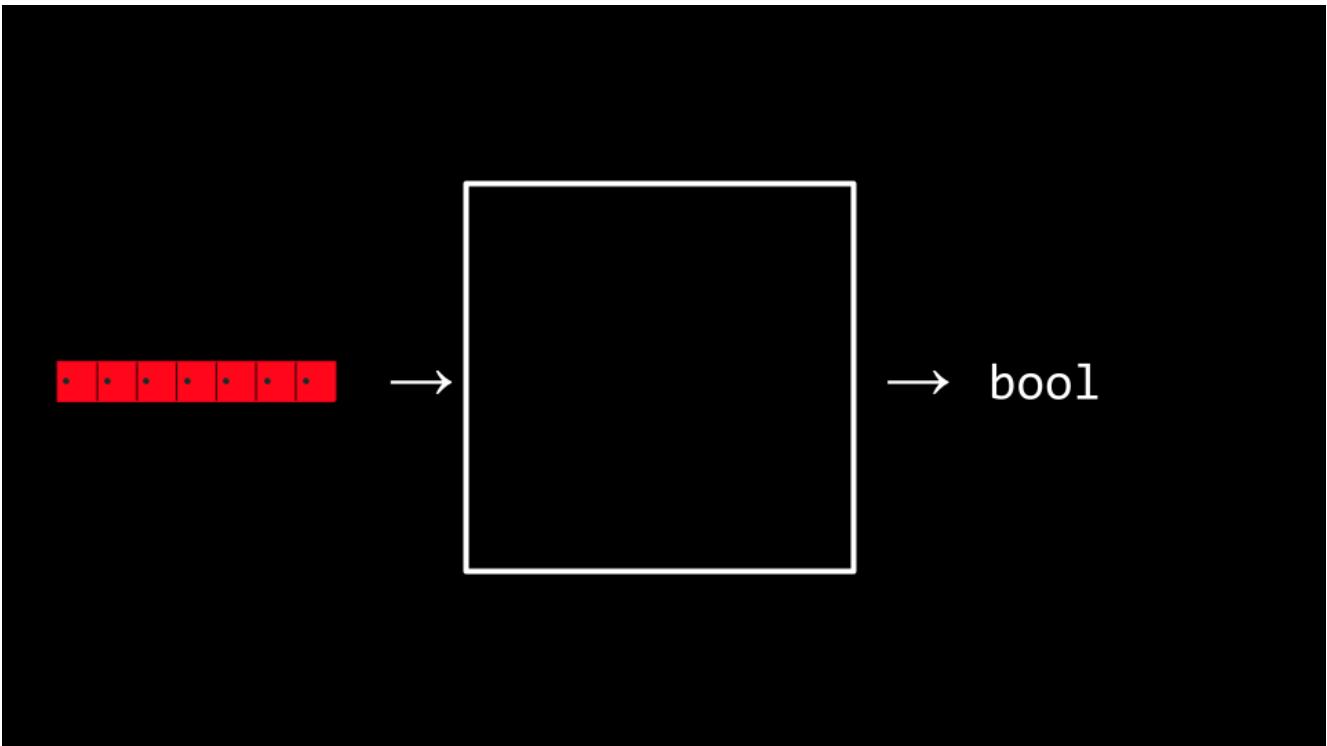
- As we step into this week, you should consider how the way an algorithm works with a problem may determine the time it takes to solve a problem! Algorithms can be designed to be more and more efficient, to a limit.
- Today, we will focus upon the design of algorithms and how to measure their efficiency.

Linear Search

- Recall that last week you were introduced to the idea of an *array*, blocks of memory that are consecutive: side-by-side with one another.
- You can metaphorically imagine an array like a series of seven red lockers as follows:



- We can imagine that we have an essential problem of wanting to know, “Is the number 50 inside an array?” A computer must look at each locker to be able to see if the number 50 is inside. We call this process of finding such a number, character, string, or other item *searching*.
- We can potentially hand our array to an algorithm, wherein our algorithm will search through our lockers to see if the number 50 is behind one of the doors: Returning the value true or false.



- We can imagine various instructions we might provide our algorithm to undertake this task as follows:

```

1 For each door from left to right
2   If 50 is behind door
3     Return true
4 Return false
  
```

Notice that the above instructions are called *pseudocode*: A human-readable version of the instructions that we could provide the computer.

- A computer scientist could translate that pseudocode as follows:

```

1 For i from 0 to n-1
2   If 50 is behind doors[i]
3     Return true
4 Return false
  
```

Notice that the above is still not code, but it is a pretty close approximation of what the final code might look like.

Binary Search

- *Binary search* is another *search algorithm* that could be employed in our task of finding the 50.

- Assuming that the values within the lockers have been arranged from smallest to largest, the pseudocode for binary search would appear as follows:

```

1 If no doors left
2   Return false
3 If 50 is behind middle door
4   Return true
5 Else if 50 < middle door
6   Search left half
7 Else if 50 > middle door
8   Search right half

```

- Using the nomenclature of code, we can further modify our algorithm as follows:

```

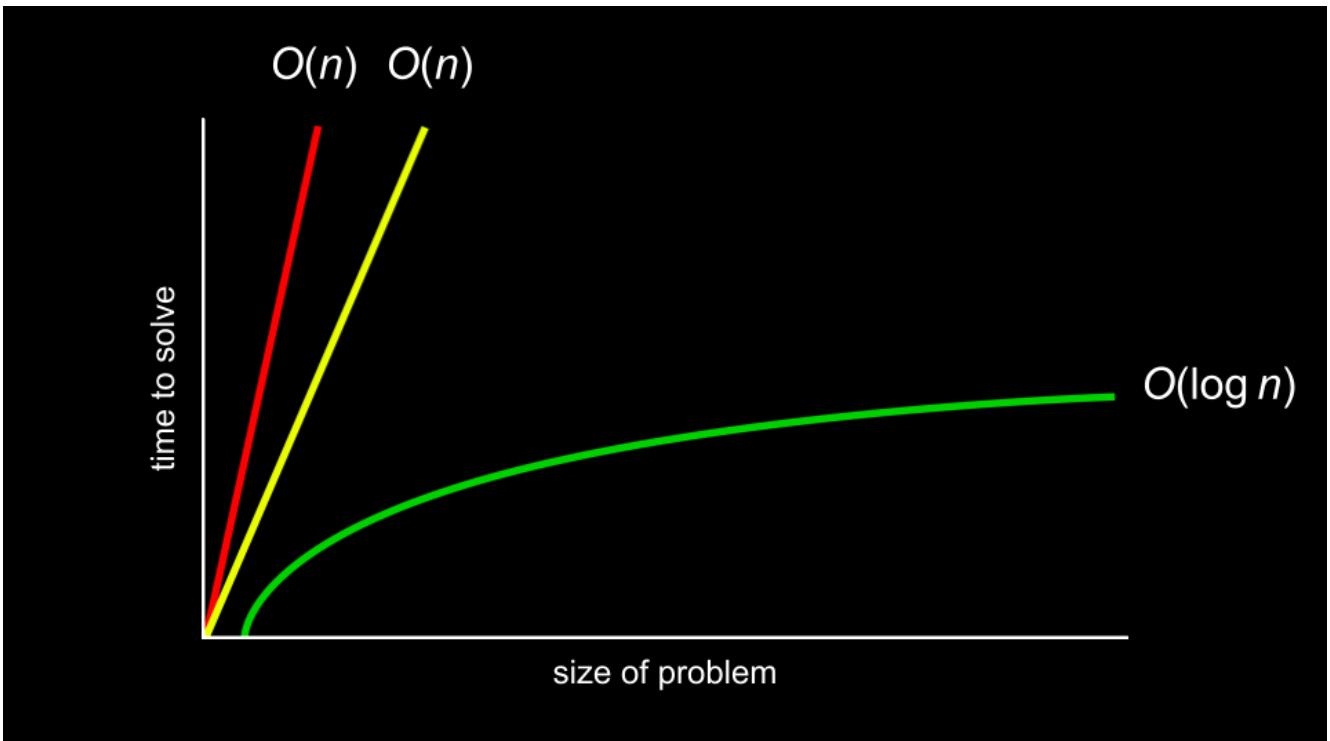
1 If no doors left
2   Return false
3 If 50 is behind doors[middle]
4   Return true
5 Else if 50 < doors[middle]
6   Search doors[0] through doors[middle - 1]
7 Else if 50 > doors[middle]
8   Search doors[middle + 1] through doors[n - 1]

```

Notice, looking at this approximation of code, you can nearly imagine what this might look like in actual code.

Running Time

- running time* involves an analysis using *big O* notation. Take a look at the following graph:



- Rather than being ultra-specific about the mathematical efficiency of an algorithm, computer scientists discuss efficiency in terms of *the order* of various running times.
- In the above graph, the first algorithm is $\mathcal{O}(n)$ or *in the order of n*. The second is in $\mathcal{O}(n)$ as well. The third is in $\mathcal{O}(\log n)$.
- It's the shape of the curve that shows the efficiency of an algorithm. Some common running times we may see are:
 - $\mathcal{O}(n^2)$
 - $\mathcal{O}(n \log n)$
 - $\mathcal{O}(n)$
 - $\mathcal{O}(\log n)$
 - $\mathcal{O}(1)$
- Of the running times above, $\mathcal{O}(n^2)$ is considered the worst running time, $\mathcal{O}(1)$ is the fastest.
- Linear search was of order $\mathcal{O}(n)$ because it could take n steps in the worst case to run.
- Binary search was of order $\mathcal{O}(\log n)$ because it would take fewer and fewer steps to run even in the worst case.
- Programmers are interested in both the worst case, or *upper bound*, and the best case, or *lower bound*.
- The Ω symbol is used to denote the best case of an algorithm, such as $\Omega(\log n)$.
- The Θ symbol is used to denote where the upper bound and lower bound are the same, where the best case and the worst case running times are the same.
- As you continue to develop your knowledge in computer science, you will explore these topics in more detail in future courses.

search.c

- You can implement linear search ourselves by typing `code search.c` in your terminal window and by writing code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // An array of integers
7     int numbers[] = {20, 500, 10, 5, 100, 1, 50};
8
9     // Search for number
10    int n = get_int("Number: ");
11    for (int i = 0; i < 7; i++)
12    {
13        if (numbers[i] == n)
14        {
15            printf("Found\n");
16            return 0;
17        }

```

```

18 }
19 printf("Not found\n");
20 return 1;
21 }
```

Notice that the line beginning with `int numbers[]` allows us to define the values of each element of the array as we create it. Then, in the `for` loop, we have an implementation of linear search. `return 0` is used to indicate success and exit the program. `return 1` is used to exist the program with an error (failure).

- We have now implemented linear search ourselves in C!
- What if we wanted to search for a string within an array? Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     // An array of strings
8     string strings[] = {"battleship", "boot", "cannon", "iron", "thimble", "top
9     hat"};
10
11    // Search for string
12    string s = get_string("String: ");
13    for (int i = 0; i < 6; i++)
14    {
15        if (strcmp(strings[i], s) == 0)
16        {
17            printf("Found\n");
18            return 0;
19        }
20    }
21    printf("Not found\n");
22    return 1;
23 }
```

Notice that we cannot utilize `==` as in our previous iteration of this program. Instead, we use `strcmp`, which comes from the `string.h` library. `strcmp` will return `0` if the strings are the same.

- Indeed, running this code allows us to iterate over this array of strings to see if a certain string was within it. However, if you see a *segmentation fault*, where a part of memory was touched by your program that it should not have access to, do make sure you have `i < 6` noted above instead of `i < 7`.
- We can combine these ideas of both numbers and strings into a single program. Type `code phonebook.c` into your terminal window and write code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
```

```

4
5 int main(void)
6 {
7     // Arrays of strings
8     string names[] = {"Carter", "David", "John"};
9     string numbers[] = {"+1-617-495-1000", "+1-617-495-1000", "+1-949-468-2750"};
10
11    // Search for name
12    string name = get_string("Name: ");
13    for (int i = 0; i < 3; i++)
14    {
15        if (strcmp(names[i], name) == 0)
16        {
17            printf("Found %s\n", numbers[i]);
18            return 0;
19        }
20    }
21    printf("Not found\n");
22    return 1;
23 }
```

Notice that Carter's number begins with `+1-617`, David's phone number starts with `+1-617`, and John's number starts with `+1-949`. Therefore, `names[0]` is Carter and `numbers[0]` is Carter's number. This code will allow us to search the phonebook to for a person's specific number.

- While this code works, there are numerous inefficiencies. Indeed, there is a chance that people's names and numbers may not correspond. Wouldn't be nice if we could create our own data type where we could associate a person with the phone number?

Data Structures

- It turns out that C allows a way by which we can create our own data types via a `struct`.
- Would it not be useful to create our own data type called a `person`, that has inside of it a `name` and `number`.
- Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 typedef struct
6 {
7     string name;
8     string number;
9 }
10 person;
11
12 int main(void)
13 {
14     person people[3];
```

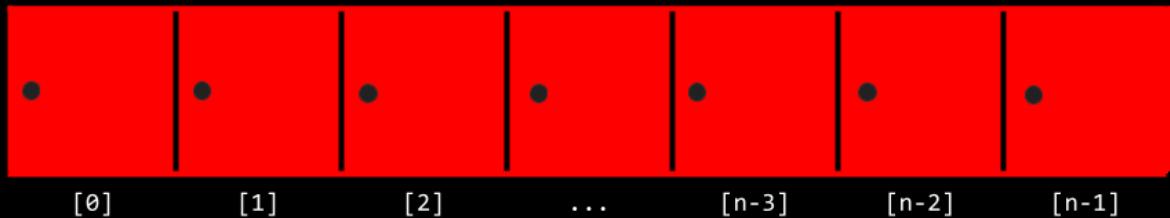
```

15
16     people[0].name = "Carter";
17     people[0].number = "+1-617-495-1000";
18
19     people[1].name = "David";
20     people[1].number = "+1-617-495-1000";
21
22     people[2].name = "John";
23     people[2].number = "+1-949-468-2750";
24
25 // Search for name
26 string name = get_string("Name: ");
27 for (int i = 0; i < 3; i++)
28 {
29     if (strcmp(people[i].name, name) == 0)
30     {
31         printf("Found %s\n", people[i].number);
32         return 0;
33     }
34 }
35 printf("Not found\n");
36 return 1;
37 }
```

Notice that the code begins with `typedef struct` where a new datatype called `person` is defined. Inside a `person` is a string called `name` and a `string` called `number`. In the `main` function, begin by creating an array called `people` that is of type `person` that is a size of 3. Then, we update the names and phone numbers of the two people in our `people` array. Most important, notice how the *dot notation* such as `people[0].name` allows us to access the `person` at the 0th location and assign that individual a name.

Sorting

- *Sorting* is the act of taking an unsorted list of values and transforming this list into a sorted one.
- When a list is sorted, searching that list is far less taxing on the computer. Recall that we can use binary search on a sorted list, but not on an unsorted one.
- It turns out that there are many different types of sorting algorithms.
- *Selection sort* is one such search algorithm.
- We can represent an array as follows:



- The algorithm for selection sort in pseudocode is:

```

1 | For i from 0 to n-1
2 |   Find smallest number between numbers[i] and numbers[n-1]
3 |   Swap smallest number with numbers[i]

```

- Summarizing those steps, the first time iterating through the list took $n - 1$ steps. The second time, it took $n - 2$ steps. Carrying this logic forward, the steps required could be represented as follows:

```

1 | (n - 1) + (n - 2) + (n - 3) + ... + 1

```

- This could be simplified to $n(n-1)/2$ or, more simply, $\mathcal{O}(n^2)$.

Bubble Sort

- Bubble sort* is another sorting algorithm that works by repeatedly swapping elements to “bubble” larger elements to the end.
- The pseudocode for bubble sort is:

```

1 | Repeat n-1 times
2 |   For i from 0 to n-2
3 |     If numbers[i] and numbers[i+1] out of order
4 |       Swap them
5 |     If no swaps
6 |       Quit

```

- As we further sort the array, 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.

- Analyzing selection sort, we made only seven comparisons. Representing this mathematically, where n represents the number of cases, it could be said that selection sort can be analyzed as:

```
1 | (n - 1) + (n - 2) + (n - 3) + ... + 1
```

or, more simply $(n^2/2 - n/2)$.

- Considering that mathematical analysis, n^2 is really the most influential factor in determining the efficiency of this algorithm. Therefore, selection sort is considered to be of the order of $\mathcal{O}(n^2)$ in the worst case where all values are unsorted. Even when all values are sorted, it will take the same number of steps. Therefore, the best case can be noted as $\mathcal{\Omega}(n^2)$. Since both the upper bound and lower bound cases are the same, the efficiency of this algorithm as a whole can be regarded as $\Theta(n^2)$.
- Analyzing bubble sort, the worst case is $\mathcal{O}(n^2)$. The best case is $\mathcal{\Omega}(n)$.
- You can [visualize](#) a comparison of these algorithms.

Recursion

- How could we improve our efficiency in our sorting?
- Recursion* is a concept within programming where a function calls itself. We saw this earlier when we saw...

```
1 | If no doors left
2 |   Return false
3 | If number behind middle door
4 |   Return true
5 | Else if number < middle door
6 |   Search left half
7 | Else if number > middle door
8 |   Search right half
```

Notice that we are calling `search` on smaller and smaller iterations of this problem.

- Similarly, in our pseudocode for Week 0, you can see where recursion was implemented:

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

- This code could have been simplified, to highlight its recursive properties as follows:

```

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

```

- Consider how in Week 1 we wanted to create a pyramid structure as follows:
- To implement this using recursion, type `code recursion.c` into your terminal window and write code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 void draw(int n);
5
6 int main(void)
7 {
8     draw(1);
9 }
10
11 void draw(int n)
12 {
13     for (int i = 0; i < n; i++)
14     {
15         printf("#");
16     }
17     printf("\n");
18
19     draw(n + 1);
20 }

```

Notice that the `draw` function calls itself. Further, note that your code may get caught in an infinite loop. To break from this loop, if you get stuck, hit `ctrl-c` on your keyboard. The reason this creates an infinite loop is that there is nothing telling the program to end. There is no case where the program is done.

- We can correct our code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 void draw(int n);

```

```

5
6 int main(void)
7 {
8     // Get height of pyramid
9     int height = get_int("Height: ");
10
11    // Draw pyramid
12    draw(height);
13 }
14
15 void draw(int n)
16 {
17     // If nothing to draw
18     if (n <= 0)
19     {
20         return;
21     }
22
23     // Draw pyramid of height n - 1
24     draw(n - 1);
25
26     // Draw one more row of width n
27     for (int i = 0; i < n; i++)
28     {
29         printf("#");
30     }
31     printf("\n");
32 }
```

Notice the *base case* will ensure the code does not run forever. The line `if (n <= 0)` terminates the recursion because the problem has been solved. Every time `draw` calls itself, it calls itself by `n-1`. At some point, `n-1` will equal `0`, resulting in the `draw` function returning and the program will end.

Merge Sort

- We can now leverage recursion in our quest for a more efficient sort algorithm and implement what is called *merge sort*, a very efficient sort algorithm.
- The pseudocode for merge sort is quite short:

```

1 If only one number
2     Quit
3 Else
4     Sort left half of number
5     Sort right half of number
6     Merge sorted halves
```

- Consider the following list of numbers:
- First, merge sort asks, “is this one number?” The answer is “no,” so the algorithm continues.

- Second, merge sort will now split the numbers down the middle (or as close as it can get) and sort the left half of numbers.
- Third, merge sort would look at these numbers on the left and ask, “is this one number?” Since the answer is no, it would then split the numbers on the left down the middle.
- Fourth, merge sort will again ask , “is this one number?” The answer is yes this time! Therefore, it will quit this task and return to the last task it was running at this point:
- Fifth, merge sort will sort the numbers on the left.
- Now, we return to where we left off in the pseudocode now that the left side has been sorted. A similar process of steps 3-5 will occur with the right-hand numbers. This will result in:
- Both halves are now sorted. Finally, the algorithm will merge both sides. It will look at the first number on the left and the first number on the right. It will put the smaller number first, then the second smallest. The algorithm will repeat this for all numbers, resulting in:
- Merge sort is complete, and the program quits.
- Merge sort is a very efficient sort algorithm with a worst case of $\mathcal{O}(n \log n)$. The best case is still $(\Omega(n \log n))$ because the algorithm still must visit each place in the list. Therefore, merge sort is also $(\Theta(n \log n))$ since the best case and worst case are the same.
- A final [visualization](#) was shared.

Summing Up

In this lesson, you learned about algorithmic thinking and building your own data types. Specifically, you learned...

- Algorithms.
- Big O notation.
- Binary search and linear search.
- Various sort algorithms, including bubble sort, selection sort, and merge sort.
- Recursion.

See you next time!

Lecture 4-Memory

- [Welcome!](#)
- [Pixel Art](#)
- [Hexadecimal](#)
- [Memory](#)
- [Pointers](#)
- [Strings](#)
- [Pointer Arithmetic](#)

- [String Comparison](#)
- [Copying](#)
- [malloc and Valgrind](#)
- [Garbage Values](#)
- [Pointer Fun with Binky](#)
- [Swap](#)
- [Overflow](#)
- [scanf](#)
- [File I/O](#)
- [Summing Up](#)

Welcome!

- In previous weeks, we talked about images being made of smaller building blocks called pixels.
- Today, we will go into further detail about the zeros and ones that make up these images. In particular, we will be going deeper into the fundamental building blocks that make up files, including images.
- Further, we will discuss how to access the underlying data stored in computer memory.

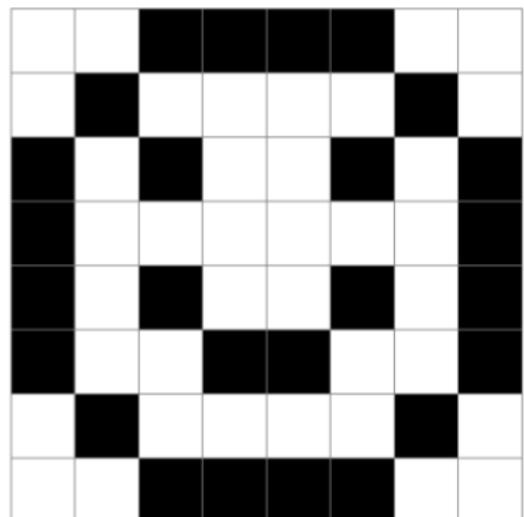
Pixel Art

- Pixels are squares, individual dots, of color that are arranged on an up-down, left-right grid.
- You can imagine an image as a map of bits, where zeros represent black and ones represent white.

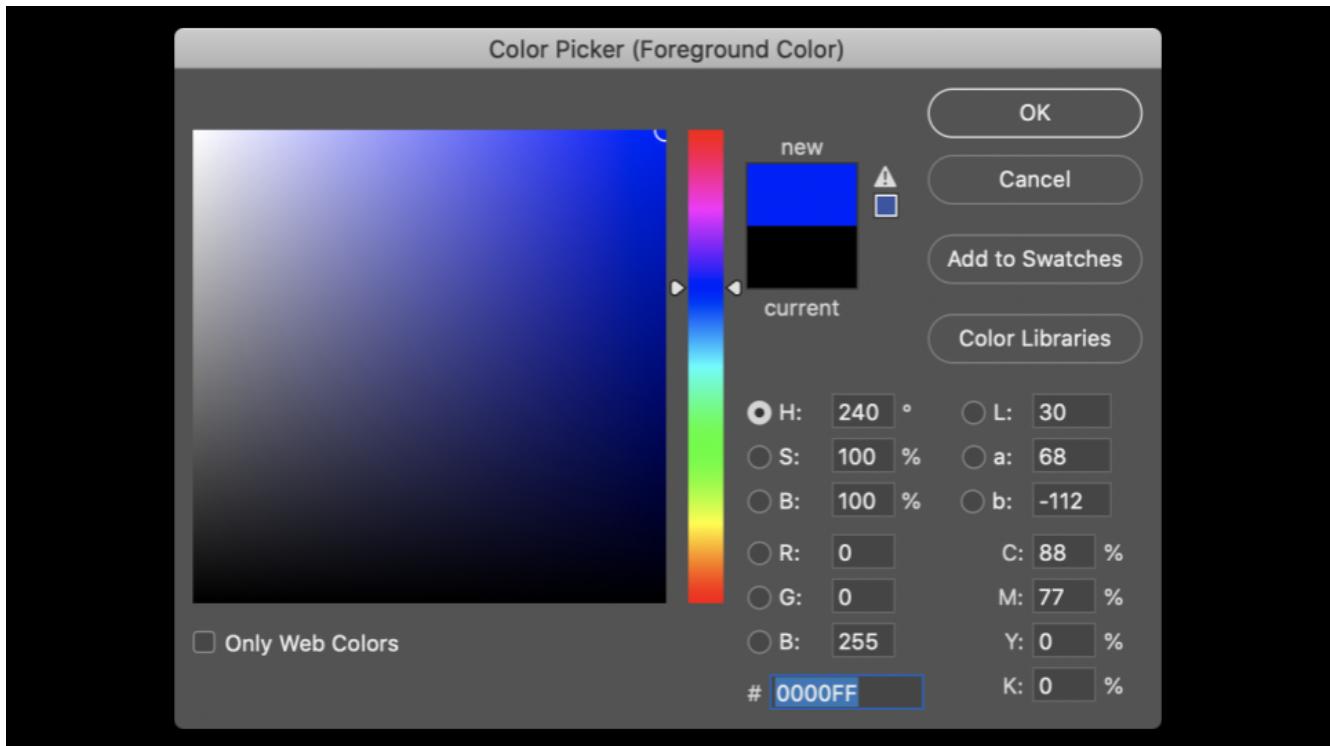
```

1 1 0 0 0 0 1 1
1 0 1 1 1 1 0 1
0 1 0 1 1 0 1 0
0 1 1 1 1 1 1 0
0 1 0 1 1 0 1 0
0 1 1 0 0 1 1 0
1 0 1 1 1 1 0 1
1 1 0 0 0 0 1 1

```



- *RGB*, or *red*, *green*, *blue*, are numbers that represent the amount of each of these colors. In Adobe Photoshop, you can see these settings as follows:



Notice how the amount of red, blue, and green changes the color selected.

- You can see by the image above that color is not just represented in three values. At the bottom of the window, there is a special value made up of numbers and characters. `255` is represented as `FF`. Why might this be?

Hexadecimal

- *Hexadecimal* is a system of counting that has 16 counting values. They are as follows:

1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Notice that `F` represents `15`.

- Hexadecimal is also known as *base-16*.
- When counting in hexadecimal, each column is a power of 16.
- The number `0` is represented as `00`.
- The number `1` is represented as `01`.
- The number `9` is represented by `09`.
- The number `10` is represented as `0A`.
- The number `15` is represented as `0F`.
- The number `16` is represented as `10`.
- The number `255` is represented as `FF`, because 16×15 (or `F`) is 240. Add 15 more to make 255. This is the highest number you can count using a two-digit hexadecimal system.
- Hexadecimal is useful because it can be represented using fewer digits. Hexadecimal allows us to represent information more succinctly.

Memory

- In weeks past, you may recall our artist rendering of concurrent blocks of memory. Applying hexadecimal numbering to each of these blocks of memory, you can visualize these as follows:

0	1	2	3	4	5	6	7
8	9	A	B	C	D	E	F
10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F

- You can imagine how there may be confusion regarding whether the `10` block above may represent a location in memory or the value `10`. Accordingly, by convention, all hexadecimal numbers are often represented with the `0x` prefix as follows:

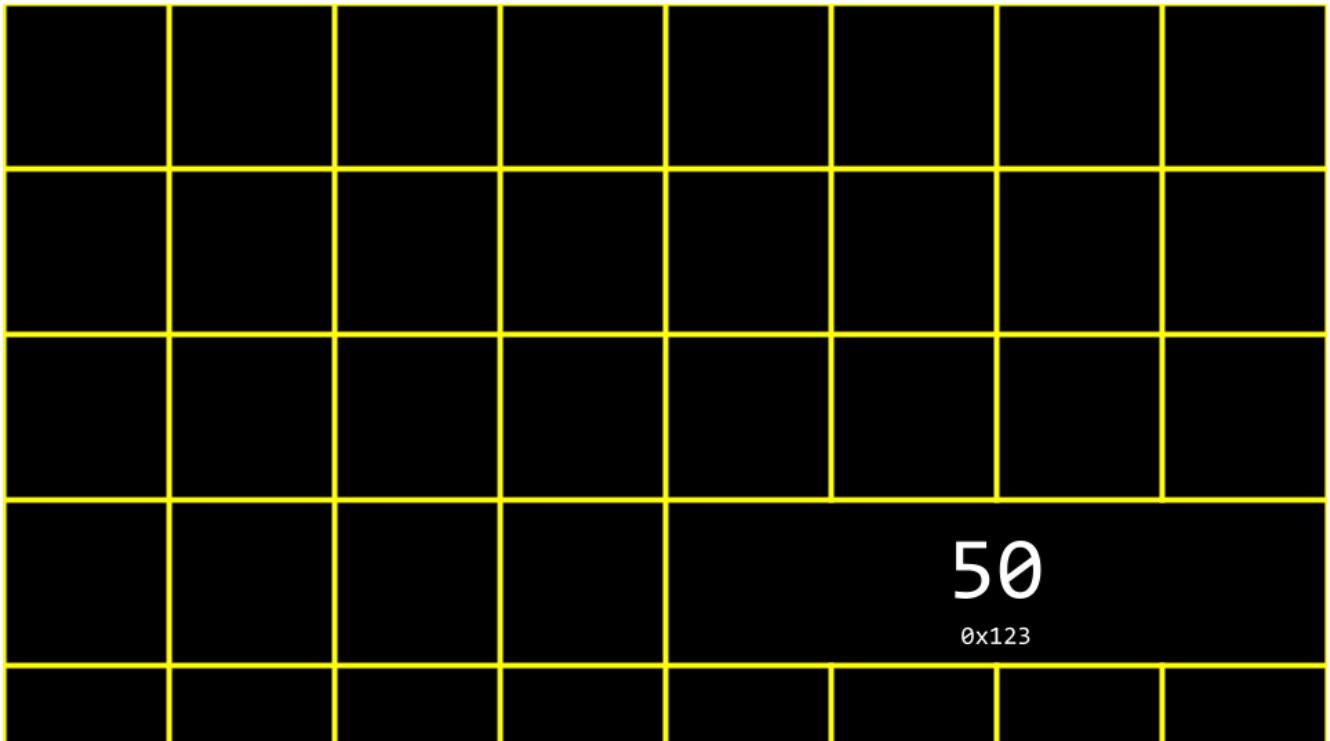
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F

- In your terminal window, type `code addresses.c` and write your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n = 50;
6     printf("%i\n", n);
7 }
```

Notice how `n` is stored in memory with the value `50`.

- You can visualize how this program stores this value as follows:



- The `c` language has two powerful operators that relate to memory:

```
1     & Provides the address of something stored in memory.
2     * Instructs the compiler to go to a location in memory.
```

- We can leverage this knowledge by modifying our code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n = 50;
6     printf("%p\n", &n);
7 }
```

Notice the `%p`, which allows us to view the address of a location in memory. `&n` can be literally translated as "the address of `n`." Executing this code will return an address of memory beginning with `0x`.

Pointers

- A *pointer* is a variable that contains the address of some value. Most succinctly, a pointer is an address in your computer's memory.
- Consider the following code:
Notice that `p` is a pointer that contains the address of an integer `n`.
- Modify your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n = 50;
6     int *p = &n;
7     printf("%p\n", p);
8 }
```

Notice that this code has the same effect as our previous code. We have simply leveraged our new knowledge of the `&` and `*` operators.

- To illustrate the use of the `*` operator, consider the following:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n = 50;
6     int *p = &n;
7     printf("%i\n", *p);
8 }
```

Notice that the `printf` line prints the integer at the location of `p`. `int *p` creates a pointer whose job is to store the memory address of an integer.

- You can visualize our code as follows:

0x123

p

50

0x123

Notice the pointer seems rather large. Indeed, a pointer is usually stored as an 8-byte value. `p` is storing the address of the `50`.

- You can more accurately visualize a pointer as one address that points to another:



50

0x123

Strings

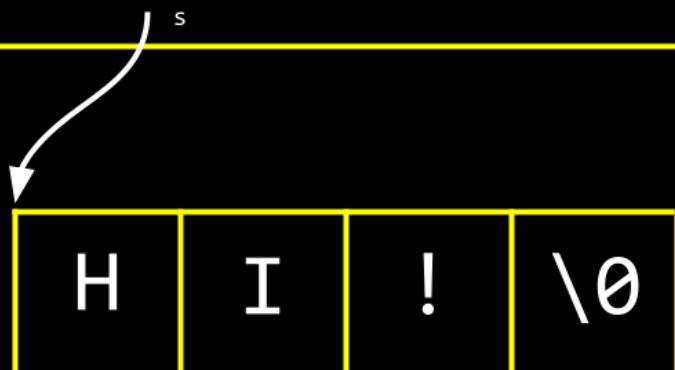
- Now that we have a mental model for pointers, we can peel back a level of simplification that was offered earlier in this course.
- Recall that a string is simply an array of characters. For example, `string s = "HI!"` can be represented as follows:

0x123

s



- However, what is `s` really? Where is the `s` stored in memory? As you can imagine, `s` needs to be stored somewhere. You can visualize the relationship of `s` to the string as follows:



Notice how a pointer called `s` tells the compiler where the first byte of the string exists in memory.

- Modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
{
    string 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]);
}

```

Notice the above prints the memory locations of each character in the string `s`. The `&` symbol is used to show the address of each element of the string. When running this code, notice that elements `0`, `1`, `2`, and `3` are next to one another in memory.

- Likewise, you can modify your code as follows:

```

1 #include <stdio.h>
2
3 int main(void)
{
    char *s = "HI!";
    printf("%s\n", s);
}

```

Notice that this code will present the string that starts at the location of `s`. This code effectively removes the training wheels of the `string` data type offered by `cs50.h`. This is raw C code, without the scaffolding of the `cs50` library.

- You can imagine how a string, as a data type, is created.
- Last week, we learned how to create your own data type as a struct.
- The `cs50` library includes a struct as follows: `typedef char *string`
- This struct, when using the `cs50` library, allows one to use a custom data type called `string`.

Pointer Arithmetic

- You can modify your code to accomplish the same thing in a longer form as follows:

```

1 #include <stdio.h>
2
3 int main(void)
{
    char *s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
}

```

Notice that we are printing each character at the location of `s`.

- Further, you can modify your code as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *s = "HI!";
6     printf("%c\n", *s);
7     printf("%c\n", *(s + 1));
8     printf("%c\n", *(s + 2));
9 }
```

Notice that the first character at the location of `s` is printed. Then, the character at the location `s + 1` is printed, and so on.

String Comparison

- A string of characters is simply an array of characters identified by its first byte.
- Earlier in the course, we considered the comparison of integers. We could represent this in code by typing `code compare.c` into the terminal window and writing code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get two integers
7     int i = get_int("i: ");
8     int j = get_int("j: ");
9
10    // Compare integers
11    if (i == j)
12    {
13        printf("Same\n");
14    }
15    else
16    {
17        printf("Different\n");
18    }
19 }
```

Notice that this code takes two integers from the user and compares them.

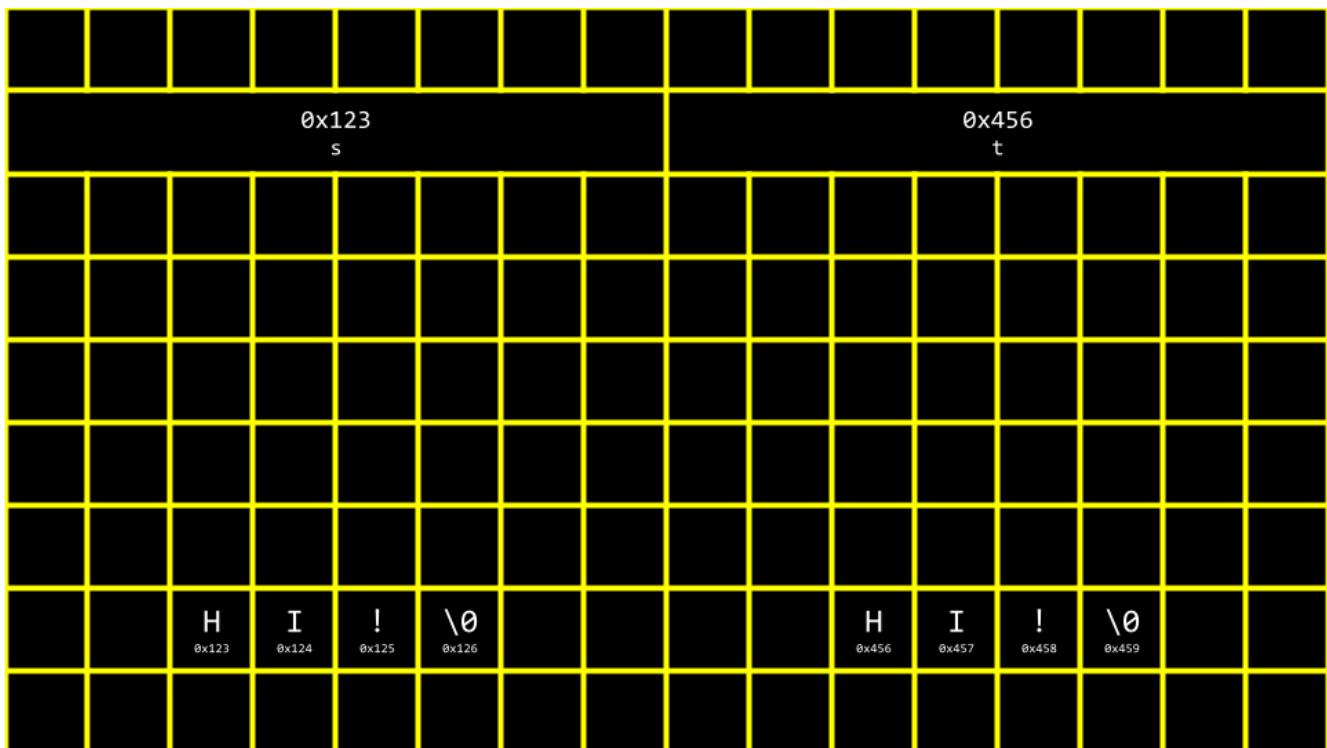
- In the case of strings, however, one cannot compare two strings using the `==` operator.
- Utilizing the `==` operator in an attempt to compare strings will attempt to compare the memory locations of the strings instead of the characters therein. Accordingly, we recommended the use of `strcmp`.
- To illustrate this, modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get two strings
7     char *s = get_string("s: ");
8     char *t = get_string("t: ");
9
10    // Compare strings' addresses
11    if (s == t)
12    {
13        printf("Same\n");
14    }
15    else
16    {
17        printf("Different\n");
18    }
19 }
```

Noticing that typing in `HI!` for both strings still results in the output of `Different`.

- Why are these strings seemingly different? You can use the following to visualize why:



- Therefore, the code for `compare.c` above is actually attempting to see if the memory addresses are different: not the strings themselves.
- Using `strcmp`, we can correct our code:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
```

```

5 int main(void)
6 {
7     // Get two strings
8     char *s = get_string("s: ");
9     char *t = get_string("t: ");
10
11    // Compare strings
12    if (strcmp(s, t) == 0)
13    {
14        printf("Same\n");
15    }
16    else
17    {
18        printf("Different\n");
19    }
20 }
```

Notice that `strcmp` can return `0` if the strings are the same.

- To further illustrate how these two strings are living in two locations, modify your code as follows:

```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get two strings
7     char *s = get_string("s: ");
8     char *t = get_string("t: ");
9
10    // Print strings
11    printf("%s\n", s);
12    printf("%s\n", t);
13 }
```

Notice how we now have two separate strings stored likely at two separate locations.

- You can see the locations of these two stored strings with a small modification:

```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     // Get two strings
7     char *s = get_string("s: ");
8     char *t = get_string("t: ");
9
10    // Print strings' addresses
11    printf("%p\n", s);
12    printf("%p\n", t);
13 }
```

Notice that the `%s` has been changed to `%p` in the print statement.

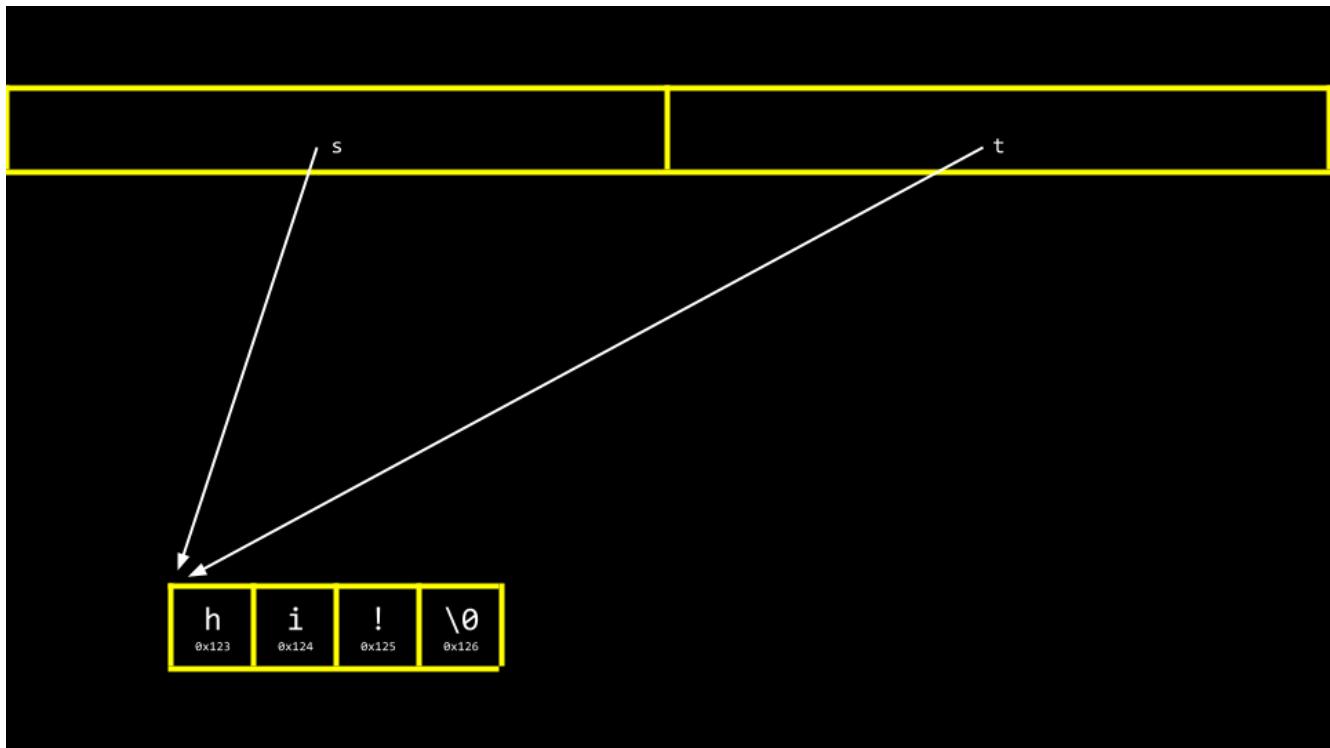
Copying

- A common need in programming is to copy one string to another.
- In your terminal window, type `code copy.c` and write code as follows:

```
1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void)
7 {
8     // Get a string
9     string s = get_string("s: ");
10
11    // Copy string's address
12    string t = s;
13
14    // Capitalize first letter in string
15    t[0] = toupper(t[0]);
16
17    // Print string twice
18    printf("s: %s\n", s);
19    printf("t: %s\n", t);
20 }
```

Notice that `string t = s` copies the address of `s` to `t`. This does not accomplish what we are desiring. The string is not copied – only the address is.

- You can visualize the above code as follows:



Notice that `s` and `t` are still pointing at the same blocks of memory. This is not an authentic copy of a string. Instead, these are two pointers pointing at the same string.

- Before we address this challenge, it's important to ensure that we don't experience a *segmentation fault* through our code, where we attempt to copy `string s` to `string t`, where `string t` does not exist. We can employ the `strlen` function as follows to assist with that:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(void)
7 {
8     // Get a string
9     string s = get_string("s: ");
10
11    // Copy string's address
12    string t = s;
13
14    // Capitalize first letter in string
15    if (strlen(t) > 0)
16    {
17        t[0] = toupper(t[0]);
18    }
19
20    // Print string twice
21    printf("s: %s\n", s);
22    printf("t: %s\n", t);
23 }
```

Notice that `strlen` is used to make sure `string t` exists. If it does not, nothing will be copied.

- To be able to make an authentic copy of the string, we will need to introduce two new building blocks. First, `malloc` allows you, the programmer, to allocate a block of a specific size of memory. Second, `free` allows you to tell the compiler to *free up* that block of memory you previously allocated.
- We can modify our code to create an authentic copy of our string as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main(void)
8 {
9     // Get a string
10    char *s = get_string("s: ");
11
12    // Allocate memory for another string
13    char *t = malloc(strlen(s) + 1);
14
15    // Copy string into memory, including '\0'
16    for (int i = 0; i <= strlen(s); i++)
17    {
18        t[i] = s[i];
19    }
20
21    // Capitalize copy
22    t[0] = toupper(t[0]);
23
24    // Print strings
25    printf("s: %s\n", s);
26    printf("t: %s\n", t);
27 }
```

Notice that `malloc(strlen(s) + 1)` creates a block of memory that is the length of the string `s` plus one. This allows for the inclusion of the *null* `\0` character in our final, copied string. Then, the `for` loop walks through the string `s` and assigns each value to that same location on the string `t`.

- It turns out that there is an inefficiency in our code. Modify your code as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main(void)
8 {
9     // Get a string
10    char *s = get_string("s: ");
11
12    // Allocate memory for another string
```

```

13  char *t = malloc(strlen(s) + 1);
14
15  // Copy string into memory, including '\0'
16  for (int i = 0, n = strlen(s); i <= n; i++)
17  {
18      t[i] = s[i];
19  }
20
21  // Capitalize copy
22  t[0] = toupper(t[0]);
23
24  // Print strings
25  printf("s: %s\n", s);
26  printf("t: %s\n", t);
27 }
```

Notice that `n = strlen(s)` is defined now in the left-hand side of the `for` loop. It's best not to call unneeded functions in the middle condition of the `for` loop, as it will run over and over again. When moving `n = strlen(s)` to the left-hand side, the function `strlen` only runs once.

- The `c` Language has a built-in function to copy strings called `strcpy`. It can be implemented as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main(void)
8 {
9     // Get a string
10    char *s = get_string("s: ");
11
12    // Allocate memory for another string
13    char *t = malloc(strlen(s) + 1);
14
15    // Copy string into memory
16    strcpy(t, s);
17
18    // Capitalize copy
19    t[0] = toupper(t[0]);
20
21    // Print strings
22    printf("s: %s\n", s);
23    printf("t: %s\n", t);
24 }
```

Notice that `strcpy` does the same work that our `for` loop previously did.

- Both `get_string` and `malloc` return `NULL`, a special value in memory, in the event that something goes wrong. You can write code that can check for this `NULL` condition as follows:

```

1 #include <cs50.h>
2 #include <ctype.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main(void)
8 {
9     // Get a string
10    char *s = get_string("s: ");
11    if (s == NULL)
12    {
13        return 1;
14    }
15
16    // Allocate memory for another string
17    char *t = malloc(strlen(s) + 1);
18    if (t == NULL)
19    {
20        return 1;
21    }
22
23    // Copy string into memory
24    strcpy(t, s);
25
26    // Capitalize copy
27    if (strlen(t) > 0)
28    {
29        t[0] = toupper(t[0]);
30    }
31
32    // Print strings
33    printf("s: %s\n", s);
34    printf("t: %s\n", t);
35
36    // Free memory
37    free(t);
38    return 0;
39 }

```

Notice that if the string obtained is of length `0` or `malloc` fails, `NULL` is returned. Further, notice that `free` lets the computer know you are done with this block of memory you created via `malloc`.

malloc and Valgrind

- *Valgrind* is a tool that can check to see if there are memory-related issues with your programs wherein you utilized `malloc`. Specifically, it checks to see if you `free` all the memory you allocated.
- Consider the following code for `memory.c`:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *x = malloc(3 * sizeof(int));
7     x[1] = 72;
8     x[2] = 73;
9     x[3] = 33;
10 }

```

Notice that running this program does not cause any errors. While `malloc` is used to allocate enough memory for an array, the code fails to `free` that allocated memory.

- If you type `make memory` followed by `valgrind ./memory`, you will get a report from valgrind that will report where memory has been lost as a result of your program. One error that valgrind reveals is that we attempted to assign the value of `33` at the 4th position of the array, where we only allocated an array of size `3`. Another error is that we never freed `x`.
- You can modify your code as follows:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *x = malloc(3 * sizeof(int));
7     x[0] = 72;
8     x[1] = 73;
9     x[2] = 33;
10    free(x);
11 }

```

Notice that running valgrind again now results in no memory leaks.

Garbage Values

- When you ask the compiler for a block of memory, there is no guarantee that this memory will be empty.
- It's very possible that this memory that you allocated was previously utilized by the computer. Accordingly, you may see *junk* or *garbage values*. This is a result of you getting a block of memory but not initializing it. For example, consider the following code for `garbage.c`:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int scores[1024];
6     for (int i = 0; i < 1024; i++)
7     {
8         printf("%i\n", scores[i]);
9     }
10 }
```

Notice that running this code will allocate 1024 locations in memory for your array, but the for loop will likely show that not all values therein are 0. It's always best practice to be aware of the potential for garbage values when you do not initialize blocks of memory to some other value like zero or otherwise.

Pointer Fun with Binky

- We watched a [video from Stanford University](#) that helped us visualize and understand pointers.

Swap

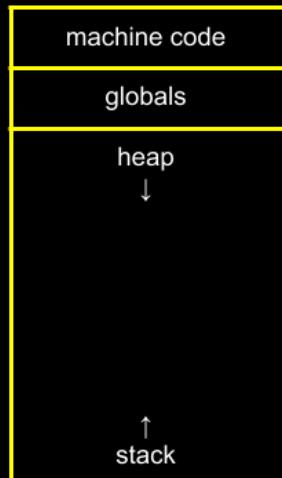
- In the real world, a common need in programming is to swap two values. Naturally, it's hard to swap two variables without a temporary holding space. In practice, you can type `code swap.c` and write code as follows to see this in action:

```

1 #include <stdio.h>
2
3 void swap(int a, int b);
4
5 int main(void)
6 {
7     int x = 1;
8     int y = 2;
9
10    printf("x is %i, y is %i\n", x, y);
11    swap(x, y);
12    printf("x is %i, y is %i\n", x, y);
13 }
14
15 void swap(int a, int b)
16 {
17     int tmp = a;
18     a = b;
19     b = tmp;
20 }
```

Notice that while this code runs, it does not work. The values, even after being sent to the swap function, do not swap. Why?

- When you pass values to a function, you are only providing copies. In previous weeks, we discussed the concept of *scope*. The values of `x` and `y` created in the curly `{}` braces of the `main` function only have the scope of the `main` function. Consider the following image:



Notice that *global* variables, which we have not used in this course, live in one place in memory. Various functions are stored in the `stack` in another area of memory.

- Now, consider the following image:



Notice that `main` and `swap` have two separate *frames* or areas of memory. Therefore, we cannot simply pass the values from one function to another to change them.

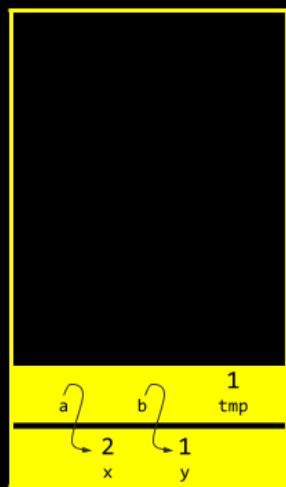
- Modify your code as follows:

```

1 #include <stdio.h>
2
3 void swap(int *a, int *b);
4
5 int main(void)
6 {
7     int x = 1;
8     int y = 2;
9
10    printf("x is %i, y is %i\n", x, y);
11    swap(&x, &y);
12    printf("x is %i, y is %i\n", x, y);
13 }
14
15 void swap(int *a, int *b)
16 {
17     int tmp = *a;
18     *a = *b;
19     *b = tmp;
20 }
```

Notice that variables are not passed by *value* but by *reference*. That is, the addresses of `a` and `b` are provided to the function. Therefore, the `swap` function can know where to make changes to the actual `a` and `b` from the main function.

- You can visualize this as follows:



Overflow

- A *heap overflow* is when you overflow the heap, touching areas of memory you are not supposed to.
- A *stack overflow* is when too many functions are called, overflowing the amount of memory available.

- Both of these are considered *buffer overflows*.

scanf

- In CS50, we have created functions like `get_int` to simplify the act of getting input from the user.
- `scanf` is a built-in function that can get user input.
- We can reimplement `get_int` rather easily using `scanf` as follows:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x;
6     printf("x: ");
7     scanf("%i", &x);
8     printf("x: %i\n", x);
9 }
```

Notice that the value of `x` is stored at the location of `x` in the line `scanf("%i", &x)`.

- However, attempting to reimplement `get_string` is not easy. Consider the following:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *s;
6     printf("s: ");
7     scanf("%s", s);
8     printf("s: %s\n", s);
9 }
```

Notice that no `&` is required because strings are special. Still, this program will not function. Nowhere in this program do we allocate the amount of memory required for our string. Indeed, we don't know how long of a string may be inputted by the user!

- Further, your code could be modified as follows. However, we have to pre-allocate a certain amount of memory for a string:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     char *s = malloc(4);
7     if (s == NULL)
8     {
9         return 1;
10    }
11    printf("s: ");
```

```

12     scanf("%s", s);
13     printf("s: %s\n", s);
14     free(s);
15     return 0;
16 }
```

Notice that if a string that is six bytes is provided you *might* get an error.

- Simplifying our code as follows we can further understand this essential problem of pre-allocation:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char s[4];
6     printf("s: ");
7     scanf("%s", s);
8     printf("s: %s\n", s);
9 }
```

Notice that if we pre-allocate an array of size 4, we can type `cat` and the program functions. However, a string larger than this *could* create an error.

- Sometimes, the compiler or the system running it may allocate more memory than we indicate. Fundamentally, though, the above code is unsafe. We cannot trust that the user will input a string that fits into our pre-allocated memory.

File I/O

- You can read from and manipulate files. While this topic will be discussed further in a future week, consider the following code for `phonebook.c`:

```

1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     // Open CSV file
8     FILE *file = fopen("phonebook.csv", "a");
9
10    // Get name and number
11    char *name = get_string("Name: ");
12    char *number = get_string("Number: ");
13
14    // Print to file
15    fprintf(file, "%s,%s\n", name, number);
16
17    // Close file
18    fclose(file);
19 }
```

Notice that this code uses pointers to access the file.

- You can create a file called `phonebook.csv` in advance of running the above code. After running the above program and inputting a name and phone number, you will notice that this data persists in your CSV file.
- If we want to ensure that `phonebook.csv` exists prior to running the program, we can modify our code as follows:

```
1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     // Open CSV file
8     FILE *file = fopen("phonebook.csv", "a");
9     if (!file)
10    {
11        return 1;
12    }
13
14    // Get name and number
15    char *name = get_string("Name: ");
16    char *number = get_string("Number: ");
17
18    // Print to file
19    fprintf(file, "%s,%s\n", name, number);
20
21    // Close file
22    fclose(file);
23 }
```

Notice that this program protects against a `NULL` pointer by invoking `return 1`.

- We can implement our own copy program by typing `code cp.c` and writing code as follows:

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 typedef uint8_t BYTE;
5
6 int main(int argc, char *argv[])
7 {
8     FILE *src = fopen(argv[1], "rb");
9     FILE *dst = fopen(argv[2], "wb");
10
11     BYTE b;
12
13     while (fread(&b, sizeof(b), 1, src) !=0)
14     {
15         fwrite(&b, sizeof(b), 1, dst);
```

```
16    }
17
18    fclose(dst);
19    fclose(src);
20 }
```

Notice that this file creates our own data type called a `BYTE` that is the size of a `uint8_t`. Then, the file reads a `BYTE` and writes it to a file.

- BMPs are also assortments of data that we can examine and manipulate. This week, you will be doing just that in your problem sets!

Summing Up

In this lesson, you learned about pointers that provide you with the ability to access and manipulate data at specific memory locations. Specifically, we delved into...

- Pixel art
- Hexadecimal
- Memory
- Pointers
- Strings
- Pointer Arithmetic
- String Comparison
- Copying
- malloc and Valgrind
- Garbage values
- Swapping
- Overflow
- `scanf`
- File I/O

See you next time!

Lecture 5-Data Structures

- [Welcome!](#)
- [Data Structures](#)
- [Stacks and Queues](#)
- [Jack Learns the Facts](#)
- [Resizing Arrays](#)
- [Linked Lists](#)

- [Trees](#)
- [Dictionaries](#)
- [Hashing and Hash Tables](#)
- [Tries](#)
- [Summing Up](#)

Welcome!

- All the prior weeks have presented you with the fundamental building blocks of programming.
- All you have learned in C will enable you to implement these building blocks in higher-level programming languages such as Python.
- Today, we are going to talk about organizing data in memory and design possibilities that emerge from your growing knowledge.

Data Structures

- *Data structures* essentially are forms of organization in memory.
- There are many ways to organize data in memory.
- *Abstract data structures* are those that we can conceptually imagine. When learning about computer science, it's often useful to begin with these conceptual data structures. Learning these will make it easier later to understand how to implement more concrete data structures.

Stacks and Queues

- *Queues* are one form of abstract data structure.
 - Queues have specific properties. Namely, they are *FIFO* or “first in first out.” You can imagine yourself in a line for a ride at an amusement park. The first person in the line gets to go on the ride first. The last person gets to go on the ride last.
 - Queues have specific actions associated with them. For example, an item can be *enqueued*; that is, the item can join the line or queue. Further, an item can be *dequeued* or leave the queue once it reaches the front of the line.
 - Queues contrast a *stack*. Fundamentally, the properties of a stack are different than a queue. Specifically, it is *LIFO* or “last in first out.” Just like stacking trays in a cafeteria, a tray that is placed in a stack last is the first that may be picked up.
 - Stacks have specific actions associated with them. For example, *push* places something on top of a stack. *Pop* is removing something from the top of the stack.
 -
- In code, you might imagine a stack as follows:

```
1 typedef struct
2 {
3     person people[CAPACITY];
4     int size;
5 }
6 stack;
```

Notice that an array called `people` is of type `person`. The `CAPACITY` is how high the stack could be. The integer `size` is how full the stack actually is, regardless of how much it *could* hold.

- You might imagine that the above code has a limitation. Since the capacity of the array is always predetermined in this code. Therefore, the stack may always be oversized. You might imagine only using one place in the stack out of 5000.
- It would be nice for our stack to be dynamic – able to grow as items are added to it.

Jack Learns the Facts

- We watched a video called [Jack Learns the Facts](#) by Professor Shannon Duvall of Elon University.

Resizing Arrays

- Rewinding to Week 2, we introduced you to your first data structure.
- An array is a block of contiguous memory.
-

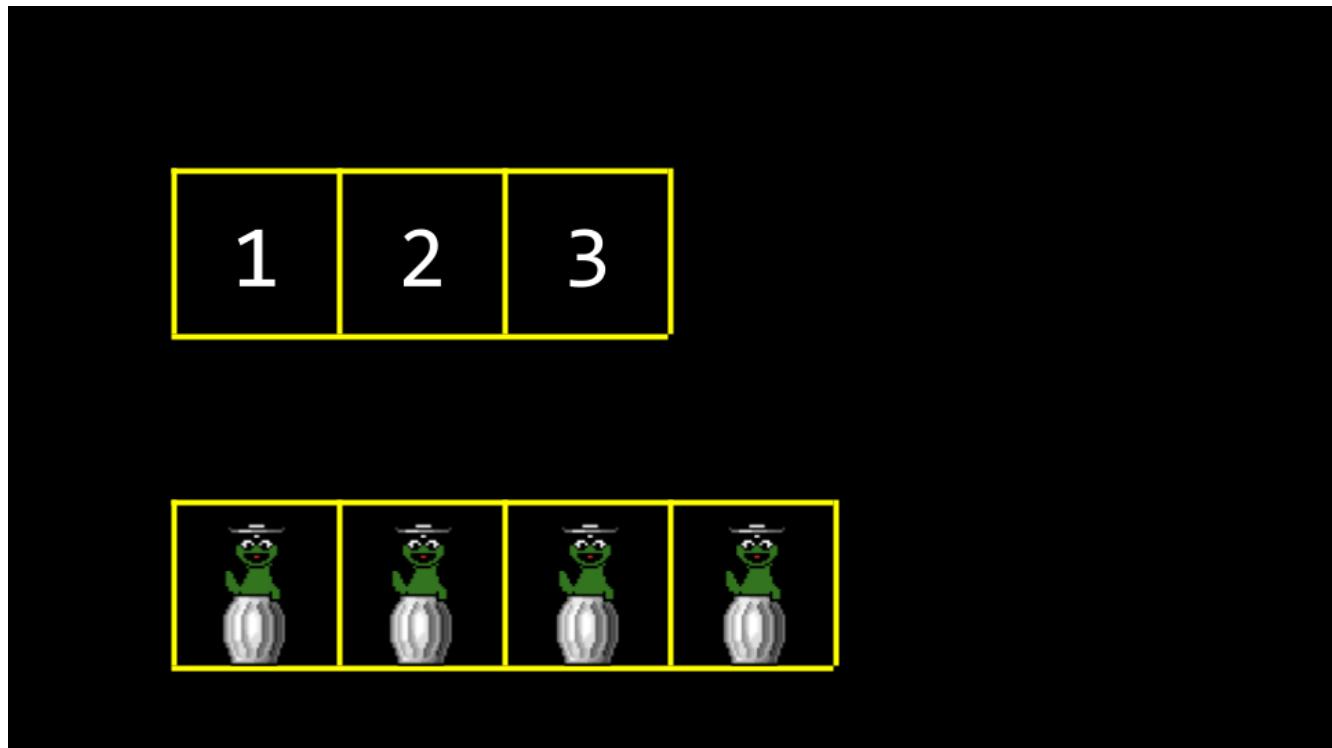
You might imagine an array as follows:

1	2	3
---	---	---

In memory, there are other values being stored by other programs, functions, and variables. Many of these may be unused garbage values that were utilized at one point but are available now for use.

							
	1	2	3	h	e	l	l
o	,		w	o	r	l	d
\0							
							

Imagine you wanted to store a fourth value 4 in our array? What would be needed is to allocate a new area of memory and move the old array to a new one. Initially, this new area of memory would be populated with garbage values.



As values are added to this new area of memory, old garbage values would be overwritten.

1	2	3
---	---	---

1	2	3	
---	---	---	---

- Eventually, all old garbage values would be overwritten with our new data.

1	2	3
---	---	---

1	2	3	4
---	---	---	---

- One of the drawbacks of this approach is that it's bad design: Every time we add a number, we have to copy the array item by item.
- Wouldn't it be nice if we were able to put the 4 somewhere else in memory? By definition, this would no longer be an array because 4 would no longer be in contiguous memory.
- In your terminal, type `code list.c` and write code as follows:

```
code list.c
```

```

1 // Implements a list of numbers with an array of fixed size
2
3 #include <stdio.h>
4
5 int main(void)
6 {
7     // List of size 3
8     int list[3];
9
10    // Initialize list with numbers
11    list[0] = 1;
12    list[1] = 2;
13    list[2] = 3;
14
15    // Print list
16    for (int i = 0; i < 3; i++)
17    {
18        printf("%i\n", list[i]);
19    }
20}

```

Notice that the above is very much like what we learned earlier in this course. We have memory being preallocated for three items.

- Building upon our knowledge obtained more recently, we can leverage our understanding of pointers to create a better design in this code. Modify your code as follows:

```

1 // Implements a list of numbers with an array of dynamic size
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8     // List of size 3
9     int *list = malloc(3 * sizeof(int));
10    if (list == NULL)
11    {
12        return 1;
13    }
14
15    // Initialize list of size 3 with numbers
16    list[0] = 1;
17    list[1] = 2;
18    list[2] = 3;
19
20    // List of size 4
21    int *tmp = malloc(4 * sizeof(int));
22    if (tmp == NULL)
23    {

```

```

24         free(list);
25         return 1;
26     }
27
28     // Copy list of size 3 into list of size 4
29     for (int i = 0; i < 3; i++)
30     {
31         tmp[i] = list[i];
32     }
33
34     // Add number to list of size 4
35     tmp[3] = 4;
36
37     // Free list of size 3
38     free(list);
39
40     // Remember list of size 4
41     list = tmp;
42
43     // Print list
44     for (int i = 0; i < 4; i++)
45     {
46         printf("%i\n", list[i]);
47     }
48
49     // Free list
50     free(list);
51     return 0;
52 }
```

Notice that a list of size three integers is created. Then, three memory addresses can be assigned the values `1`, `2`, and `3`. Then, a list of size four is created. Next, the list is copied from the first to the second. The value for the `4` is added to the `tmp` list. Since the block of memory that `list` points to is no longer used, it is freed using the command `free(list)`. Finally, the compiler is told to point `list` pointer now to the block of memory that `tmp` points to. The contents of `list` are printed and then freed.

- It's useful to think about `list` and `tmp` as both signs that point at a chunk of memory. As in the example above, `list` at one point *pointed* to an array of size 3. By the end, `list` was told to point to a chunk of memory of size 4. Technically, by the end of the above code, `tmp` and `list` both pointed to the same block of memory.
- One may be tempted to allocate way more memory than required for the list, such as 30 items instead of the required 3 or 4. However, this is bad design as it taxes system resources when they are not potentially needed. Further, there is little guarantee that memory for more than 30 items will be needed eventually.

Linked Lists

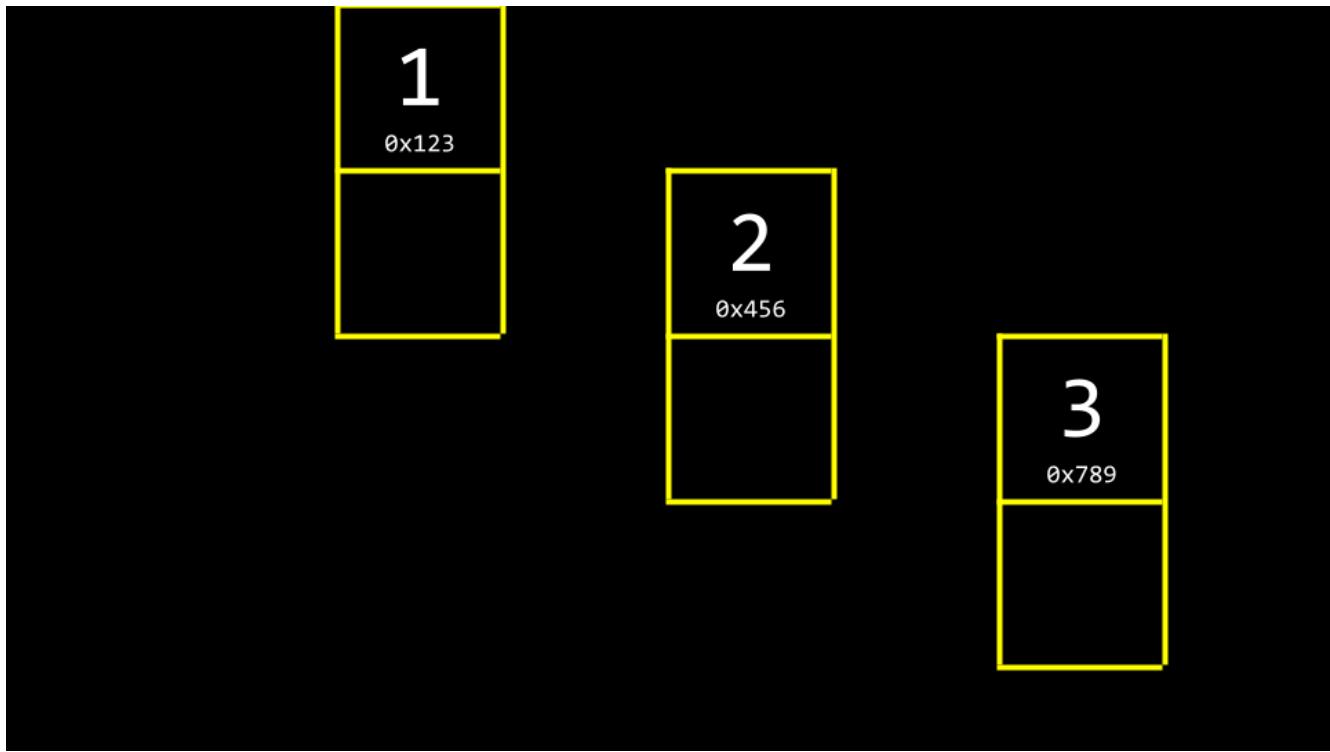
- In recent weeks, you have learned about three useful primitives. A `struct` is a data type that you can define yourself. A `.` in *dot notation* allows you to access variables inside that structure. The `*` operator is used to declare a pointer or dereference a variable.
- Today, you are introduced to the `->` operator. It is an arrow. This operator goes to an address and looks inside of a structure.
- A *linked list* is one of the most powerful data structures within C. A linked list allows you to include values that are located at varying areas of memory. Further, they allow you to dynamically grow and shrink the list as you desire.
-

You might imagine three values stored at three different areas of memory as follows:

		1 0x123					
				2 0x456			
					3 0x789		

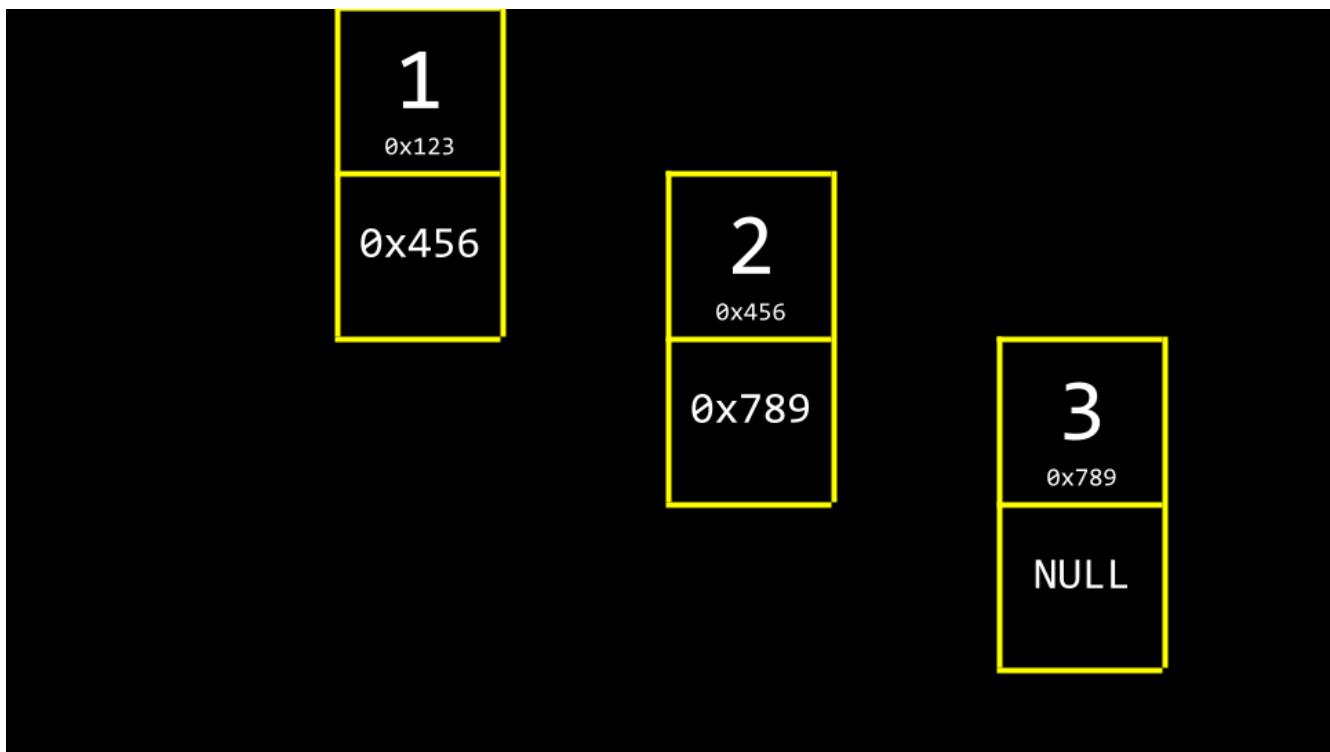
- How could one stitch together these values in a list?
-

We could imagine this data pictured above as follows:



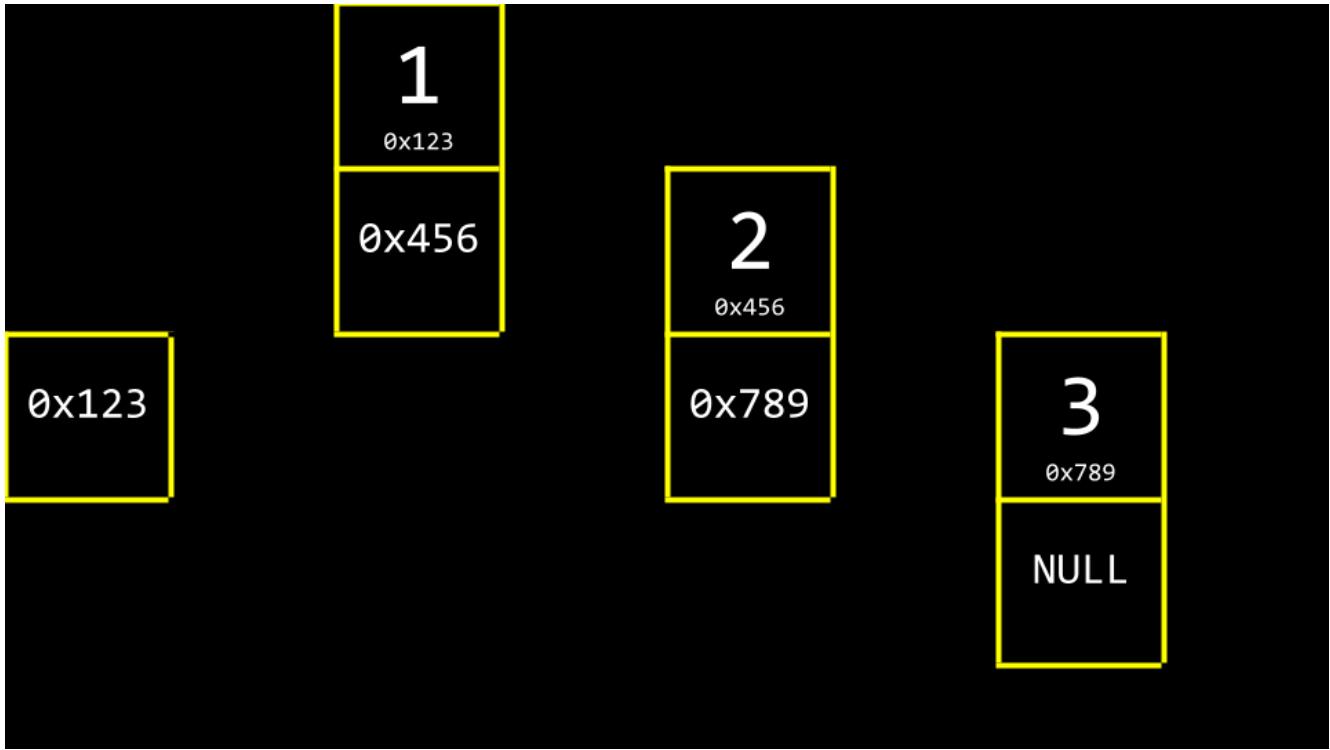
-

We could utilize more memory to keep track of where the next item is.

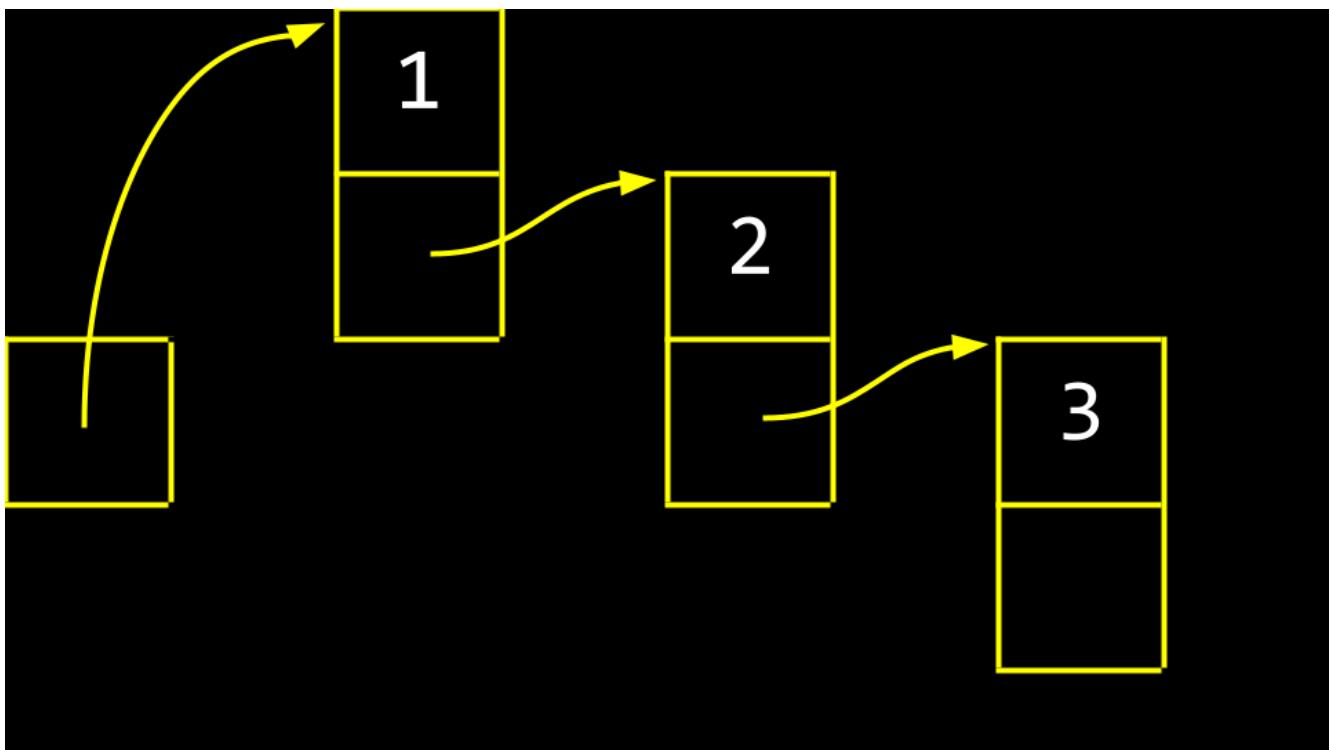


-

By convention, we would keep one more element in memory, a pointer, that keeps track of the first item in the list.



- Abstracting away the memory addresses, the list would appear as follows:



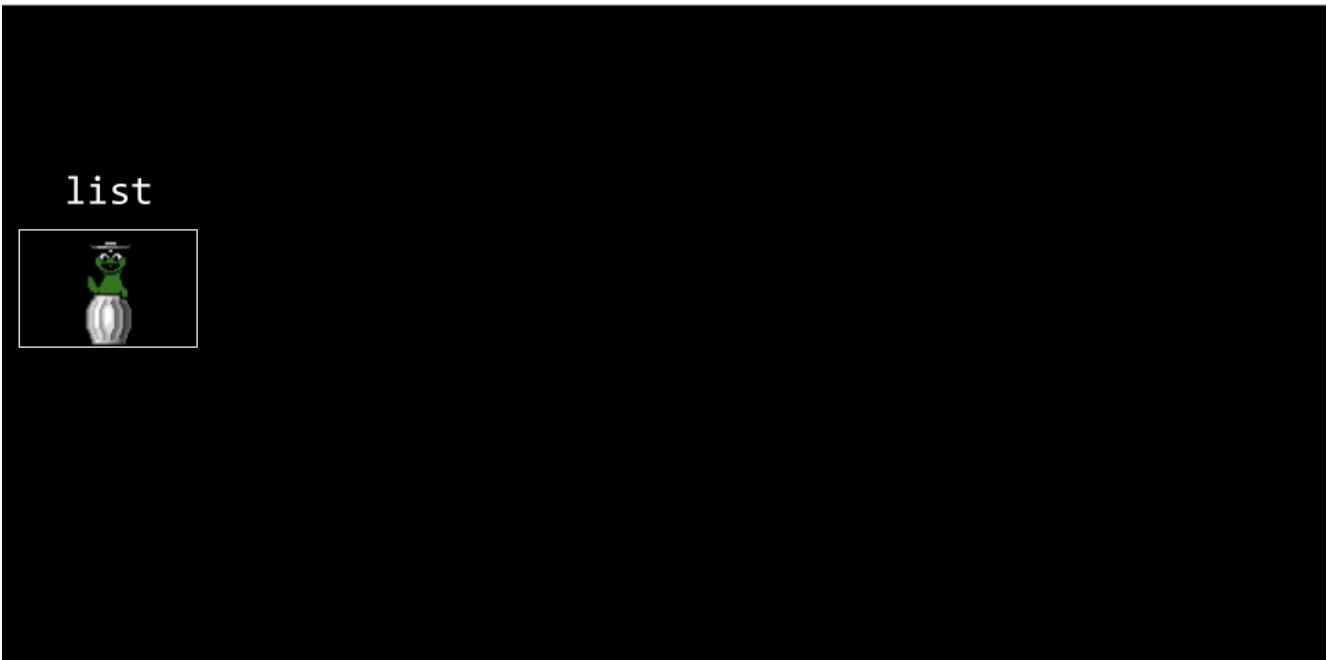
- These boxes are called *nodes*. A *node* contains both an *item* and a pointer called *next*. In code, you can imagine a node as follows:

```
1  typedef struct node
2  {
3      int number;
4      struct node *next;
5  }
6  node;
```

Notice that the item contained within this node is an integer called `number`. Second, a pointer to a node called `next` is included, which will point to another node somewhere in memory.

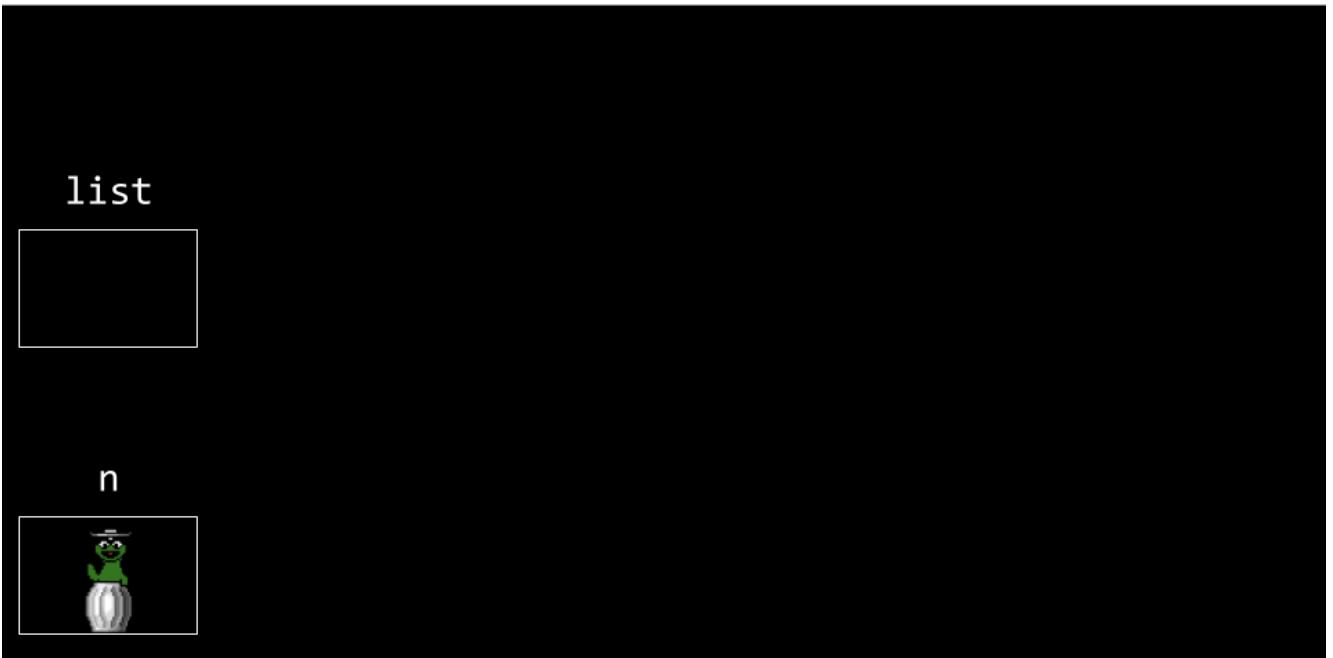
- Conceptually, we can imagine the process of creating a linked list. First, `node *list` is declared, but it is of a garbage value.

`node *list;`



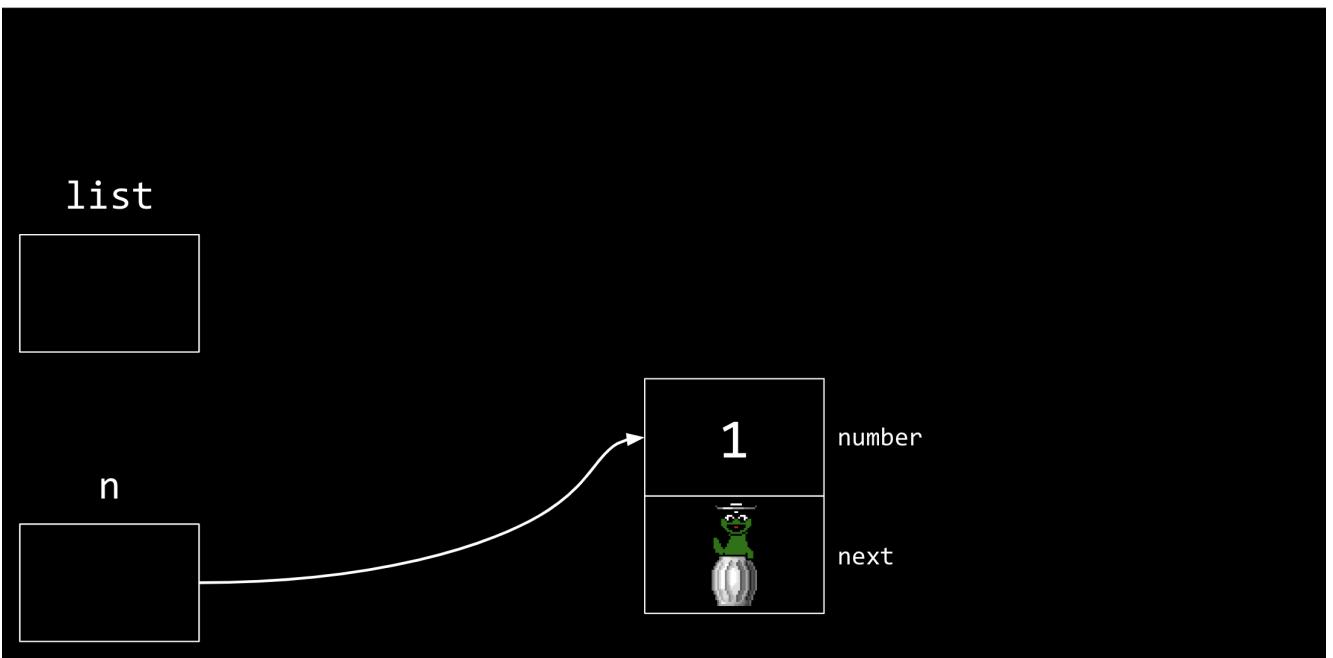
- Next, a node called `n` is allocated in memory.

```
node *n = malloc(sizeof(node));
```



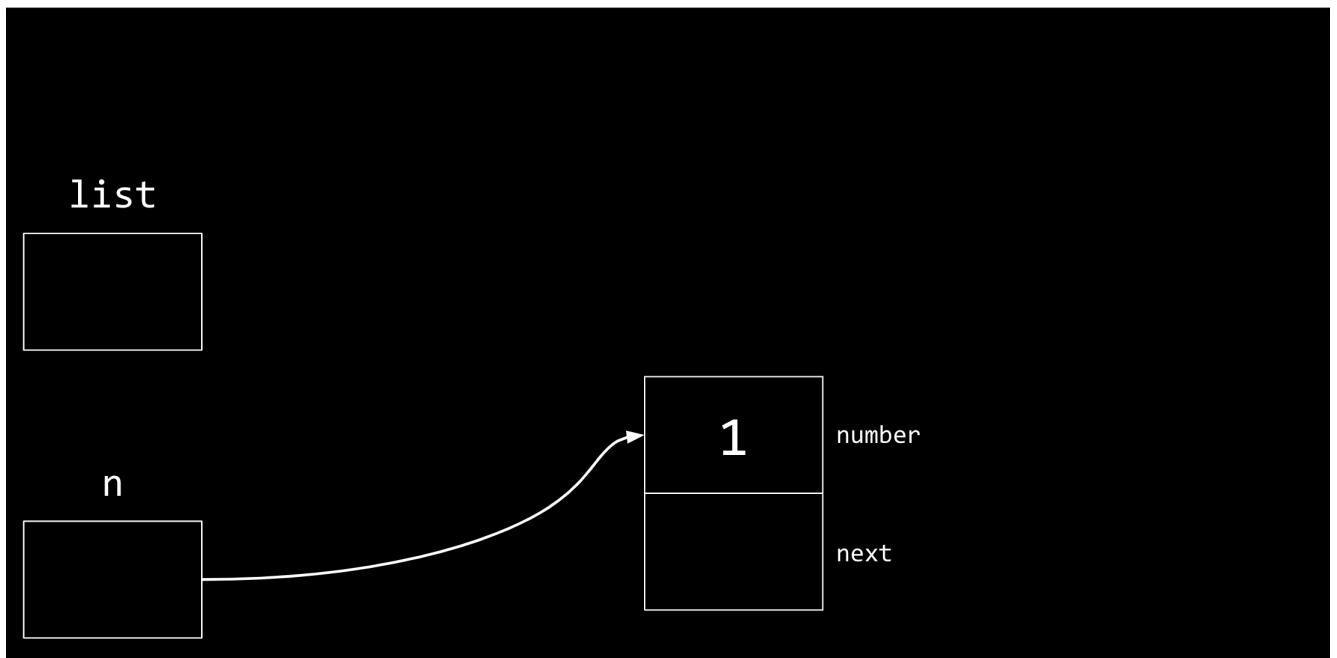
- Next, the `number` of node is assigned the value `1`.

```
n->number = 1;
```



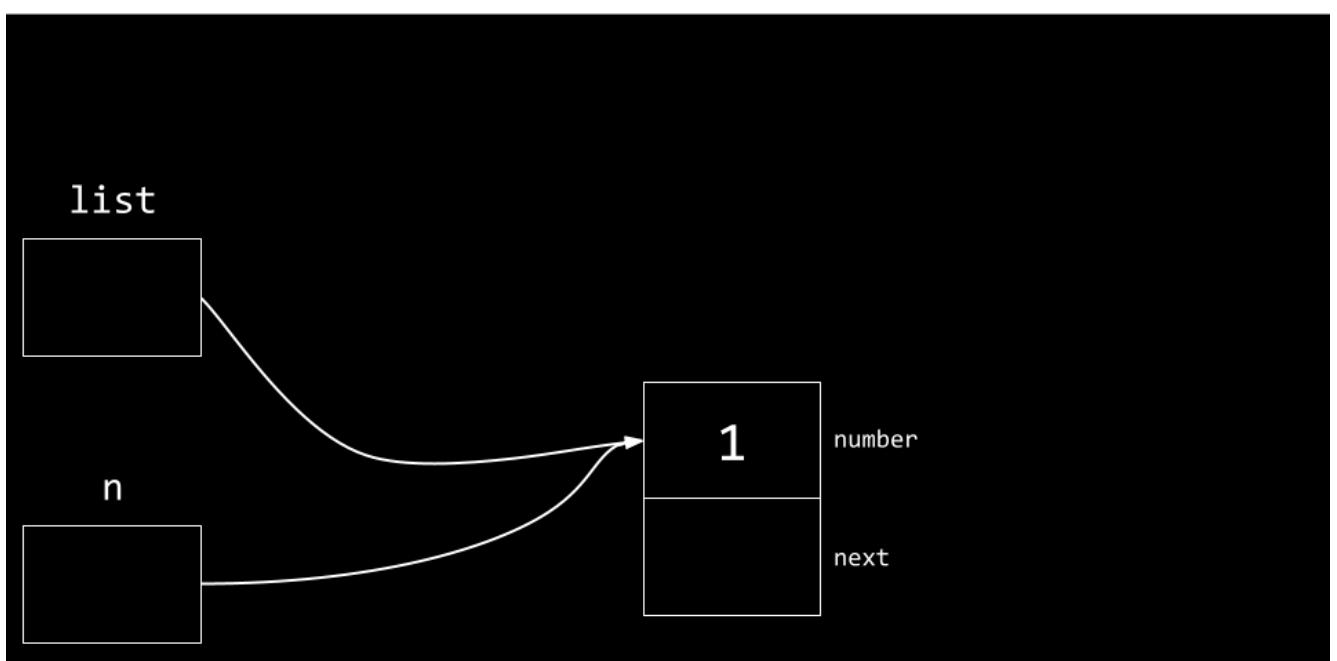
- Next, the node's `next` field is assigned `NULL`.

```
n->next = NULL;
```



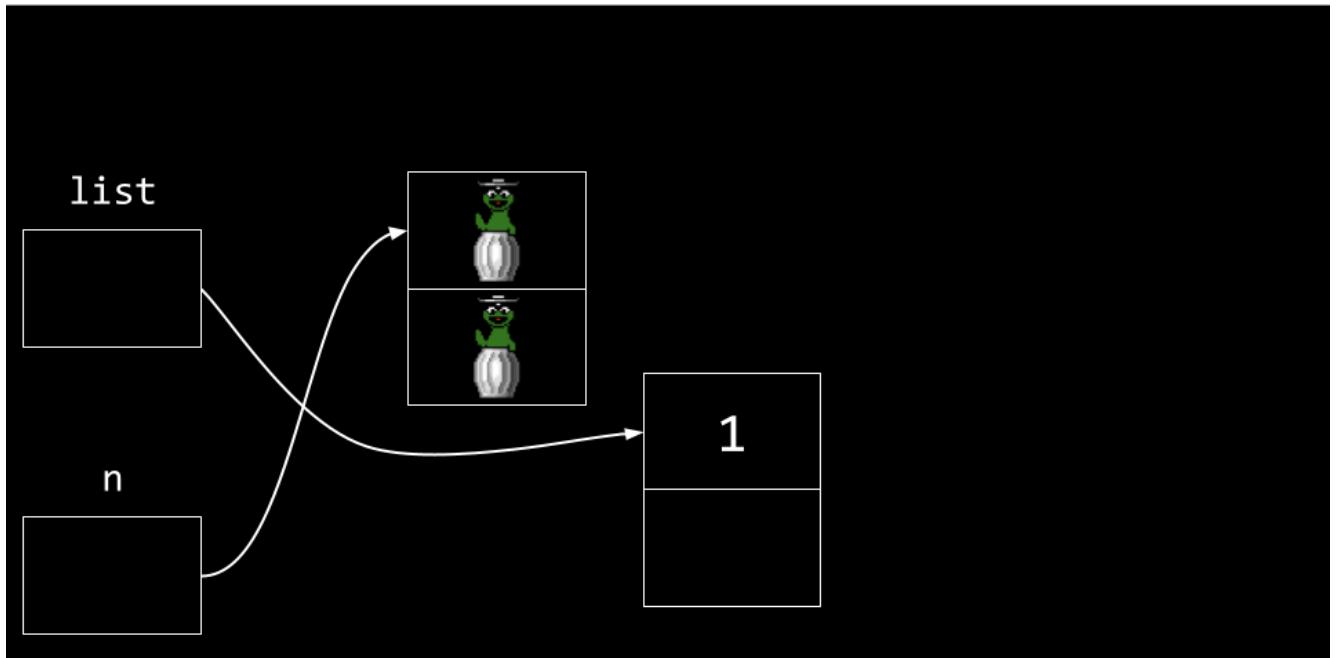
- Next, `list` is pointed at the memory location to where `n` points. `n` and `list` now point to the same place.

```
list = n;
```



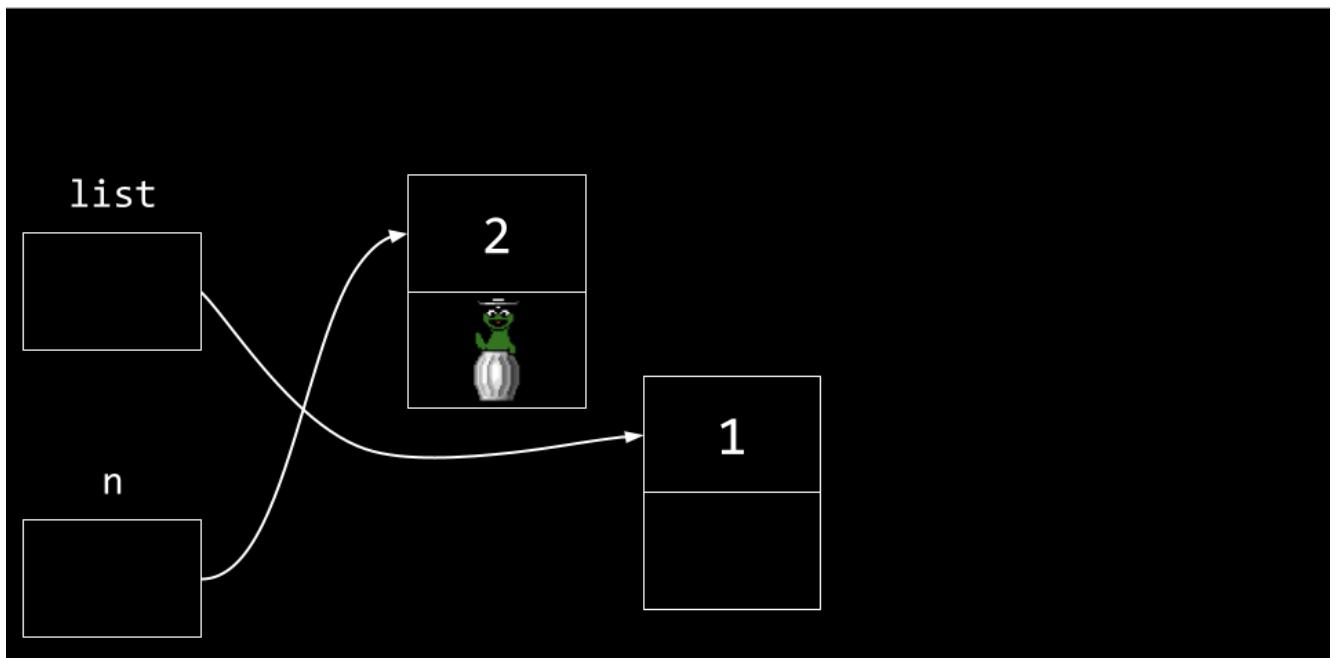
- A new node is then created. Both the `number` and `next` field are both filled with garbage values.

```
node *n = malloc(sizeof(node));
```



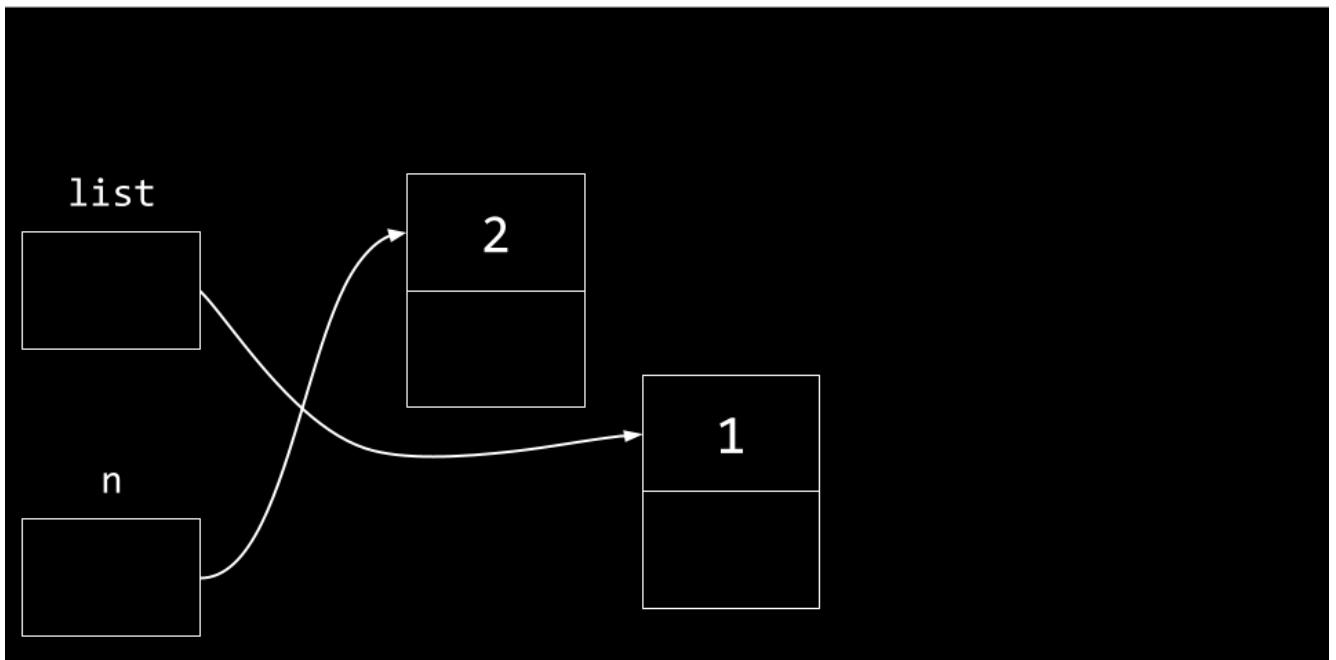
- The `number` value of `n`'s node (the new node) is updated to `2`.

```
n->number = 2;
```



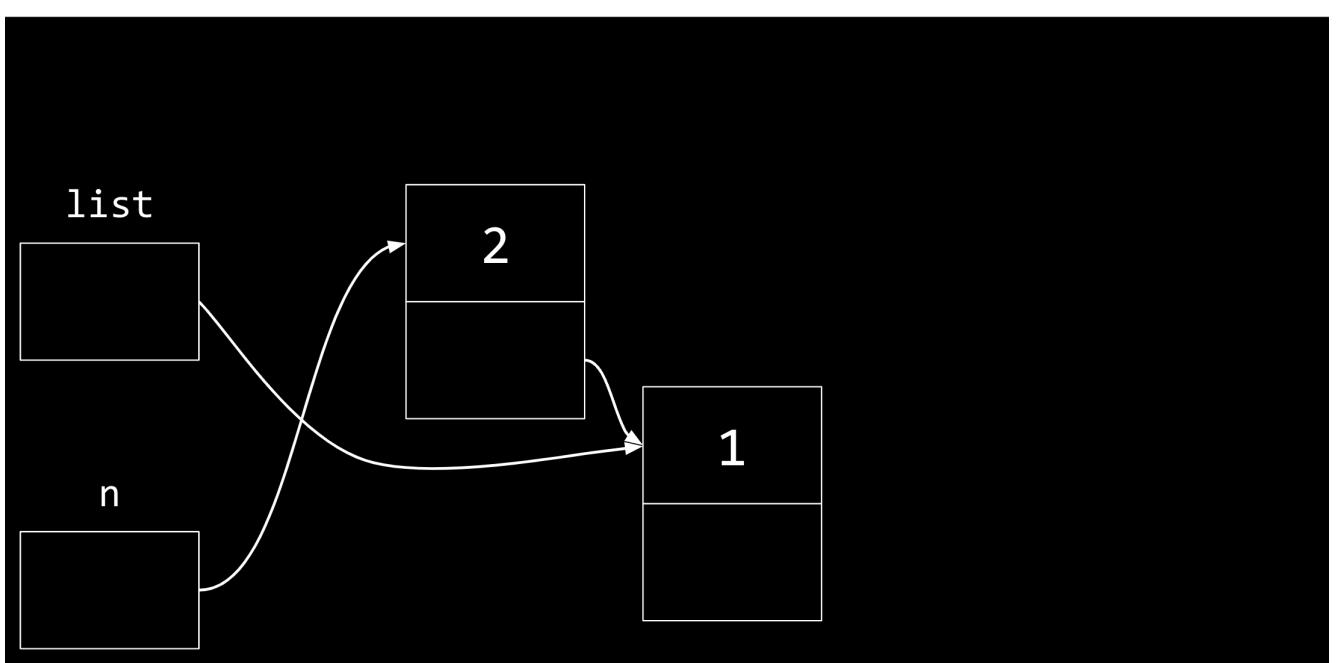
- Also, the `next` field is updated as well.

```
n->next = NULL;
```



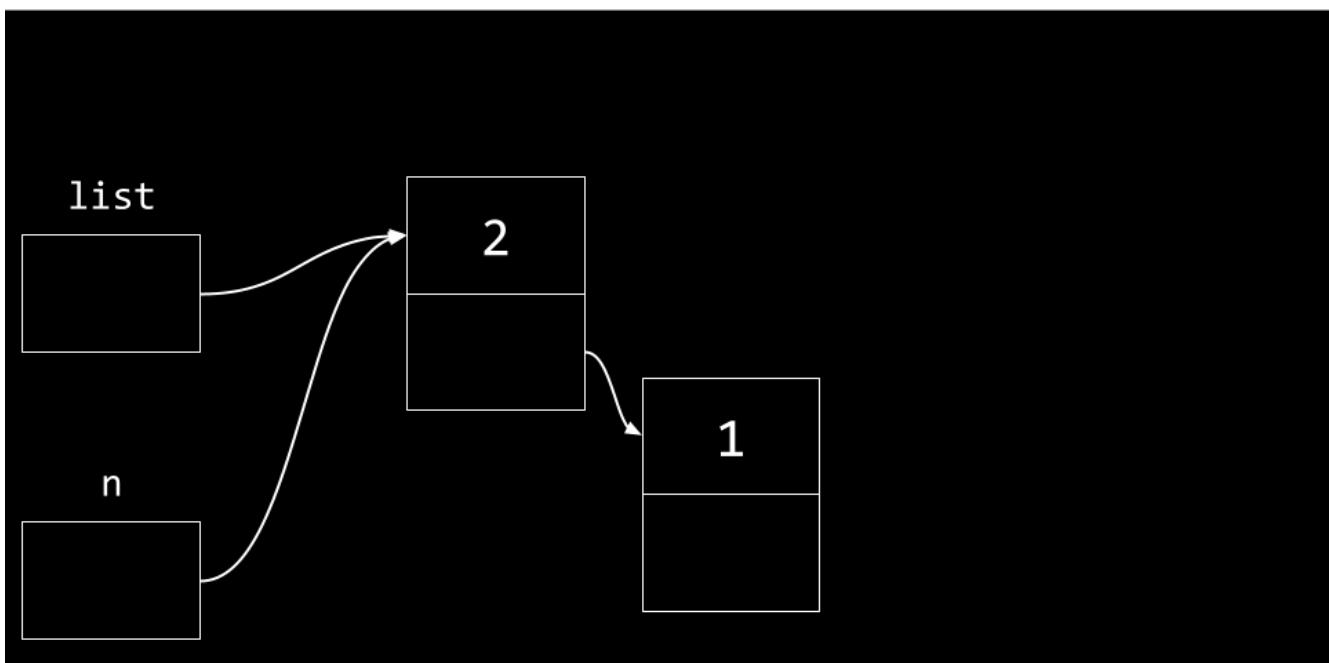
- Most important, we do not want to lose our connection to any of these nodes lest they be lost forever. Accordingly, `n`'s `next` field is pointed to the same memory location as `list`.

```
n->next = list;
```



- Finally, `list` is updated to point at `n`. We now have a linked list of two items.

```
list = n;
```



- To implement this in code, modify your code as follows:

```
1 // Prepends numbers to a linked list, using while loop to print
2
3 #include <cs50.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 typedef struct node
8 {
9     int number;
10    struct node *next;
11 }
12 node;
13
14 int main(int argc, char *argv[])
15 {
16     // Memory for numbers
17     node *list = NULL;
18
19     // For each command-line argument
20     for (int i = 1; i < argc; i++)
21     {
22         // Convert argument to int
23         int number = atoi(argv[i]);
24
25         // Allocate node for number
26         node *n = malloc(sizeof(node));
27         if (n == NULL)
28         {
```

```

29         return 1;
30     }
31     n->number = number;
32     n->next = NULL;
33
34     // Prepend node to list
35     n->next = list;
36     list = n;
37 }
38
39 // Print numbers
40 node *ptr = list;
41 while (ptr != NULL)
42 {
43     printf("%i\n", ptr->number);
44     ptr = ptr->next;
45 }
46
47 // Free memory
48 ptr = list;
49 while (ptr != NULL)
50 {
51     node *next = ptr->next;
52     free(ptr);
53     ptr = next;
54 }
55 }
```

Notice that what the user inputs at the command line is put into the `number` field of a node called `n`, and then that node is added to the `list`. For example, `./list 1 2` will put the number `1` into the `number` field of a node called `n`, then put a pointer to `list` into the `next` field of the node, and then update `list` to point to `n`. That same process is repeated for `2`. Next, `node *ptr = list` creates a temporary variable that points at the same spot that `list` points to. The `while` prints what at the node `ptr` points to, and then updates `ptr` to point to the `next` node in the list. Finally, all the memory is freed.

- In this example, inserting into the list is always in the order of $O(1)$, as it only takes a very small number of steps to insert at the front of a list.
- Considering the amount of time required to search this list, it is in the order of $O(n)$, as in the worst case the entire list must always be searched to find an item. The time complexity for adding a new element to the list will depend on where that element is added. This is illustrated in the examples below.
- Linked lists are not stored in a contiguous block of memory. They can grow as large as you wish, provided that enough system resources exist. The downside, however, is that more memory is required to keep track of the list instead of an array. This is because for each element, you must store not just the value of the element, but also a pointer to the next node. Further, linked lists cannot be indexed into like is possible in an array because we need to pass through the first $n-1$ elements to find the location of the n th element. Because of this, the list pictured above must be linearly searched. Binary search, therefore, is not possible in a list constructed as above.

Further, you could place numbers at the end of the list as illustrated in this code:

```
1 // Implements a list of numbers using a linked list
2
3 #include <cs50.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 typedef struct node
8 {
9     int number;
10    struct node *next;
11 }
12 node;
13
14 int main(int argc, char *argv[])
15 {
16     // Memory for numbers
17     node *list = NULL;
18
19     // For each command-line argument
20     for (int i = 1; i < argc; i++)
21     {
22         // Convert argument to int
23         int number = atoi(argv[i]);
24
25         // Allocate node for number
26         node *n = malloc(sizeof(node));
27         if (n == NULL)
28         {
29             return 1;
30         }
31         n->number = number;
32         n->next = NULL;
33
34         // If list is empty
35         if (list == NULL)
36         {
37             // This node is the whole list
38             list = n;
39         }
40
41         // If list has numbers already
42         else
43         {
44             // Iterate over nodes in list
45             for (node *ptr = list; ptr != NULL; ptr = ptr->next)
46             {
47                 // If at end of list
48                 if (ptr->next == NULL)
```

```

49         {
50             // Append node
51             ptr->next = n;
52             break;
53         }
54     }
55 }
56
57 // Print numbers
58 for (node *ptr = list; ptr != NULL; ptr = ptr->next)
59 {
60     printf("%i\n", ptr->number);
61 }
62
63
64 // Free memory
65 node *ptr = list;
66 while (ptr != NULL)
67 {
68     node *next = ptr->next;
69     free(ptr);
70     ptr = next;
71 }
72 }
```

Notice how this code *walks down* this list to find the end. When appending an element, (adding to the end of the list) our code will run in O(n), as we have to go through our entire list before we can add the final element.

- Further, you could sort your list as items are added:

```

1 // Implements a sorted list of numbers using a linked list
2
3 #include <cs50.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 typedef struct node
8 {
9     int number;
10    struct node *next;
11 }
12 node;
13
14 int main(int argc, char *argv[])
15 {
16     // Memory for numbers
17     node *list = NULL;
18
19     // For each command-line argument
```

```
20  for (int i = 1; i < argc; i++)
21  {
22      // Convert argument to int
23      int number = atoi(argv[i]);
24
25      // Allocate node for number
26      node *n = malloc(sizeof(node));
27      if (n == NULL)
28      {
29          return 1;
30      }
31      n->number = number;
32      n->next = NULL;
33
34      // If list is empty
35      if (list == NULL)
36      {
37          list = n;
38      }
39
40      // If number belongs at beginning of list
41      else if (n->number < list->number)
42      {
43          n->next = list;
44          list = n;
45      }
46
47      // If number belongs later in list
48      else
49      {
50          // Iterate over nodes in list
51          for (node *ptr = list; ptr != NULL; ptr = ptr->next)
52          {
53              // If at end of list
54              if (ptr->next == NULL)
55              {
56                  // Append node
57                  ptr->next = n;
58                  break;
59              }
60
61              // If in middle of list
62              if (n->number < ptr->next->number)
63              {
64                  n->next = ptr->next;
65                  ptr->next = n;
66                  break;
67              }
68          }
69      }
70  }
```

```

72 // Print numbers
73 for (node *ptr = list; ptr != NULL; ptr = ptr->next)
74 {
75     printf("%i\n", ptr->number);
76 }
77
78 // Free memory
79 node *ptr = list;
80 while (ptr != NULL)
81 {
82     node *next = ptr->next;
83     free(ptr);
84     ptr = next;
85 }
86 }
```

Notice how this list is sorted as it is built. To insert an element in this specific order, our code will still run in O(n) for each insertion, as in the worst case we will have to look through all current elements.

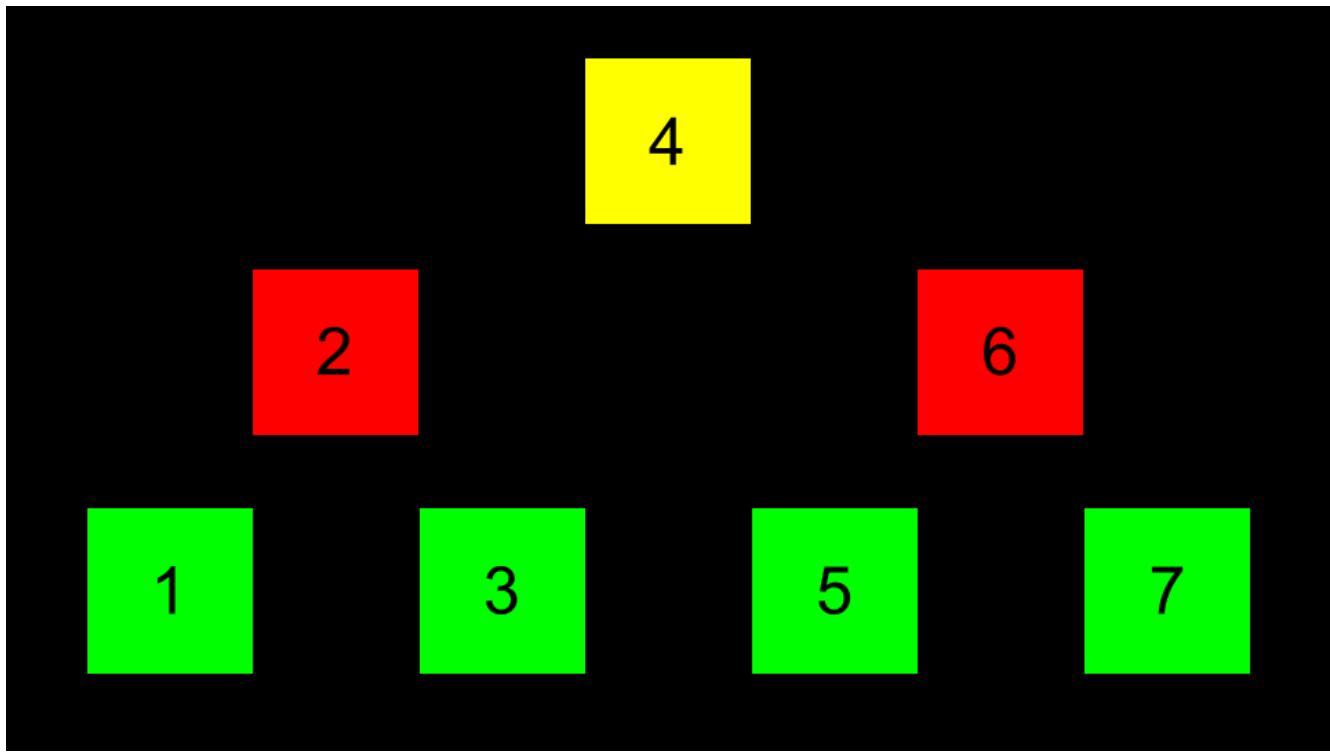
Trees

- *Binary search trees* are another data structure that can be used to store data more efficiently such that it can be searched and retrieved.
-

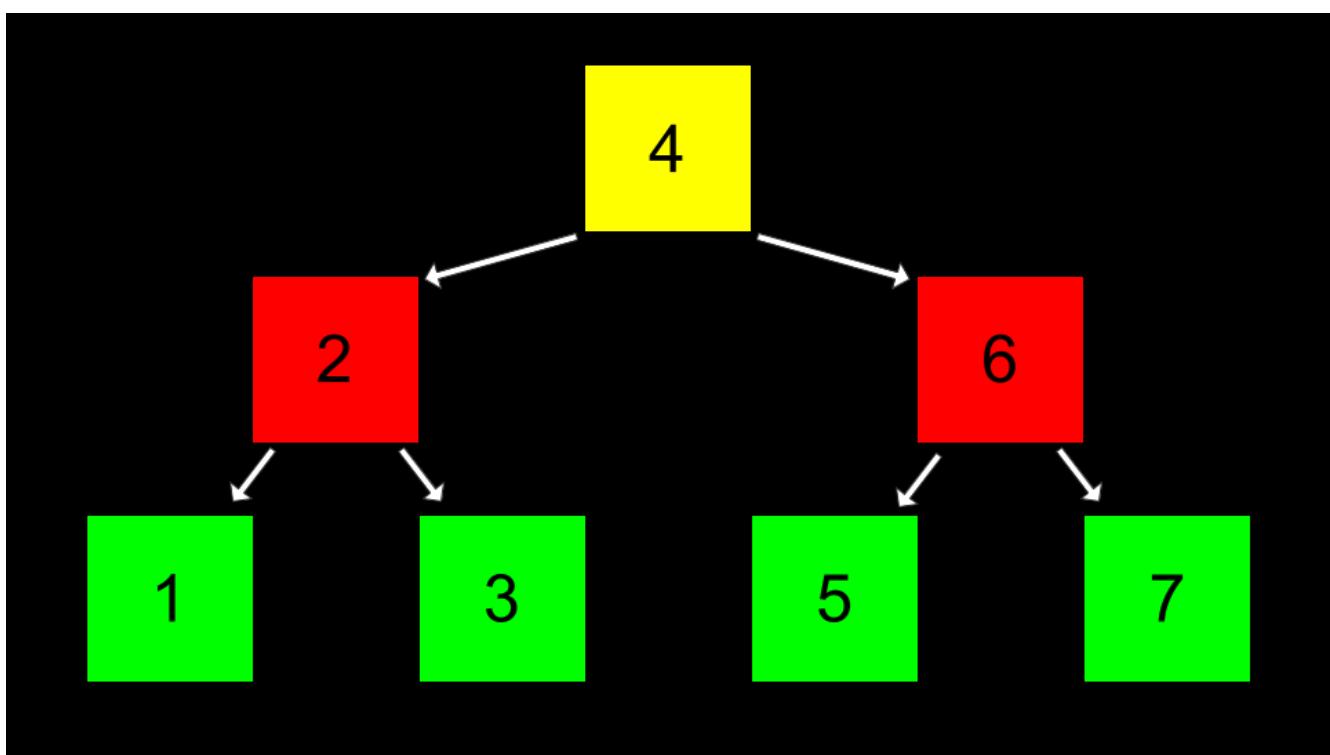
You can imagine a sorted sequence of numbers.



- Imagine then that the center value becomes the top of a tree. Those that are less than this value are placed to the left. Those values that are more than this value are to the right.



- Pointers can then be used to point to the correct location of each area of memory such that each of these nodes can be connected.



- In code, this can be implemented as follows.

```
1 // Implements a list of numbers as a binary search tree
2
3 #include <stdio.h>
4 #include <stdlib.h>
```

```
5 // Represents a node
6 typedef struct node
7 {
8     int number;
9     struct node *left;
10    struct node *right;
11 }
12 }
13 node;
14
15 void free_tree(node *root);
16 void print_tree(node *root);
17
18 int main(void)
19 {
20     // Tree of size 0
21     node *tree = NULL;
22
23     // Add number to list
24     node *n = malloc(sizeof(node));
25     if (n == NULL)
26     {
27         return 1;
28     }
29     n->number = 2;
30     n->left = NULL;
31     n->right = NULL;
32     tree = n;
33
34     // Add number to list
35     n = malloc(sizeof(node));
36     if (n == NULL)
37     {
38         free_tree(tree);
39         return 1;
40     }
41     n->number = 1;
42     n->left = NULL;
43     n->right = NULL;
44     tree->left = n;
45
46     // Add number to list
47     n = malloc(sizeof(node));
48     if (n == NULL)
49     {
50         free_tree(tree);
51         return 1;
52     }
53     n->number = 3;
54     n->left = NULL;
55     n->right = NULL;
56     tree->right = n;
```

```

57
58     // Print tree
59     print_tree(tree);
60
61     // Free tree
62     free_tree(tree);
63     return 0;
64 }
65
66 void free_tree(node *root)
67 {
68     if (root == NULL)
69     {
70         return;
71     }
72     free_tree(root->left);
73     free_tree(root->right);
74     free(root);
75 }
76
77 void print_tree(node *root)
78 {
79     if (root == NULL)
80     {
81         return;
82     }
83     print_tree(root->left);
84     printf("%i\n", root->number);
85     print_tree(root->right);
86 }
```

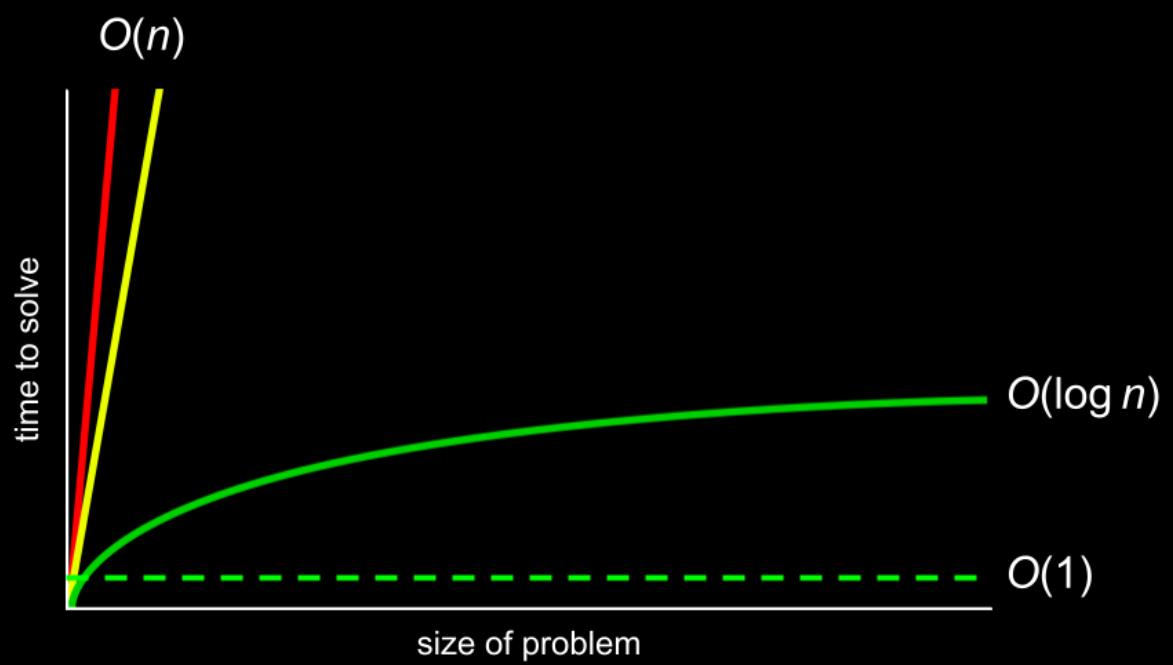
Notice this search function begins by going to the location of `tree`. Then, it uses recursion to search for `number`. The `free_tree` function recursively frees the tree. `print_tree` recursively prints the tree.

- A tree like the above offers dynamism that an array does not offer. It can grow and shrink as we wish.
- Further, this structure offers a search time of $O(\log n)$.

Dictionaries

- *Dictionaries* are another data structure.
- Dictionaries, like actual book-form dictionaries that have a word and a definition, have a *key* and a *value*.
-

The *holy grail* of algorithmic time complexity is $O(1)$ or *constant time*. That is, the ultimate is for access to be instantaneous.



- Dictionaries can offer this speed of access through hashing.

Hashing and Hash Tables

- *Hashing* is the idea of taking a value and being able to output a value that becomes a shortcut to it later.
- For example, hashing *apple* may hash as a value of `1`, and *berry* may be hashed as `2`. Therefore, finding *apple* is as easy as asking the *hash* algorithm where *apple* is stored. While not ideal in terms of design, ultimately, putting all *a*'s in one bucket and *b*'s in another, this concept of *bucketizing* hashed values illustrates how you can use this concept: a hashed value can be used to shortcut finding such a value.
- A *hash function* is an algorithm that reduces a larger value to something small and predictable. Generally, this function takes in an item you wish to add to your hash table, and returns an integer representing the array index in which the item should be placed.
- A *hash table* is a fantastic combination of both arrays and linked lists. When implemented in code, a hash table is an *array of pointers to nodes*.
-

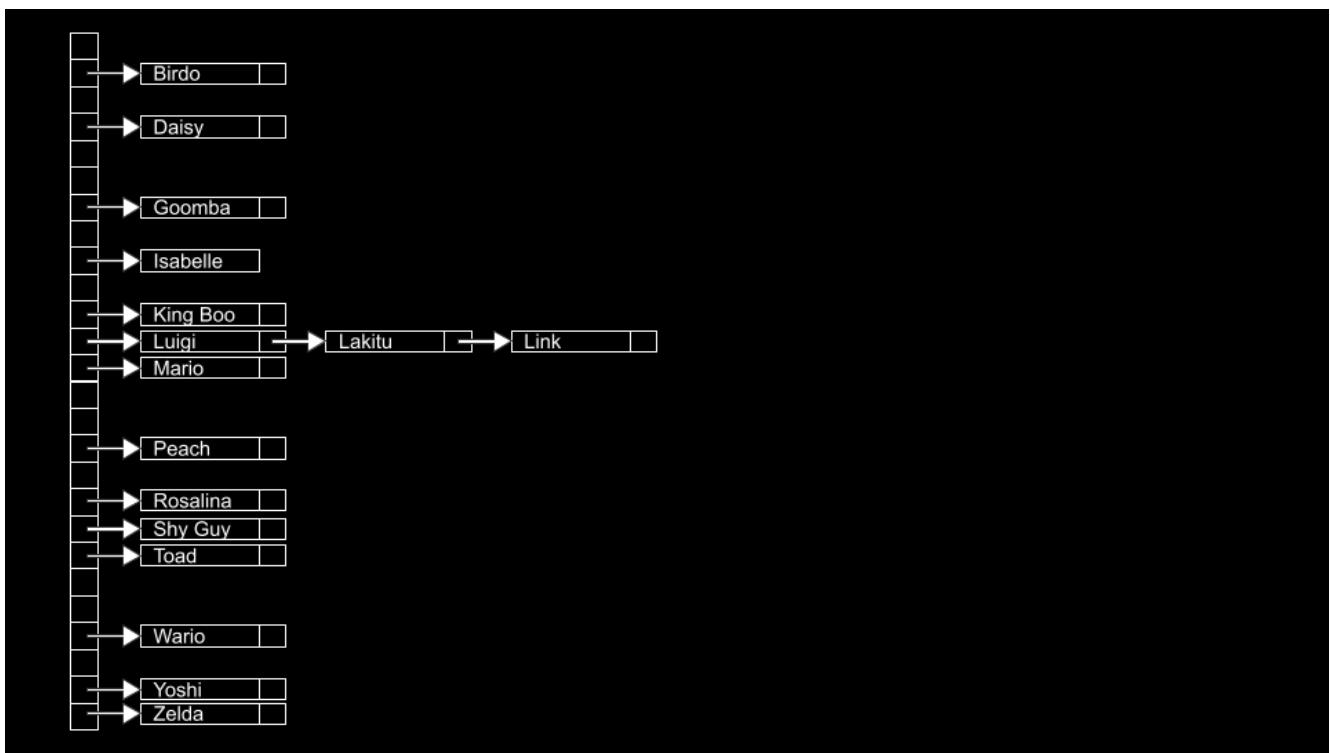
A hash table could be imagined as follows:

A	
B	
C	
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	
N	
O	
P	
Q	
R	
S	
T	
U	
V	
W	
X	
Y	
Z	

Notice that this is an array that is assigned each value of the alphabet.

-

Then, at each location of the array, a linked list is used to track each value being stored there:

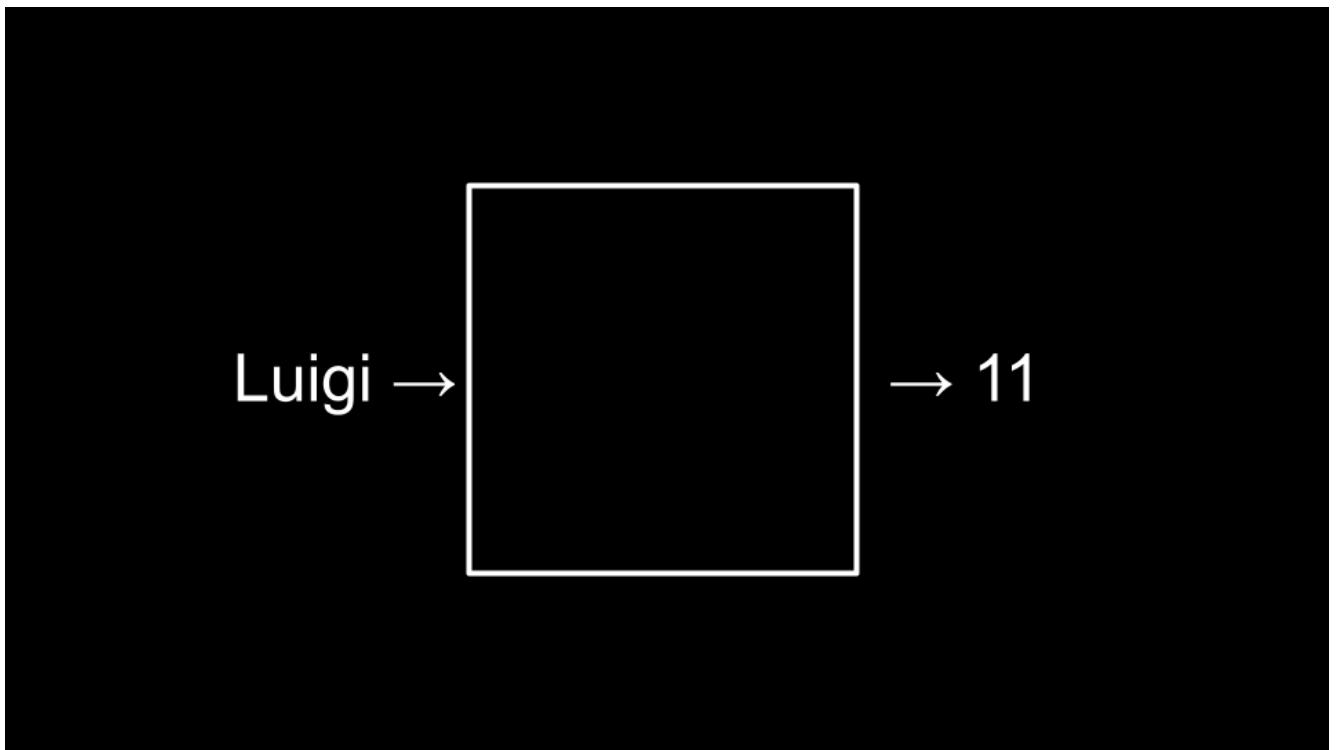


- *Collisions* are when you add values to the hash table, and something already exists at the hashed location. In the above, collisions are simply appended to the end of the list.
-

Collisions can be reduced by better programming your hash table and hash algorithm. You can imagine an improvement upon the above as follows:



- Consider the following example of a hash algorithm:



- This could be implemented in code as:

```

1 #include <ctype.h>
2
3 unsigned int hash(const char *word)
4 {
5     return toupper(word[0]) - 'A';
6 }
```

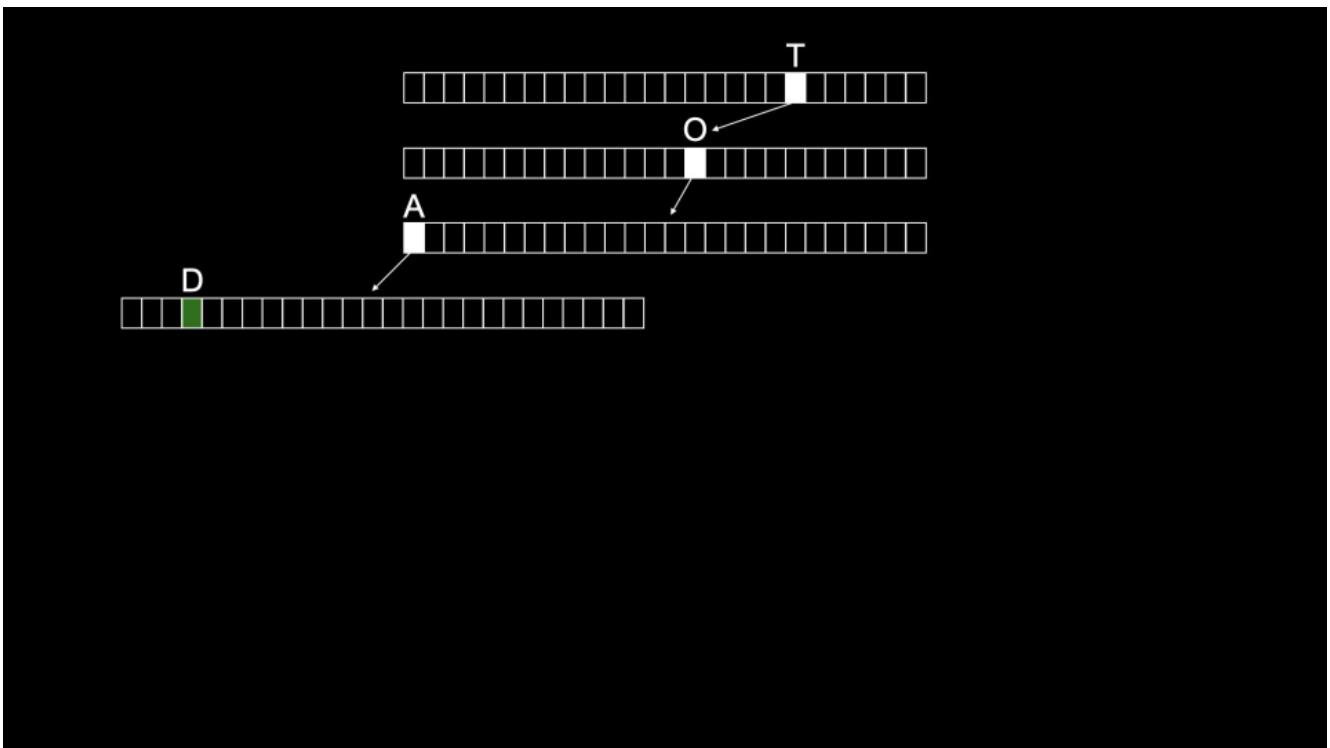
Notice how the hash function returns the value of `toupper(word[0]) - 'A'`.

- You, as the programmer, have to make a decision about the advantages of using more memory to have a large hash table and potentially reducing search time or using less memory and potentially increasing search time.

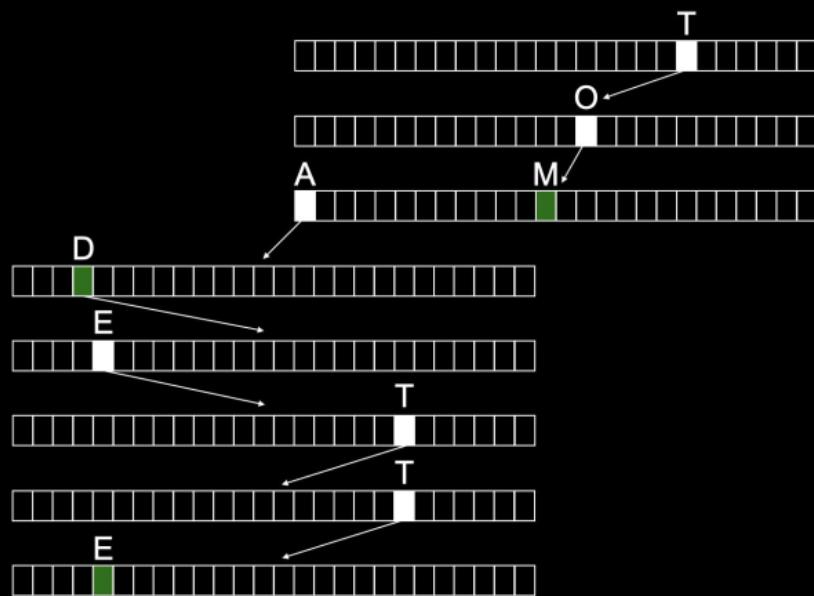
Tries

- *Tries* are another form of data structure.
- *Tries* are always searchable in constant time.
- One downside to *Tries* is that they tend to take up a large amount of memory. Notice that we need $26 \times 4 = 104$ nodes just to store *Toad*!
-

Toad would be stored as follows:



Tom would then be stored as follows:



- The downside of this structure is how many resources are required to use it.

Summing Up

In this lesson, you learned about using pointers to build new data structures. Specifically, we delved into...

- Data structures
- Stacks and queues
- Resizing arrays
- Linked lists
- Dictionaries
- Tries

See you next time!

Lecture 6-Python

- [Welcome!](#)
- [Python](#)
- [Hello](#)
- [Speller](#)
- [Filter](#)
- [CS50 Library](#)

- [Strings](#)
- [Variables](#)
- [Types](#)
- [Calculator](#)
- [Conditionals](#)
- [Object-Oriented Programming](#)
- [Loops](#)
- [Abstraction](#)
- [Truncation and Floating Point Imprecision](#)
- [Exceptions](#)
- [Mario](#)
- [Lists](#)
- [Searching and Dictionaries](#)
- [Command-Line Arguments](#)
- [Exit Status](#)
- [Third-Party Libraries](#)
- [Summing Up](#)

Welcome!

- In previous weeks, you were introduced to the fundamental building blocks of programming.
- You learned about programming in a lower-level programming language called C.
- Today, we are going to work with a higher-level programming language called *Python*.
- As you learn this new language, you're going to find that you are going to be more able to teach yourself new programming languages.

Python

- Humans, over the decades, have seen how previous design decisions could be improved upon.
- Python is a programming language that builds upon what you have already learned in C.
- Unlike in C, Python is an interpreted language, where you need not separately compile your program. Instead, you run your program in the *Python Interpreter*.

Hello

- Up until this point, the code has looked like this:

```
1 // A program that says hello to the world
2
3 #include <stdio.h>
4
5 int main(void)
6 {
7     printf("hello, world\n");
8 }
```

- Today, you'll find that the process of writing and compiling code has been simplified.
- For example, the above code will be rendered in Python as:

```
1 # A program that says hello to the world
2
3 print("hello, world")
```

Notice that the semicolon is gone and that no library is needed.

- Python notably can implement what was quite complicated in C with relative simplicity.

Speller

- To illustrate this simplicity, let's type 'code dictionary.py' in the terminal window and write code as follows:

```
1 # Words in dictionary
2 words = set()
3
4
5 def check(word):
6     """Return true if word is in dictionary else false"""
7     return word.lower() in words
8
9
10 def load(dictionary):
11     """Load dictionary into memory, returning true if successful else false"""
12     with open(dictionary) as file:
13         words.update(file.read().splitlines())
14     return True
15
16
17 def size():
18     """Returns number of words in dictionary if loaded else 0 if not yet
19     loaded"""
20     return len(words)
21
22 def unload():
23     """Unloads dictionary from memory, returning true if successful else false"""
24     return True
```

Notice that there are four functions above. In the `check` function, if a `word` is in `words`, it returns `True`. So much easier than an implementation in C! Similarly, in the `load` function the dictionary file is opened. For each line in that file, we add that line to `words`. Using `rstrip`, the trailing new line is removed from the added word. `size` simply returns the `len` or length of `words`. `unload` only needs to return `True` because Python handles memory management on its own.

- The above code illustrates why higher-level languages exist: To simplify and allow you to write code more easily.
- However, speed is a tradeoff. Because C allows you, the programmer, to make decisions about memory management, it may run faster than Python – depending on your code. While C only runs your lines of code, Python runs all the code that comes under the hood with it when you call Python's built-in functions.
- You can learn more about functions in the [Python documentation](#)

Filter

- To further illustrate this simplicity, create a new file by typing `code blur.py` in your terminal window and write code as follows:

```
1 # Blurs an image
2
3 from PIL import Image, ImageFilter
4
5 # Blur image
6 before = Image.open("bridge.bmp")
7 after = before.filter(ImageFilter.BoxBlur(1))
8 after.save("out.bmp")
```

Notice that this program imports modules `Image` and `ImageFilter` from a library called `PIL`. This takes an input file and creates and output file.

- Further, you can create a new file called `edges.py` as follows:

```
1 # Finds edges in an image
2
3 from PIL import Image, ImageFilter
4
5 # Find edges
6 before = Image.open("bridge.bmp")
7 after = before.filter(ImageFilter.FIND_EDGES)
8 after.save("out.bmp")
```

Notice that this code is a small adjustment to your `blur` code, but produces a dramatically different result.

- Finally, you can even do face detection as follows:

```
1 # Find faces in picture
```

```

2  #
3  https://github.com/ageitgey/face_recognition/blob/master/examples/find_faces_in_p
4  icture.py
5
6
7  # Load the jpg file into a numpy array
8  image = face_recognition.load_image_file("office.jpg")
9
10 # Find all the faces in the image using the default HOG-based model.
11 # This method is fairly accurate, but not as accurate as the CNN model and not
12 # GPU accelerated.
13 # See also: find_faces_in_picture_cnn.py
14 face_locations = face_recognition.face_locations(image)
15
16 for face_location in face_locations:
17
18     # Print the location of each face in this image
19     top, right, bottom, left = face_location
20
21     # You can access the actual face itself like this:
22     face_image = image[top:bottom, left:right]
23     pil_image = Image.fromarray(face_image)
24     pil_image.show()

```

Notice how this file uses a third-party library called `face_recognition`. This is enabled by running `pip install face_recognition` in one's terminal window.

- Python allows you to abstract away programming that would be much more complicated within C and other *lower-level* programming languages.

CS50 Library

- As with C, the CS50 library can be utilized within Python.
- The following functions will be of particular use:

```

1  get_float
2  get_int
3  get_string

```

- You also have the option of importing only specific functions from the CS50 library as follows:

```

1  from CS50 import get_float, get_int, get_string

```

Strings

- In C, you might remember this code:

```
1 // get_string and printf with %s
2
3 #include <cs50.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     string answer = get_string("What's your name? ");
9     printf("hello, %s\n", answer);
10 }
```

- This code is transformed in Python to:

```
1 # get_string and print, with concatenation
2
3 from cs50 import get_string
4
5 answer = get_string("What's your name? ")
6 print("hello, " + answer)
```

You can write this code by executing `code hello.py` in the terminal window. Then, you can execute this code by running `python hello.py`. Notice how the `+` sign concatenates `"hello, "` and `answer`.

- Similarly, you could implement the above code as:

```
1 # get_string and print, with format strings
2
3 from cs50 import get_string
4
5 answer = get_string("What's your name? ")
6 print(f"hello, {answer}")
```

Notice how the curly braces allow for the `print` function to interpolate the `answer` such that `answer` appears within. The `f` is required to include the `answer` properly formatting.

Variables

- Variable declaration is simplified too. In C, you might have `int counter = 0;`. In Python, this same line would read `counter = 0`. You need not declare the type of the variable.
- Python favors `counter += 1` to increment by one, losing the ability found in C to type `counter++`.

Types

- Data types in Python do not need to be explicitly declared. For example, you saw how `answer` above is a string, but we did not have to tell the interpreter this was the case: It knew on its own.
- In Python, commonly used types include:

Notice that `long` and `double` are missing. Python will handle what data type should be used for larger and smaller numbers.

- Some other data types in Python include:

```

1  range
2  list
3  tuple
4  dict
5  set

```

- Each of these data types can be implemented in C, but in Python they can be implemented more simply.

Calculator

- You might recall `calculator.c` from earlier in the course:

```

1 // Addition with int
2
3 #include <cs50.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     // Prompt user for x
9     int x = get_int("x: ");
10
11    // Prompt user for y
12    int y = get_int("y: ");
13
14    // Perform addition
15    printf("%i\n", x + y);
16 }

```

- We can implement a simple calculator just as we did within C. Type `calculator.py` into the terminal window and write code as follows:

```

1 # Addition with int [using get_int]
2
3 from cs50 import get_int
4
5 # Prompt user for x
6 x = get_int("x: ")
7
8 # Prompt user for y
9 y = get_int("y: ")
10
11 # Perform addition
12 print(x + y)

```

Notice how the CS50 library is imported. Then, `x` and `y` are gathered from the user. Finally, the result is printed. Notice that the `main` function that would have been seen in a C program is gone entirely!

While one could utilize a `main` function, it is not required.

- It's possible for one to remove the training wheels of the CS50 library. Modify your code as follows:

```
1 # Addition with int [using input]
2
3 # Prompt user for x
4 x = input("x: ")
5
6 # Prompt user for y
7 y = input("y: ")
8
9 # Perform addition
10 print(x + y)
```

Notice how executing the above code results in strange program behavior. Why might this be so?

- You may have guessed that the interpreter understood `x` and `y` to be strings. You can fix your code by employing the `int` function as follows:

```
1 # Addition with int [using input]
2
3 # Prompt user for x
4 x = int(input("x: "))
5
6 # Prompt user for y
7 y = int(input("y: "))
8
9 # Perform addition
10 print(x + y)
```

Notice how the input for `x` and `y` is passed to the `int` function which converts it to an integer. Without converting `x` and `y` to be integers, the characters will concatenate.

Conditionals

- In C, you might remember a program like this:

```
1 // Conditionals, Boolean expressions, relational operators
2
3 #include <cs50.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     // Prompt user for integers
9     int x = get_int("What's x? ");
10    int y = get_int("What's y? ");
11
12    // Compare integers
13    if (x < y)
```

```

14 {
15     printf("x is less than y\n");
16 }
17 else if (x > y)
18 {
19     printf("x is greater than y\n");
20 }
21 else
22 {
23     printf("x is equal to y\n");
24 }
25 }
```

- In Python, it would appear as follows:

```

1 # Conditionals, Boolean expressions, relational operators
2
3 from cs50 import get_int
4
5 # Prompt user for integers
6 x = get_int("What's x? ")
7 y = get_int("What's y? ")
8
9 # Compare integers
10 if x < y:
11     print("x is less than y")
12 elif x > y:
13     print("x is greater than y")
14 else:
15     print("x is equal to y")
```

Notice that there are no more curly braces. Instead, indentations are utilized. Second, a colon is utilized in the `if` statement. Further, `elif` replaces `else if`. Parentheses are also no longer required in the `if` and `elif` statements.

- In C, we faced challenges when we wanted to compare two values. Consider the following code:

```

1 // Conditionals, Boolean expressions, relational operators
2
3 #include <cs50.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     // Prompt user for integers
9     int x = get_int("What's x? ");
10    int y = get_int("What's y? ");
11
12    // Compare integers
13    if (x < y)
14    {
```

```

15     printf("x is less than y\n");
16 }
17 else if (x > y)
18 {
19     printf("x is greater than y\n");
20 }
21 else
22 {
23     printf("x is equal to y\n");
24 }
25 }
```

- In Python, we can execute the above as follows:

```

1 # Conditionals, Boolean expressions, relational operators
2
3 from cs50 import get_int
4
5 # Prompt user for integers
6 x = get_int("What's x? ")
7 y = get_int("What's y? ")
8
9 # Compare integers
10 if x < y:
11     print("x is less than y")
12 elif x > y:
13     print("x is greater than y")
14 else:
15     print("x is equal to y")
```

Notice that the CS50 library is imported. Further, minor changes exist in the `if` statement.

- Further looking at comparisons, consider the following code in C:

```

1 // Logical operators
2
3 #include <cs50.h>
4 #include <stdio.h>
5
6 int main(void)
7 {
8     // Prompt user to agree
9     char c = get_char("Do you agree? ");
10
11    // Check whether agreed
12    if (c == 'Y' || c == 'y')
13    {
14        printf("Agreed.\n");
15    }
16    else if (c == 'N' || c == 'n')
17    {
```

```
18     printf("Not agreed.\n");
19 }
20 }
```

- The above can be implemented as follows:

```
1 # Logical operators
2
3 from cs50 import get_string
4
5 # Prompt user to agree
6 s = get_string("Do you agree? ")
7
8 # Check whether agreed
9 if s == "Y" or s == "y":
10     print("Agreed.")
11 elif s == "N" or s == "n":
12     print("Not agreed.")
```

Notice that the two vertical bars utilized in C is replaced with `or`. Indeed, people often enjoy Python because it is more readable by humans. Also, notice that `char` does not exist in Python. Instead, `strs` are utilized.

- Another approach to this same code could be as follows using *lists*:

```
1 # Logical operators, using lists
2
3 from cs50 import get_string
4
5 # Prompt user to agree
6 s = get_string("Do you agree? ")
7
8 # Check whether agreed
9 if s in ["y", "yes"]:
10     print("Agreed.")
11 elif s in ["n", "no"]:
12     print("Not agreed.")
```

Notice how we are able to express multiple keywords like `y` and `yes` in a `list`.

Object-Oriented Programming

- Up until this point, our programs in this course have been linear: sequential.
- It's possible to have certain types of values not only have properties or attributes inside of them but have functions as well. In Python, these values are known as *objects*
- In C, we could create a `struct` where you could associate multiple variables inside a single self-created data type. In Python, we can do this and also include functions in a self-created data type. When a function belongs to a specific *object*, it is known as a *method*.

- For example, `strs` in Python have a built-in *methods*. Therefore, you could modify your code as follows:

```

1 # Logical operators, using lists
2
3 from cs50 import get_string
4
5 # Prompt user to agree
6 s = get_string("Do you agree? ").lower()
7
8 # Check whether agreed
9 if s.lower() in ["y", "yes"]:
10     print("Agreed.")
11 elif s.lower() in ["n", "no"]:
12     print("Not agreed.")

```

Notice how the old value of `s` is overwritten with the result of `s.lower()`, a built-in method of `strs`.

- In this class, we will only scratch the surface of Python. Therefore, the [Python documentation](#) will be of particular importance as you continue.
- You can learn more about string methods in the [Python documentation](#)

Loops

- Loops in Python are very similar to C. You may recall the following code in C:

```

1 // Demonstrates while loop
2
3 #include <stdio.h>
4
5 int main(void)
6 {
7     int i = 0;
8     while (i < 3)
9     {
10         printf("meow\n");
11         i++;
12     }
13 }

```

- In Python, this code appears as:

```

1 # Demonstrates while loop
2
3 i = 0
4 while i < 3:
5     print("meow")
6     i += 1

```

- `for` loops can be implemented in Python as follows:

```
1 # Better design
2
3 for i in range(3):
4     print("meow")
```

Notice that `i` is never explicitly used. However, Python will increment the value of `i`.

- Similarly, one could express the above code as:

```
1 # Abstraction with parameterization
2
3 def main():
4     meow(3)
5
6
7 # Meow some number of times
8 def meow(n):
9     for i in range(n):
10        print("meow")
11
12
13 main()
```

Notice that a function is utilized to abstract away the meowing.

- Finally, a `while` loop could be implemented as follows:

```
1 # Demonstrates while loop
2
3 i = 0
4 while i < 3:
5     print("meow")
6     i += 1
```

- To further our understanding of loops and iteration in Python, let's create a new file called `uppercase.py` as follows:

```
1 # Uppercases string one character at a time
2
3 before = input("Before: ")
4 print("After: ", end="")
5 for c in before:
6     print(c.upper(), end="")
7 print()
```

Notice how `end=` is used to pass a parameter to the `print` function that continues the line without a line ending. This code passes one string at a time.

- Reading the documentation, we discover that Python has methods that can be implemented upon the entire string as follows:

```
1 # Uppercases string all at once
2
3 before = input("Before: ")
4 after = before.upper()
5 print(f"After: {after}")
```

Notice how `.upper` is applied to the entire string.

Abstraction

- As we hinted at earlier today, you can further improve upon our code using functions and abstracting away various code into functions. Modify your earlier-created `meow.py` code as follows:

```
1 # Abstraction
2
3 def main():
4     for i in range(3):
5         meow()
6
7 # Meow once
8 def meow():
9     print("meow")
10
11
12 main()
```

Notice that the `meow` function abstracts away the `print` statement. Further, notice that the `main` function appears at the top of the file. At the bottom of the file, the `main` function is called. By convention, it's expected that you create a `main` function in Python.

- Indeed, we can pass variables between our functions as follows:

```
1 # Abstraction with parameterization
2
3 def main():
4     meow(3)
5
6
7 # Meow some number of times
8 def meow(n):
9     for i in range(n):
10        print("meow")
11
12
13 main()
```

Notice how `meow` now takes a variable `n`. In the `main` function, you can call `meow` and pass a value like `3` to it. Then, `meow` utilizes the value of `n` in the `for` loop.

- Reading the above code, notice how you, as a C programmer, are able to quite easily make sense of the above code. While some conventions are different, the building blocks you previously learned are very apparent in this new programming language.

Truncation and Floating Point Imprecision

- Recall that in C, we experienced truncation where one integer being divided by another could result in an imprecise result.
- You can see how Python handles such division as follows by modifying your code for `calculator.py`:

```

1 # Division with integers, demonstration lack of truncation
2
3 # Prompt user for x
4 x = int(input("x: "))
5
6 # Prompt user for y
7 y = int(input("y: "))
8
9 # Divide x by y
10 z = x / y
11 print(z)

```

Notice that executing this code results in a value, but that if you were to see more digits after `.333333` you'd see that we are faced with *floating-point imprecision*. Truncation does not occur.

- We can reveal this imprecision by modifying our codes slightly:

```

1 # Floating-point imprecision
2
3 # Prompt user for x
4 x = int(input("x: "))
5
6 # Prompt user for y
7 y = int(input("y: "))
8
9 # Divide x by y
10 z = x / y
11 print(f"{z:.50f}")

```

Notice that this code reveals the imprecision. Python still faces this issue, just as C does.

Exceptions

- Let's explore more about exceptions that can occur when we run Python code.
- Modify `calculator.py` as follows:

```

1 # Implements get_int
2
3 def get_int(prompt):

```

```

4     return int(input(prompt))
5
6
7 def main():
8
9     # Prompt user for x
10    x = get_int("x: ")
11
12    # Prompt user for y
13    y = get_int("y: ")
14
15    # Perform addition
16    print(x + y)
17
18
19 main()

```

Notice that inputting the wrong data could result in an error.

- We can `try` to handle and *catch* potential exceptions by modifying our code as follows:

```

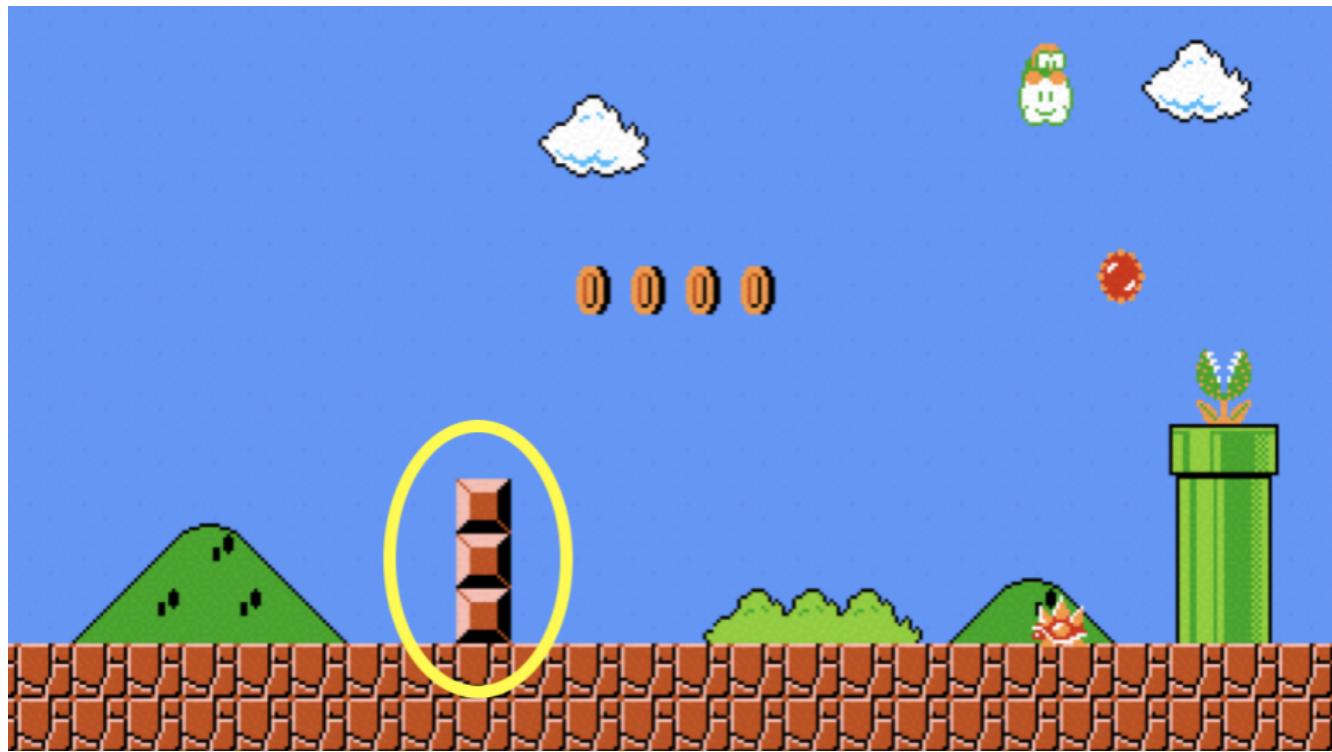
1 # Implements get_int with a loop
2
3 def get_int(prompt):
4     while True:
5         try:
6             return int(input(prompt))
7         except ValueError:
8             print("Not an integer")
9
10
11 def main():
12
13     # Prompt user for x
14     x = get_int("x: ")
15
16     # Prompt user for y
17     y = get_int("y: ")
18
19     # Perform addition
20     print(x + y)
21
22
23 main()

```

Notice that the above code repeatedly tries to get the correct type of data, providing additional prompts when needed.

Mario

- Recall a few weeks ago our challenge of building three blocks on top of one another, like in Mario.



- In Python, we can implement something akin to this as follows:

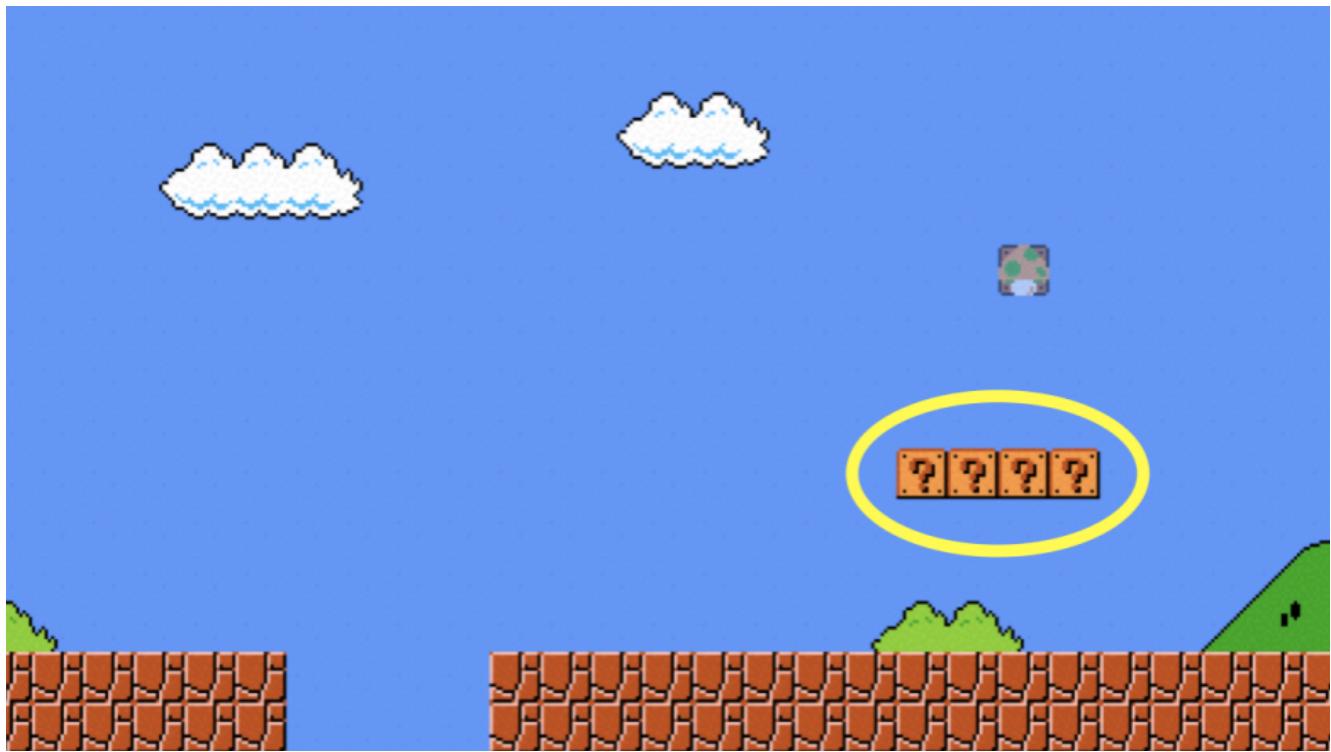
```
1 # Prints a column of 3 bricks with a loop
2
3 for i in range(3):
4     print("#")
```

- In C, we had the advantage of a `do-while` loop. However, in Python it is convention to utilize a `while` loop, as Python does not have a `do while` loop. You can write code as follows in a file called `mario.py`:

```
1 # Prints a column of n bricks with a loop
2
3 from cs50 import get_int
4
5 while True:
6     n = get_int("Height: ")
7     if n > 0:
8         break
9
10    for i in range(n):
11        print("#")
```

Notice how the while loop is used to obtain the height. Once a height greater than zero is inputted, the loop breaks.

- Consider the following image:



- In Python, we could implement by modifying your code as follows:

```
1 # Prints a row of 4 question marks with a loop
2
3 for i in range(4):
4     print "?", end=""
5 print()
```

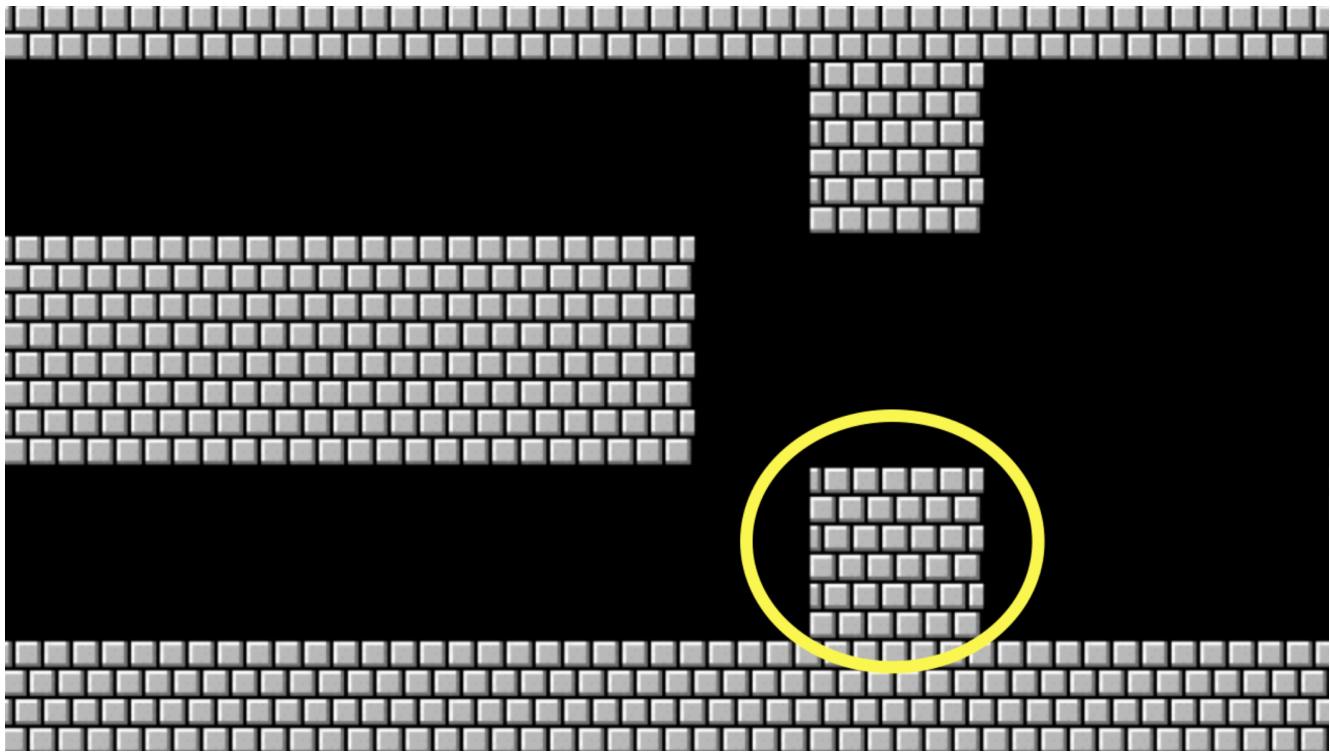
Notice that you can override the behavior of the `print` function to stay on the same line as the previous `print`.

- Similar in spirit to previous iterations, we can further simplify this program:

```
1 # Prints a row of 4 question marks without a loop
2
3 print "?" * 4
```

Notice that we can utilize `*` to multiply the `print` statement to repeat `4` times.

- What about a large block of bricks?



- To implement the above, you can modify your code as follows:

```
1 # Prints a 3-by-3 grid of bricks with loops
2
3 for i in range(3):
4     for j in range(3):
5         print("#", end="")
6     print()
```

Notice how one `for` loop exists inside another. The `print` statement adds a new line at the end of each row of bricks.

- You can learn more about the `print` function in the [Python documentation](#)

Lists

- `lists` are a data structure within Python.
- `lists` have built in methods or functions within them.
- For example, consider the following code:

```
1 # Averages three numbers using a list
2
3 # Scores
4 scores = [72, 73, 33]
5
6 # Print average
7 average = sum(scores) / len(scores)
8 print(f"Average: {average}")
```

Notice that you can use the built-in `sum` method to calculate the average.

- You can even utilize the following syntax to get values from the user:

```

1 # Averages three numbers using a list and a loop
2
3 from cs50 import get_int
4
5 # Get scores
6 scores = []
7 for i in range(3):
8     score = get_int("Score: ")
9     scores.append(score)
10
11 # Print average
12 average = sum(scores) / len(scores)
13 print(f"Average: {average}")

```

Notice that this code utilizes the built-in `append` method for lists.

- You can learn more about lists in the [Python documentation](#)
- You can also learn more about `len` in the [Python documentation](#)

Searching and Dictionaries

- We can also search within a data structure.
- Consider a program called `phonebook.py` as follows:

```

1 # Implements linear search for names using loop
2
3 # A list of names
4 names = ["Carter", "David", "John"]
5
6 # Ask for name
7 name = input("Name: ")
8
9 # Search for name
10 for n in names:
11     if name == n:
12         print("Found")
13         break
14 else:
15     print("Not found")

```

Notice how this implements linear search for each name.

- However, we don't need to iterate through a list. In Python, we can execute linear search as follows:

```

1 # Implements linear search for names using `in`
2
3 # A list of names
4 names = ["Carter", "David", "John"]
5
6 # Ask for name
7 name = input("Name: ")
8
9 # Search for name
10 if name in names:
11     print("Found")
12 else:
13     print("Not found")

```

Notice how `in` is used to implement linear search.

- Still, this code could be improved.
- Recall that a *dictionary* or `dict` is a collection of *key* and *value* pairs.
- You can implement a dictionary in Python as follows:

```

1 # Implements a phone book as a list of dictionaries, without a variable
2
3 from cs50 import get_string
4
5 people = [
6     {"name": "Carter", "number": "+1-617-495-1000"},
7     {"name": "David", "number": "+1-617-495-1000"},
8     {"name": "John", "number": "+1-949-468-2750"},
9 ]
10
11 # Search for name
12 name = get_string("Name: ")
13 for person in people:
14     if person["name"] == name:
15         print(f"Found {person['number']}")
16         break
17 else:
18     print("Not found")

```

Notice that the dictionary is implemented having both `name` and `number` for each entry.

- Even better, strictly speaking, we don't need both a `name` and a `number`. We can simplify this code as follows:

```

1 # Implements a phone book using a dictionary
2
3 from cs50 import get_string
4
5 people = {
6     "Carter": "+1-617-495-1000",

```

```

7     "David": "+1-617-495-1000",
8     "John": "+1-949-468-2750",
9 }
10
11 # Search for name
12 name = get_string("Name: ")
13 if name in people:
14     print(f"Number: {people[name]}")
15 else:
16     print("Not found")

```

Notice that the dictionary is implemented using curly braces. Then, the statement `if name in people` searches to see if the `name` is in the `people` dictionary. Further, notice how, in the `print` statement, we can index into the `people` dictionary using the value of `name`. Very useful!

- Python has done their best to get to *constant time* using their built-in searches.
- You can learn more about dictionaries in the [Python documentation](#)

Command-Line Arguments

- As with C, you can also utilize command-line arguments. Consider the following code:

```

1 # Prints a command-line argument
2
3 from sys import argv
4
5 if len(argv) == 2:
6     print(f"hello, {argv[1]}")
7 else:
8     print("hello, world")

```

Notice that `argv[1]` is printed using a *formatted string*, noted by the `f` present in the `print` statement.

- You can print all the arguments in `argv` as follows:

```

1 # Printing command-line arguments, indexing into argv
2
3 from sys import argv
4
5 for i in range(len(argv)):
6     print(argv[i])

```

Notice that the above will not present the word `python` if executed, and the first argument will be the name of the file you are running. You can think of the word `python` as being analogous to `./` when we were running programs in C.

- You can slice pieces of lists away. Consider the following code:

```
1 # Printing command-line arguments
2
3 from sys import argv
4
5 for arg in argv:
6     print(arg)
```

Notice that executing this code will result in the name of the file you are running being sliced away.

- You can learn more about the `sys` library in the [Python documentation](#)

Exit Status

- The `sys` library also has built-in methods. We can use `sys.exit(i)` to exit the program with a specific exit code:

```
1 # Exits with explicit value, importing sys
2
3 import sys
4
5 if len(sys.argv) != 2:
6     print("Missing command-line argument")
7     sys.exit(1)
8
9 print(f"hello, {sys.argv[1]}")
10 sys.exit(0)
```

Notice that dot-notation is used to utilize the built-in functions of `sys`.

Third-Party Libraries

- One of the advantages of Python is its massive user-base and similarly large number of third-party libraries.
- For example, David demoed the use of `cowsay` and `qrCode` libraries.

Summing Up

In this lesson, you learned how the building blocks of programming from prior lessons can be implemented within Python. Further, you learned about how Python allowed for more simplified code. Also, you learned how to utilize various Python libraries. In the end, you learned that your skills as a programmer are not limited to a single programming language. Already, you are seeing how you are discovering a new way of learning through this course that could serve you in any programming language – and, perhaps, in nearly any avenue of learning! Specifically, we discussed...

- Python
- Variables
- Conditionals
- Loops

- Types
- Object-Oriented programming
- Truncation and floating point imprecision
- Exceptions
- Dictionaries
- Command-line arguments
- Third-Party libraries

See you next time!

lecture 6.5-Artificial Intelligence

Artificial Intelligence

- [Welcome!](#)
- [Image Generation](#)
- [ChatGPT](#)
- [Prompt Generation](#)
- [CS50.ai](#)
- [Generative AI](#)
- [Decision Trees](#)
- [Minimax](#)
- [Machine Learning](#)
- [Deep Learning](#)
- [Generative Artificial Intelligence](#)
- [Summing Up](#)

Welcome!

- In computer science and programming circles, *rubber ducking* or *rubber duck debugging* is the act of speaking to an inanimate object to be able to *talk through* a challenging problem.
- Most recently, CS50 created our own rubber duck debugger at [cs50.ai](#), which uses artificial intelligence as a way by which to interact with students to help them with their own challenging problems.
- Students engaging with this tool can begin understanding the potential of what AI can offer the world.

Image Generation

- Numerous AI tools have created the potential for artificially generated images to enter the world.
- Up until recently, most of these tools had numerous tells that might indicate to an observer that an image is AI-generated.

- However, tools are becoming exceedingly good at generating these images.
- Indeed, as technology improves, it will soon be almost, if not entirely, impossible for such images to be detected with the naked eye.
- Software has also gained the ability to mutate individual images within video.

ChatGPT

- A very well-known bleeding-edge tool is the text generation tool *chatGPT*.
- In CS50, we do not allow the use of ChatGPT. However, we do allow the use of our own rubber duck debugger at cs50.ai.
- In CS50, we leverage the tools of Azure and OpenAI, along with our own vector database that holds very recent information from our most recent lectures and offerings, to provide our rubber duck debugger toll.

Prompt Generation

- *Prompt generation* is the way by which an individual can communicate with an AI platform.
- We use a *system prompt* to teach the AI how to interact with users. We teach the AI how to work with students.
- *User prompts* are those provided by users to interact with the AI. With these prompts, students interact with the AI.

CS50.ai

- Our [rubber duck debugger](#) can provide conceptual help with computer science concepts.
- Further, the [rubber duck debugger](#) can help students write more efficient code.
- Additionally, the [rubber duck debugger](#) can help when a student is stuck in one of their assignments. For example, students may encounter errors that prevent them from progressing in their assignments. When students hit a wall, they don't have to wait for support staff to be available.
- The [rubber duck debugger](#) does stipulate, however, that it is an AI and that it is experimental. Students should be conscious of the degree to which they blindly trust the AI. Consider the following image:



CS50 Duck BOT



- 3 It seems like you're trying to declare the `main` function with only one parameter. However, in C, the `main` function should have either zero parameters or two parameters, which are `int argc` and `char *argv[]`. The first parameter, `argc`, represents the number of command-line arguments, and the second parameter, `argv`, is an array of strings representing each of those arguments.

For the Caesar problem, you'll want to use these parameters to access the command-line argument that represents the key. To do this, you should declare your `main` function as follows:

```
int main(int argc, char *argv[])
{
    // Your code here
}
```

Then, you can check if the user has provided the correct number of arguments (in this case, just one additional argument for the key) and handle the input accordingly.

P.S. I am an experimental bot. Quack. Do not assume that my reply is accurate unless you see that it's been "endorsed" by (human) staff. Quack.

[Comment](#) [Edit](#) [Delete](#) [Unendorse](#) ...

- AI has an inhuman level of patience.

Generative AI

- AI has been with us for much time! Software has long adapted to users. Algorithms look for patterns in junk mail, images saved on your phone, and to play games.
- In games, for example, step-by-step instructions may allow a computerized adversary play a game of Breakout.

Decision Trees

- Decision trees* are used by an algorithm to decide what decision to make.
- For example, in Breakout, an algorithm may consider what choice to make based on the instructions in the code:

```
1 While game is ongoing:
2   If ball is left of paddle:
3     Move paddle left
4   Else if ball is right of padding:
5     Move paddle right
6   Else:
7     Don't move paddle
```

- With most games, they attempt to minimize the number of calculations required to compete with the player.

Minimax

- You can imagine where an algorithm may score outcomes as positive, negative, and neutral.

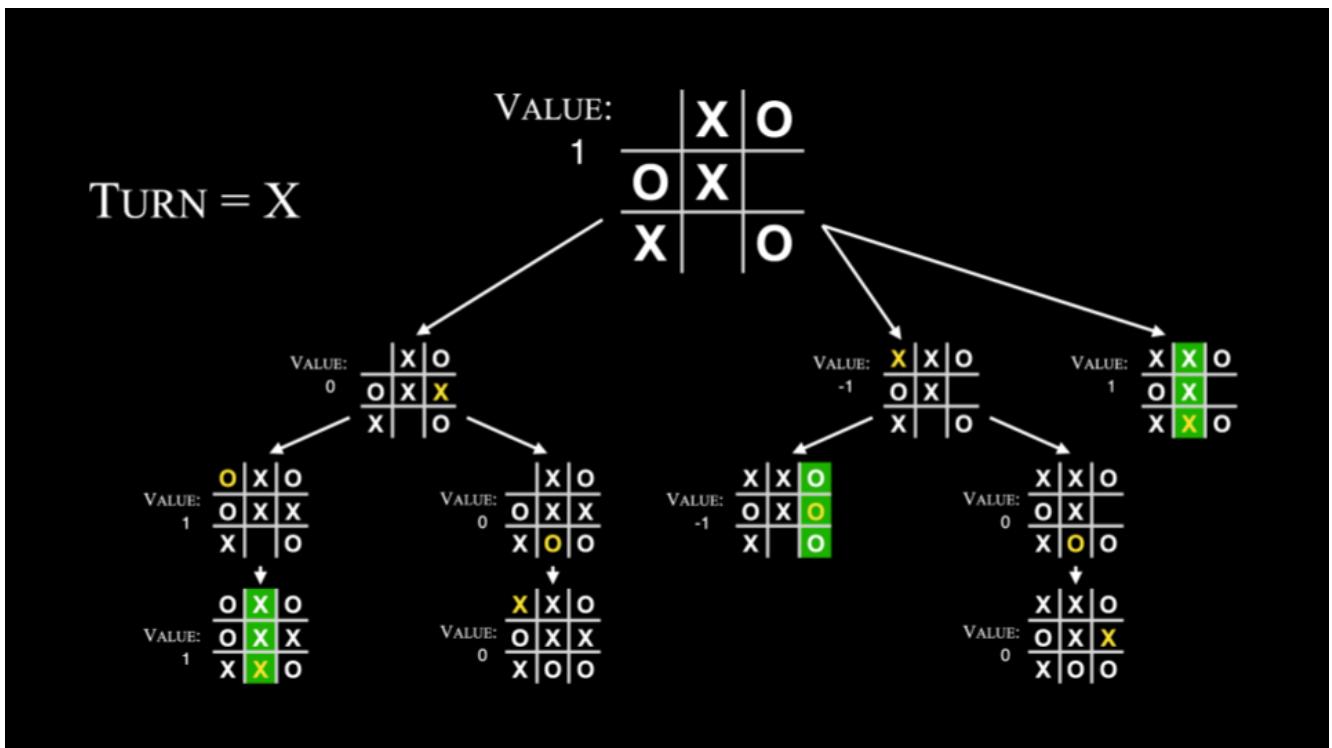
- In tic-tac-toe, the AI may consider a board where the computer wins as 1 and one where the computer loses as -1.
- You can imagine how a computer may look at a decision tree of potential outcomes and assign scores to each potential move.
- The computer will attempt to win by maximizing its own score.
- In the context of tic-tac-toe, the algorithm may conceptualize this as follows:

```

1 If player is X:
2 For each possible move:
3   Calculate score for board
4 Choose move with highest score
5
6 Else if player is O:
7   For each possible move:
8     Calculate score for board
9   Choose move with lowest score

```

- This could be pictured as follows:



- Because computers are so powerful, they can crunch massive potential outcomes. However, the computers in our pockets or on our desks may not be able to calculate trillions of options. This is where machine learning can help.

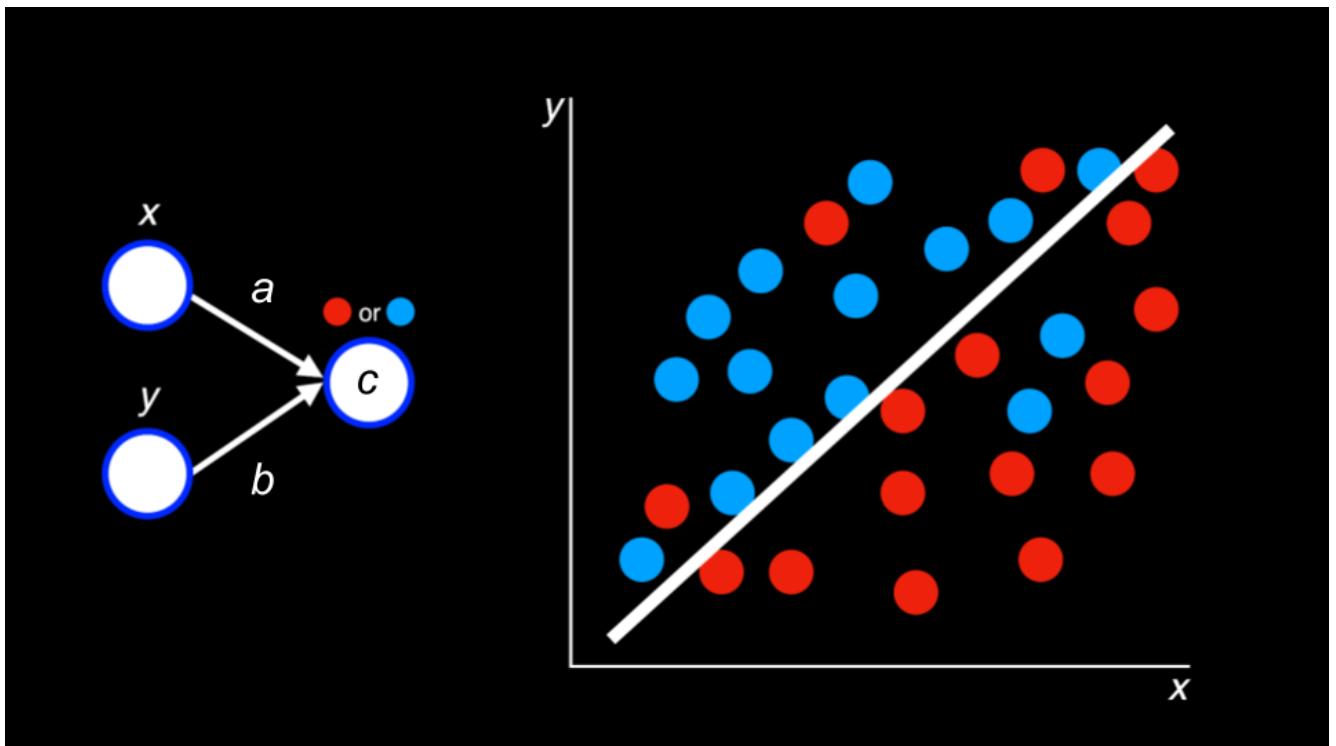
Machine Learning

- Machine learning is a way by which a computer can learn through reinforcement.
- A computer can learn how to flip a pancake.
- A computer can learn how to play Mario.
- A computer can learn how to play The Floor is Lava.

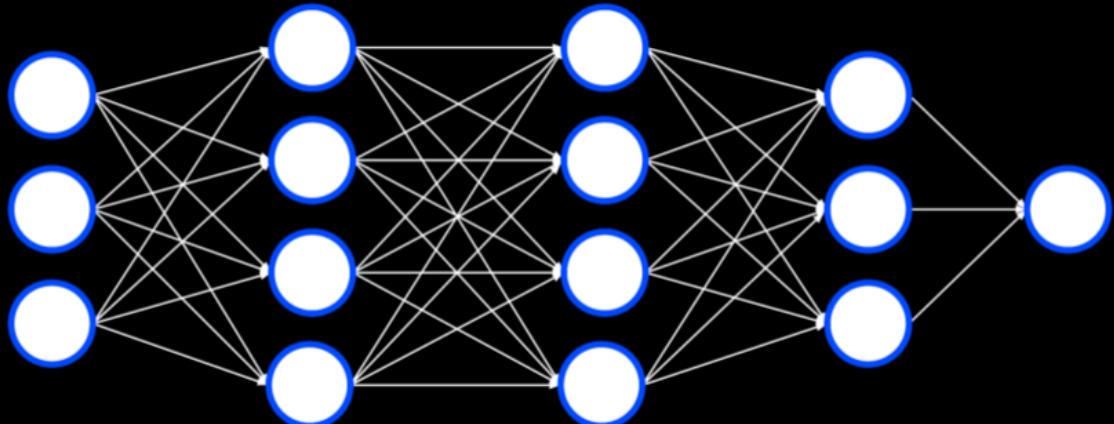
- The computer repeats trial after trial after trial to discover what behaviors to repeat and those not to repeat.
- Within much of AI-based algorithms, there are concepts of *explore vs. exploit*, where the AI may randomly try something that may not be considered optimal. Randomness can yield better outcomes.

Deep Learning

- *Deep learning* uses neural networks whereby problems and solutions are explored.
- For example, deep learning may attempt to predict whether a blue or red dot will appear somewhere on a graph. Consider the following image:



- Existing training data is used to predict an outcome. Further, more training data may be created by the AI to discover further patterns.
- Deep learning creates nodes (pictured below), which associate inputs and outputs.



Generative Artificial Intelligence

- *Large language models* are massive models that make predictions based on huge amounts of training.
- Just a few years ago, AI was not very good at completing and generating sentences.
- The AI encodes words into *embeddings* to find relationships between words. Thus, through a huge amount of training, a massive neural network can predict the association between words - resulting in the ability for generative AI to generate content and even have conversations with users.
- These technologies are what is behind our [rubber duck debugger](#).

Summing Up

In this lesson, you learned about some of the technology behind our own [rubber duck debugger](#). Specifically, we discussed...

- Image Generation
- ChatGPT
- Prompt Generation
- CS50.ai
- Generative AI
- Decision Trees
- Minimax
- Machine Learning
- Deep Learning
- Generative Artificial Intelligence

This was CS50!

Lecture 7-SQL

##

- [Welcome!](#)
- [Flat-File Database](#)
- [Relational Databases](#)
- [Shows](#)
- [JOINs](#)
- [Indexes](#)
- [Using SQL in Python](#)
- [Race Conditions](#)
- [SQL Injection Attacks](#)
- [Summing Up](#)

Welcome!

- In previous weeks, we introduced you to Python, a high-level programming language that utilized the same building blocks we learned in C. However, we introduced this new language not for the purpose of learning “just another language.” Instead, we do so because some tools are better for some jobs and not so great for others!
- This week, we will be continuing more syntax related to Python.
- Further, we will be integrating this knowledge with data.
- Finally, we will be discussing *SQL* or *Structured Query Language*.
- Overall, one of the goals of this course is to learn to program generally – not simply how to program in the languages described in this course.

Flat-File Database

- As you have likely seen before, data can often be described in patterns of columns and rows.
- Spreadsheets like those created in Microsoft Excel and Google Sheets can be outputted to a `csv` or *comma-separated values* file.
- If you look at a `csv` file, you’ll notice that the file is flat in that all of our data is stored in a single table represented by a text file. We call this form of data a *flat-file database*.
- Python comes with native support for `csv` files.
- First, download [favorites.csv](#) and upload it to your file explorer inside [cs50.dev](#). Second, in your terminal window, type `code favorites.py` and write code as follows:

```
1 # Prints all favorites in CSV using csv.reader  
2
```

```

3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create reader
9     reader = csv.reader(file)
10
11    # Skip header row
12    next(reader)
13
14    # Iterate over CSV file, printing each favorite
15    for row in reader:
16        print(row[1])

```

Notice that the `csv` library is imported. Further, we created a `reader` that will hold the result of `csv.reader(file)`. The `csv.reader` function reads each row from the file, and in our code we store the results in `reader.print(row[1])`, therefore, will print the language from the `favorites.csv` file.

- You can improve your code as follows:

```

1 # Stores favorite in a variable
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create reader
9     reader = csv.reader(file)
10
11    # Skip header row
12    next(reader)
13
14    # Iterate over CSV file, printing each favorite
15    for row in reader:
16        favorite = row[1]
17        print(favorite)

```

Notice that `favorite` is stored and then printed. Also notice that we use the `next` function to skip to the next line of our reader.

- One of the disadvantages of the above approach is that we are trusting that `row[1]` is always the favorite. However, what would happen if the columns have been moved around?
- We can fix this potential issue. Python also allows you to index by the keys of a list. Modify your code as follows:

```

1 # Prints all favorites in CSV using csv.DictReader
2
3 import csv
4

```

```

5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11    # Iterate over CSV file, printing each favorite
12    for row in reader:
13        favorite = row["language"]
14        print(favorite)

```

Notice that this example directly utilizes the `language` key in the print statement.

- This could be further simplified to:

```

1 # Prints all favorites in CSV using csv.DictReader
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11    # Iterate over CSV file, printing each favorite
12    for row in reader:
13        print(row["language"])

```

- To count the number of favorite languages expressed in the `csv` file, we can do the following:

```

1 # Counts favorites using variables
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11    # Counts
12    scratch, c, python = 0, 0, 0
13
14    # Iterate over CSV file, counting favorites
15    for row in reader:
16        favorite = row["language"]
17        if favorite == "Scratch":
18            scratch += 1
19        elif favorite == "C":
20            c += 1

```

```

21     elif favorite == "Python":
22         python += 1
23
24 # Print counts
25 print(f"Scratch: {scratch}")
26 print(f"C: {c}")
27 print(f"Python: {python}")

```

Notice that each language is counted using `if` statements. Further notice the double equal `==` signs in those `if` statements.

- Python allows us to use a dictionary to count the `counts` of each language. Consider the following improvement upon our code:

```

1 # Counts favorites using dictionary
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11     # Counts
12     counts = {}
13
14     # Iterate over CSV file, counting favorites
15     for row in reader:
16         favorite = row["language"]
17         if favorite in counts:
18             counts[favorite] += 1
19         else:
20             counts[favorite] = 1
21
22     # Print counts
23     for favorite in counts:
24         print(f"{favorite}: {counts[favorite]$")

```

Notice that the value in `counts` with the key `favorite` is incremented when it exists already. If it does not exist, we define `counts[favorite]` and set it to 1. Further, the formatted string has been improved to present the `counts[favorite]`.

- Python also allows sorting `counts`. Improve your code as follows:

```

1 # Sorts favorites by key
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:

```

```

7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11    # Counts
12    counts = {}
13
14    # Iterate over CSV file, counting favorites
15    for row in reader:
16        favorite = row["language"]
17        if favorite in counts:
18            counts[favorite] += 1
19        else:
20            counts[favorite] = 1
21
22    # Print counts
23    for favorite in sorted(counts):
24        print(f"{favorite}: {counts[favorite]})
```

Notice the `sorted(counts)` at the bottom of the code.

- If you look at the parameters for the `sorted` function in the Python documentation, you will find it has many built-in parameters. You can leverage some of these built-in parameters as follows:

```

1 # Sorts favorites by value using .get
2
3 import csv
4
5 # Open CSV file
6 with open("favorites.csv", "r") as file:
7
8     # Create DictReader
9     reader = csv.DictReader(file)
10
11    # Counts
12    counts = {}
13
14    # Iterate over CSV file, counting favorites
15    for row in reader:
16        favorite = row["language"]
17        if favorite in counts:
18            counts[favorite] += 1
19        else:
20            counts[favorite] = 1
21
22    # Print counts
23    for favorite in sorted(counts, key=counts.get, reverse=True):
24        print(f"{favorite}: {counts[favorite]})
```

Notice the arguments passed to `sorted`. The `key` argument allows you to tell Python the method you wish to use to sort items. In this case `counts.get` is used to sort by the values. `reverse=True` tells `sorted` to sort from largest to smallest.

- Python has numerous libraries that we can utilize in our code. One of these libraries is `collections`, from which we can import `Counter`. `Counter` will allow you to access the counts of each language without the headaches of all the `if` statements seen in our previous code. You can implement as follows:

```
1 # Sorts favorites by value using .get
2
3 import csv
4
5 from collections import Counter
6
7 # Open CSV file
8 with open("favorites.csv", "r") as file:
9
10    # Create DictReader
11    reader = csv.DictReader(file)
12
13    # Counts
14    counts = Counter()
15
16    # Iterate over CSV file, counting favorites
17    for row in reader:
18        favorite = row["language"]
19        counts[favorite] += 1
20
21    # Print counts
22    for favorite, count in counts.most_common():
23        print(f"{favorite}: {count}")
```

Notice how `counts = Counter()` enables the use of this imported `Counter` class from `collections`.

- We can change the column we are examining, focusing on our favorite problem instead:

```
1 # Favorite problem instead of favorite language
2
3 import csv
4
5 from collections import Counter
6
7 # Open CSV file
8 with open("favorites.csv", "r") as file:
9
10    # Create DictReader
11    reader = csv.DictReader(file)
12
13    # Counts
14    counts = Counter()
```

```

15
16     # Iterate over CSV file, counting favorites
17     for row in reader:
18         favorite = row["problem"]
19         counts[favorite] += 1
20
21 # Print counts
22 for favorite, count in counts.most_common():
23     print(f"{favorite}: {count}")

```

Notice that `problem` replaced `language`.

- We can also get the count of the popularity of a specific problem in the course:

```

1 # Gets a specific count
2
3 import csv
4
5 from collections import Counter
6
7 # Open CSV file
8 with open("favorites.csv", "r") as file:
9
10    # Create DictReader
11    reader = csv.DictReader(file)
12
13    # Counts
14    counts = Counter()
15
16    # Iterate over CSV file, counting favorites
17    for row in reader:
18        favorite = row["problem"]
19        counts[favorite] += 1
20
21    # Print count
22    favorite = input("Favorite: ")
23    print(f"{favorite}: {counts[favorite]}")

```

Notice how compact our code is compared to our experience in C.

Relational Databases

- Google, Twitter, and Meta all use relational databases to store their information at scale.
- Relational databases store data in rows and columns in structures called *tables*.
- SQL allows for four types of commands:

```

1 Create
2 Read
3 Update
4 Delete

```

- These four operations are affectionately called *CRUD*.
- We can create a SQL database at the terminal by typing `sqlite3 favorites.db`. Upon being prompted, we will agree that we want to create `favorites.db` by pressing `y`.
- You will notice a different prompt as we are now inside a program called `sqlite3`.
- We can put `sqlite3` into `csv` mode by typing `.mode csv`. Then, we can import our data from our `csv` file by typing `.import favorites.csv favorites`. It seems that nothing has happened!
- We can type `.schema` to see the structure of the database.
- You can read items from a table using the syntax `SELECT columns FROM table`.
- For example, you can type `SELECT * FROM favorites;` which will iterate every row in `favorites`.
- You can get a subset of the data using the command `SELECT language FROM favorites;`.
- SQL supports many commands to access data, including:

```

1  AVG
2  COUNT
3  DISTINCT
4  LOWER
5  MAX
6  MIN
7  UPPER

```

- For example, you can type `SELECT COUNT(language) FROM favorites;`. Further, you can type `SELECT DISTINCT(language) FROM favorites;` to get a list of the individual languages within the database. You could even type `SELECT COUNT(DISTINCT(language)) FROM favorites;` to get a count of those.
- SQL offers additional commands we can utilize in our queries:

```

1  WHERE      -- adding a Boolean expression to filter our data
2  LIKE       -- filtering responses more loosely
3  ORDER BY   -- ordering responses
4  LIMIT      -- limiting the number of responses
5  GROUP BY   -- grouping responses together

```

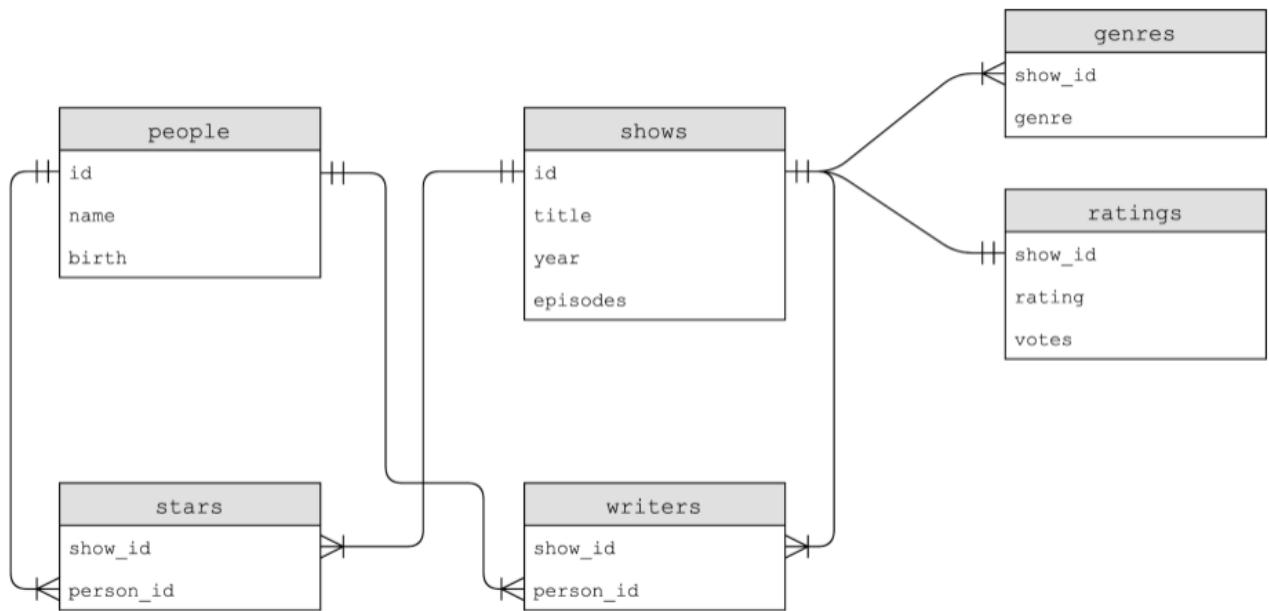
Notice that we use `--` to write a comment in SQL.

- For example, we can execute `SELECT COUNT(*) FROM favorites WHERE language = 'C';`. A count is presented.
- Further, we could type `SELECT COUNT(*) FROM favorites WHERE language = 'C' AND problem = 'Hello, World;'`. Notice how the `AND` is utilized to narrow our results.
- Similarly, we could execute `SELECT language, COUNT(*) FROM favorites GROUP BY language;`. This would offer a temporary table that would show the language and count.
- We could improve this by typing `SELECT language, COUNT(*) FROM favorites GROUP BY language ORDER BY COUNT(*);`. This will order the resulting table by the `count`.
- We can also `INSERT` into a SQL database utilizing the form `INSERT INTO table (column...) VALUES(value, ...);`.
- We can execute `INSERT INTO favorites (language, problem) VALUES ('SQL', 'Fiftyville');`.

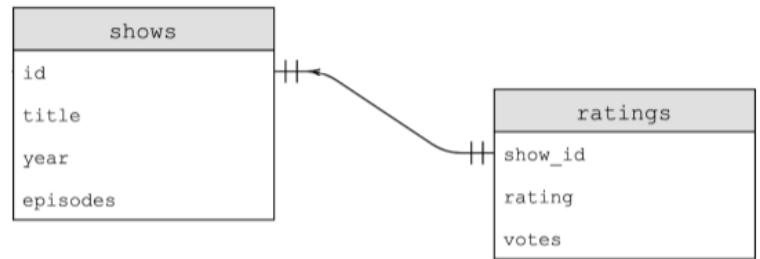
- `DELETE` allows you to delete parts of your data. For example, you could `DELETE FROM favorites WHERE Timestamp IS NULL;`.
- We can also utilize the `UPDATE` command to update your data.
- For example, you can execute `UPDATE favorites SET language = 'SQL', problem = 'Fiftyville';`. This will result in overwriting all previous statements where C was the favorite programming language.
- Notice that these queries have immense power. Accordingly, in the real-world setting, you should consider who has permissions to execute certain commands.

Shows

- We can imagine a database that we might want to create to catalog various TV shows. We could create a spreadsheet with columns like `title`, `star`, `star`, `star`, `star`, and more stars. A problem with this approach is this approach has a lot of wasted space. Some shows may have one star. Others may have dozens.
- We could separate our database into multiple sheets. We could have a `shows` sheet and a `people` sheet. On the `people` sheet, each person could have a unique `id`. On the `shows` sheet, each show could have a unique `id` too. On a third sheet called `stars` we could relate how each show has people for each show by having a `show_id` and `person_id`. While this is an improvement, this is not an ideal database.
- IMDb offers a database of people, shows, writers, stars, genres, and ratings. Each of these tables is related to one another as follows:



- After downloading [`shows.db`](#), you can execute `sqlite3 shows.db` in your terminal window.
- Let's zero in on the relationship between two tables within the database called `shows` and `ratings`. The relationship between these two tables can be illustrated as follows:



- To illustrate the relationship between these tables, we could execute the following command: `SELECT * FROM ratings LIMIT 10;`. Examining the output, we could execute `SELECT * FROM shows LIMIT 10;`.
- To understand the database, upon executing `.schema` you will find not only each of the tables but the individual fields inside each of these fields.
- As you can see, `shows` has an `id` field. The `genres` table has a `show_id` field which has data that is common between it and the `shows` table.
- Further, `show_id` exists in all of the tables. In the `shows` table, it is simply called `id`. This common field between all the fields is called a *key*. Primary keys are used to identify a unique record in a table. *Foreign keys* are used to build relationships between tables by pointing to the primary key in another table.
- By storing data in a relational database, as above, data can be more efficiently stored.
- In *sqlite*, we have five datatypes, including:

1	BLOB	-- binary large objects that are groups of ones and zeros
2	INTEGER	-- an integer
3	NUMERIC	-- for numbers that are formatted specially like dates
4	REAL	-- like a float
5	TEXT	-- for strings and the like

- Additionally, columns can be set to add special constraints:
- We could execute `SELECT * FROM stars LIMIT 10;`. `show_id` is a foreign key in this final query because `show_id` corresponds to the unique `id` field in `shows`. `person_id` corresponds to the unique `id` field in the `people` column.
- We can further play with this data to understand these relationships. Execute `SELECT * FROM ratings;`. There are a lot of ratings!

- We can further limit this data down by executing `SELECT show_id FROM ratings WHERE rating >= 6.0 LIMIT 10;`. From this query, you can see that there are 10 shows presented. However, we don't know what show each `show_id` represents.
- You can discover what shows these are by executing `SELECT * FROM shows WHERE id = 626124;`
- We can further our query to be more efficient by executing:

```

1  SELECT title
2  FROM shows
3  WHERE id IN (
4      SELECT show_id
5      FROM ratings
6      WHERE rating >= 6.0
7      LIMIT 10
8  )

```

Notice that this query nests together two queries. An inner query is used by an outer query.

JOINS

- We are pulling data from `shows` and `ratings`.
- How could we combine tables temporarily? Tables could be joined together using the `JOIN` command.
- Execute the following command:

```

1  SELECT * FROM shows
2  JOIN ratings ON shows.id = ratings.show_id
3  WHERE rating >= 6.0
4  LIMIT 10;

```

Notice this results in a wider table than we have previously seen.

- Where the previous queries have illustrated the *one-to-one* relationship between these keys, let's examine some *one-to-many* relationships. Focusing on the `genres` table, execute the following:

```

1  SELECT * FROM genres
2  LIMIT 10;

```

Notice how this provides us a sense of the raw data. You might notice that one shows have three values. This is a one-to-many relationship.

- We can learn more about the `genres` table by typing `.schema genres`.
- Execute the following command to learn more about the various comedies in the database:

```
1 SELECT title FROM shows
2 WHERE id IN (
3     SELECT show_id FROM genres
4     WHERE genre = 'Comedy'
5     LIMIT 10
6 );
```

Notice how this produces a list of comedies, including *Catweazle*.

- To learn more about Catweazle, by joining various tables through a join:

```
1 SELECT * FROM shows
2 JOIN genres
3 ON shows.id = genres.show_id
4 WHERE id = 63881;
```

Notice that this results in a temporary table. It is fine to have duplicate table.

- A final relationship is a *many-to-many* relationship.
- We can learn more about the show *The Office* by executing the following command:

```
1 SELECT person_id FROM stars
2 WHERE show_id = (
3     SELECT id FROM shows
4     WHERE title = 'The Office' AND year = 2005
5 );
```

Notice that this results in a table that includes the `person_ids` of various stars.

- I could learn more about this group of actors by executing the following:

```
1 SELECT name FROM people
2 WHERE id IN (
3     SELECT person_id FROM stars
4     WHERE show_id = (
5         SELECT id FROM shows
6         WHERE title = 'The Office' AND year = 2005
7     )
8 );
```

This results in a top-billed stars.

- We can further understand this data by executing:

```

1  SELECT title from shows
2  WHERE id IN (
3      SELECT show_id FROM stars
4      WHERE person_id =
5          SELECT id FROM people
6          WHERE name = 'Steve Carell'
7      )
8  );

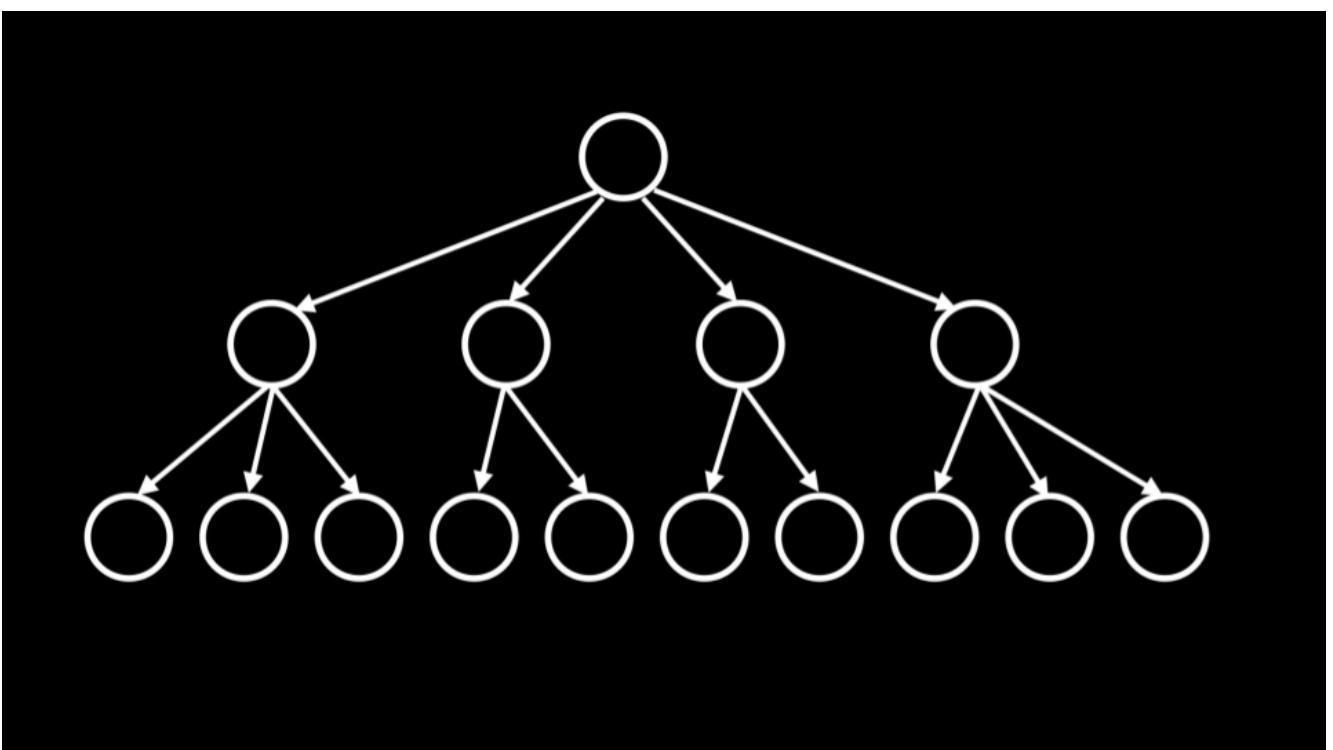
```

This results in a list of titles of shows wherein Steve Carell stared.

- The wildcard `%` operator can be used to find all people whose names start with `Steve C` one could employ the syntax `SELECT * FROM people WHERE name LIKE 'Steve C%';`.

Indexes

- While relational databases have the ability to be more fast and more robust than utilizing a `.csv` file, data can be optimized within a table using *indexes*.
- Indexes can be utilized to speed up our queries.
- We can track the speed of our queries by executing `.timer on` in `sqlite3`.
- To understand how indexes can speed up our queries, run the following: `SELECT * FROM shows WHERE title = 'The Office';` Notice the time that displays after the query executes.
- Then, we can create an index with the syntax `CREATE INDEX title_index ON shows (title);`. This tells `sqlite3` to create an index and perform some special under-the-hood optimization relating to this column `title`.
- This will create a data structure called a *B Tree*, a data structure that looks similar to a binary tree. However, unlike a binary tree, there can be more than two child notes.



- Running the query `SELECT * FROM shows WHERE title = 'The Office';`, you will notice that the query runs much more quickly!
- Unfortunately, indexing all columns would result in utilizing more storage space. Therefore, there is a tradeoff for enhanced speed.

Using SQL in Python

- To assist in working with SQL in this course, the CS50 Library can be utilized as follows in your code:
- Similar to previous uses of the CS50 Library, this library will assist with the complicated steps of utilizing SQL within your Python code.
- You can read more about the CS50 Library's SQL functionality in the [documentation](#).
- Recall where we last left off in `favorites.py`. Your code should appear as follows:

```

1 # Gets a specific count
2
3 import csv
4
5 from collections import Counter
6
7 # Open CSV file
8 with open("favorites.csv", "r") as file:
9
10    # Create DictReader
11    reader = csv.DictReader(file)
12
13    # Counts
14    counts = Counter()
15
16    # Iterate over CSV file, counting favorites
17    for row in reader:
18        favorite = row["problem"]
19        counts[favorite] += 1
20
21    # Print count
22    favorite = input("Favorite: ")
23    print(f"{favorite}: {counts[favorite]}")

```

Notice how this code is exactly as we left it prior.

- Modify your code as follows:

```

1 # Searches database popularity of a problem
2
3 import csv
4
5 from cs50 import SQL
6
7 # Open database
8 db = SQL("sqlite:///favorites.db")

```

```

9
10 # Prompt user for favorite
11 favorite = input("Favorite: ")
12
13 # Search for title
14 rows = db.execute("SELECT COUNT(*) AS n FROM favorites WHERE problem LIKE ?",
15 favorite)
16
17 # Get first (and only) row
18 row = rows[0]
19
20 # Print popularity
21 print(row["n"])

```

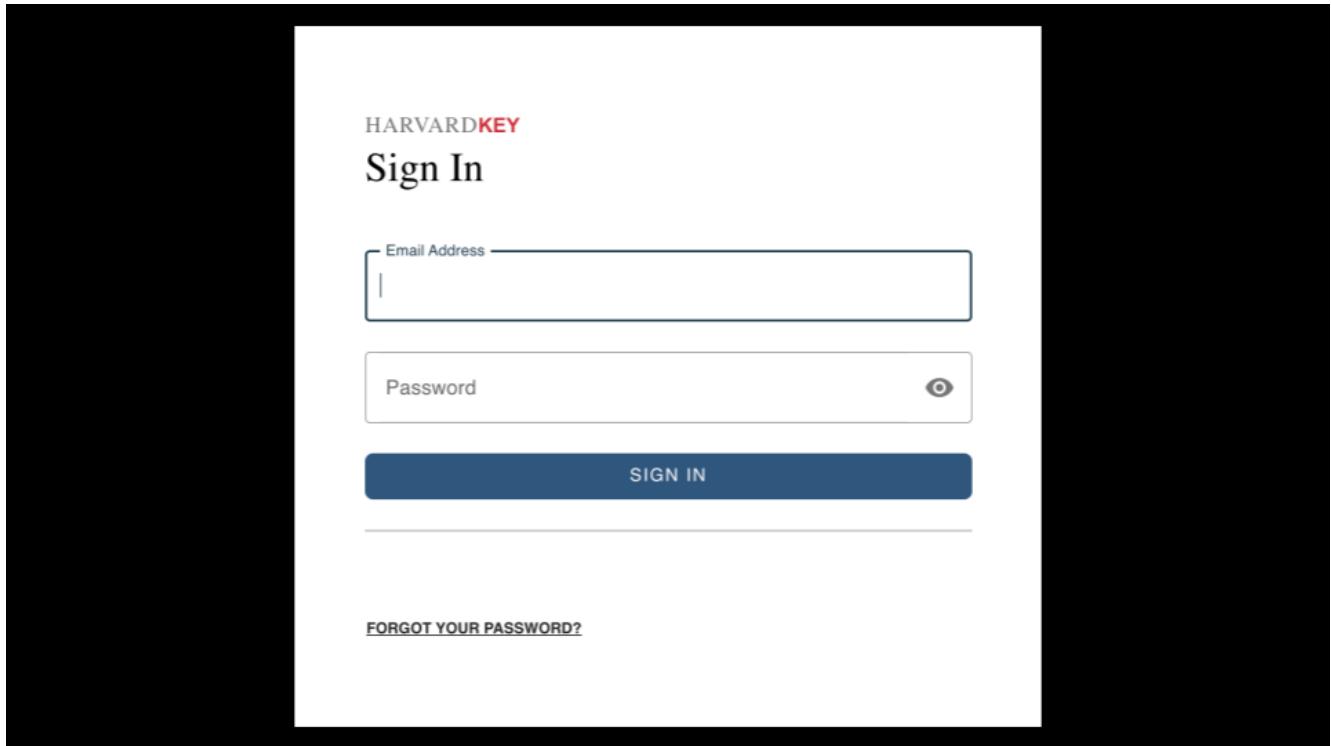
Notice that `db = SQL("sqlite:///favorites.db")` provide Python the location of the database file. Then, the line that begins with `rows` executes SQL commands utilizing `db.execute`. Indeed, this command passes the syntax within the quotation marks to the `db.execute` function. We can issue any SQL command using this syntax. Further, notice that `rows` is returned as a list of dictionaries. In this case, there is only one result, one row, returned to the `rows` list as a dictionary.

Race Conditions

- Utilization of SQL can sometimes result in some problems.
- You can imagine a case where multiple users could be accessing the same database and executing commands at the same time.
- This could result in glitches where code is interrupted by other people's actions. This could result in a loss of data.
- Built-in SQL features such as `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` help avoid some of these race condition problems.

SQL Injection Attacks

- Now, still considering the code above, you might be wondering what the `?` question marks do above. One of the problems that can arise in real-world applications of SQL is what is called an *injection attack*. An injection attack is where a malicious actor could input malicious SQL code.
- For example, consider a login screen as follows:



- Without the proper protections in our own code, a bad actor could run malicious code. Consider the following:

```
1 rows = db.execute("SELECT COUNT(*) FROM users WHERE username = ? AND password = ?",
    username, password)
```

Notice that because the `?` is in place, validation can be run on `favorite` before it is blindly accepted by the query.

- You never want to utilize formatted strings in queries as above or blindly trust the user's input.
- Utilizing the CS50 Library, the library will *sanitize* and remove any potentially malicious characters.

Summing Up

In this lesson, you learned more syntax related to Python. Further, you learned how to integrate this knowledge with data in the form of flat-file and relational databases. Finally, you learned about *SQL*. Specifically, we discussed...

- Flat-file databases
- Relational databases
- SQL
- Primary and foreign keys
- `JOINs`
- Indexes
- Using SQL in Python
- Race conditions
- SQL injection attacks

See you next time!

Lecture 8-HTML&CSS

##

- [Welcome!](#)
- [Routers](#)
- [DNS](#)
- [HTTP](#)
- [HTML](#)
- [Regular Expressions](#)
- [CSS](#)
- [Frameworks](#)
- [JavaScript](#)
- [Summing Up](#)

Welcome!

- In previous weeks, we introduced you to Python, a high-level programming language that utilized the same building blocks we learned in C. Today, we will extend those building blocks further in HTML, CSS, and JavaScript.
- The internet is a technology that we all use.
- Using our skills from previous weeks, we can build our own web pages and applications.
- The *ARPANET* connected the first points on the internet to one another.
- Dots between two points could be considered *routers*.

Routers

- To route data from one place to another, we need to make *routing decisions*. That is, someone needs to program how data is transferred from point A to point B.
- You can imagine how data could take multiple paths from point A and point B, such that when a router is congested, data can flow through another path. *Packets* of data are transferred from one router to another, from one computer to another.
- *TCP/IP* are two protocols that allow computers to transfer data between them over the internet.
- *IP* or *internet protocol* is a way by which computers can identify one another across the internet. Every computer has a unique address in the world. Addresses are in this form:
- Numbers range from `0` to `255`. IP addresses are 32-bits, meaning that these addresses could accommodate over 4 billion addresses. Newer versions of IP addresses, implementing 128-bits, can accommodate far more computers!
- In the real world, servers do a lot of work for us.
- Packets are structured as follows:

Example Internet Datagram Header

- Packets are standardized. The source and destination are held within each packet.
 - TCP, or transmission control protocol, is used to distinguish web services from one another. For example, 80 is used to denote HTTP and 443 is used to denote HTTPS. These numbers are *port numbers*.
 - When information is sent from one location to another, a source IP address, a destination IP address, and TCP port number are sent.
 - These protocols are also used to fragment large files into multiple parts or packets. For example, a large photo of a cat can be sent in multiple packets. When a packet is lost, TCP/IP can request missing packets again from the origin server.
 - TCP will acknowledge when all the data has been transmitted and received.

DNS

- It would be very tedious if you needed to remember an IP address to visit a website.
 - *DNS*, or *domain name systems*, is a collection of servers on the internet that are used to route website addresses like *harvard.edu* to a specific IP address.
 - DNS simply hold a table or database that links specific, fully qualified domain names to specific IP addresses.

HTTP

- *HTTP* or *hypertext transfer protocol* is an application-level protocol that developers use to build powerful and useful things through the transferring of data from one place to another.
 - When you see an address such as `https://www.example.com` you are actually implicitly visiting that address with a `/` at the end of it.
 - The *path* is what exists after that slash. For example, `https://www.example.com/folder/file.html` visits `example.com` and browses to the `folder` folder and then visits the file named `file.html`.

- The `.com` is called a *top-level domain* that is used to denote the location or type of organization associated with this address.
- `https` in this address is the protocol that is used to connect to that web address. By protocol, we mean that HTTP utilizes `GET` or `POST` *requests* to ask for information from a server. For example, you can launch Google Chrome, right-click, and click `inspect`. When you open the `developer tools` and visit `Network`, selecting `Preserve log`, you will see `Request Headers`. You'll see mentions of `GET`. This is possible in other browsers as well, using slightly different methods.
- For example, when issuing a GET request, your computer may send the following to a server:

```
1 | GET / HTTP/2
2 | Host: www.harvard.edu
```

Notice that this requests via HTTP the content served on www.harvard.edu

- Generally, after making a request a server, you will receive the following in `Response Headers`:

```
1 | HTTP/2 200
2 | Content-Type: text/html
```

- This approach to inspecting these logs may be a bit more complicated than need be. You can analyze the work of HTTP protocols at cs50.dev. For example, type the following in your terminal window:

```
1 | curl -I https://www.harvard.edu/
```

Notice that the output of this command returns all the header values of the responses of the server.

- Via developer tools in your web browser, you can see all the HTTP requests when browsing to the above website.
- Further, execute the following command in your terminal window:

```
1 | curl -I https://harvard.edu
```

Notice that you will see a `301` response, providing a hint to a browser of where it can find the correct website.

- Similarly, execute the following in your terminal window:

```
1 | curl -I http://www.harvard.edu/
```

Notice that the `s` in `https` has been removed. The server response will show that the response is `301`, meaning that the website has permanently moved.

- Similar to `301`, a code of `404` means that a specified URL has not been found. There are numerous other response codes, such as:

```
1 200 OK
2 301 Moved Permanently
3 302 Found
4 304 Not Modified
5 304 Temporary Redirect
6 401 Unauthorized
7 403 Forbidden
8 404 Not Found
9 418 I'm a Teapot
10 500 Internal Server Error
11 503 Service Unavailable
```

- It's worth mentioning that `500` errors are always your fault as the developer. This will be especially important for next week's problem set, and potentially for your final project!

HTML

- *HTML* or *hypertext markup language* is made up of *tags*, each of which may have some *attributes* that describe it.
- In your terminal, type `code hello.html` and write code as follows:

```
1 <!DOCTYPE html>
2
3 <!-- Demonstrates HTML -->
4
5 <html lang="en">
6   <head>
7     <title>hello, title</title>
8   </head>
9   <body>
10    hello, body
11  </body>
12 </html>
```

Notice that the `html` tag both opens and closes this file. Further, notice the `lang` attribute, which modifies the behavior of the `html` tag. Also, notice that there are both `head` tags and `body` tags. Indentation is not required but does suggest a hierarchy.

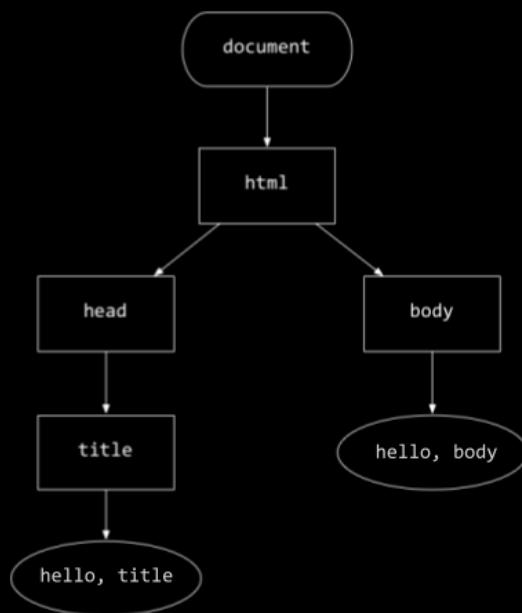
- You can serve your code by typing `http-server`. This serve is now available on a very long URL. If you click it, you can visit the website with your own code.
- When you visit this URL, notice that the file name `hello.html` appears at the end of this URL. Further, notice, based upon the URL, that the server is serving via port 8080.
- The hierarchy of tags can be represented as follows:

```

<!DOCTYPE html>

<html lang="en">
  <head>
    <title>
      hello, title
    </title>
  </head>
  <body>
    hello, body
  </body>
</html>

```



- Knowledge of this hierarchy will be useful later as we learn JavaScript.
- The browser will read your HTML file top to bottom and left to right.
- Because whitespace and indentation is effectively ignored in HTML, you will need to use `<p>` paragraph tags to open and close a paragraph. Consider the following:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates paragraphs -->
4
5  <html lang="en">
6    <head>
7      <title>paragraphs</title>
8    </head>
9    <body>
10      <p>
11        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
12        convallis scelerisque quam, vel hendrerit lectus viverra eu. Praesent posuere
13        eget lectus ut faucibus. Etiam eu velit laoreet, gravida lorem in, viverra est.
14        Cras ut purus neque. In porttitor non lorem id lobortis. Mauris gravida metus
          libero, quis maximus dui porta at. Donec lacinia felis consectetur venenatis
          scelerisque. Nulla eu nisl sollicitudin, varius velit sit amet, vehicula erat.
          Curabitur sollicitudin felis sit amet orci mattis, a tempus nulla pulvinar.
          Aliquam erat volutpat.
12      </p>
13      <p>
14        Mauris ut dui in eros semper hendrerit. Morbi vel elit mi. Sed sit
          amet ex non quam dignissim dignissim et vel arcu. Pellentesque eget elementum
          orci. Morbi ac cursus ex. Pellentesque quis turpis blandit orci dapibus semper
          sed non nunc. Nulla et dolor nec lacus finibus volutpat. Sed non lorem diam.
          Donec feugiat interdum interdum. Vivamus et justo in enim blandit fermentum vel
          at elit. Phasellus eu ante vitae ligula varius aliquet. Etiam id posuere nibh.

```

```

15      </p>
16      <p>
17          Aenean venenatis convallis ante a rhoncus. Nullam in metus vel diam
18      vehicula tincidunt. Donec lacinia metus sem, sit amet egestas elit blandit sit
19      amet. Nunc egestas sem quis nisl mattis semper. Pellentesque ut magna congue
20      lorem eleifend sodales. Donec tortor tortor, aliquam vitae mollis sed, interdum
21      ut lectus. Mauris non purus quis ipsum lacinia tincidunt.
22      </p>
23      <p>
24          Integer at justo lacinia libero blandit aliquam ut ut dui. Quisque
25      tincidunt facilisis venenatis. Nullam dictum odio quis lorem luctus, vel
26      malesuada dolor luctus. Aenean placerat faucibus enim a facilisis. Maecenas
27      eleifend quis massa sed eleifend. Ut ultricies, dui ac vulputate hendrerit, ex
28      metus iaculis diam, vitae fermentum libero dui et ante. Phasellus suscipit, arcu
29      ut consequat sagittis, massa urna accumsan massa, eu aliquet nulla lorem vitae
      arcu. Pellentesque rutrum felis et metus porta semper. Nam ac consectetur mauris.
      </p>
      <p>
      Suspendisse rutrum vestibulum odio, sed venenatis purus condimentum
      sed. Morbi ornare tincidunt augue eu auctor. Vivamus sagittis ac lectus at
      aliquet. Nulla urna mauris, interdum non nibh in, vehicula porta enim. Donec et
      posuere sapien. Pellentesque ultrices scelerisque ipsum, vel fermentum nibh
      tincidunt et. Proin gravida porta ipsum nec scelerisque. Vestibulum fringilla
      erat at turpis laoreet, nec hendrerit nisi scelerisque.
      </p>
      <p>
      Sed quis malesuada mi. Nam id purus quis augue sagittis pharetra.
      Nulla facilisi. Maecenas vel fringilla ante. Cras tristique, arcu sit amet
      blandit auctor, urna elit ultricies lacus, a malesuada eros dui id massa. Aliquam
      sem odio, pretium vel cursus eget, scelerisque at urna. Vestibulum posuere a
      turpis consectetur consectetur. Cras consequat, risus quis tempor egestas, nulla
      ipsum ornare erat, nec accumsan nibh lorem nec risus. Integer at iaculis lacus.
      Integer congue nunc massa, quis molestie felis pellentesque vestibulum. Nulla
      odio tortor, aliquam nec quam in, ornare aliquet sapien.
      </p>
      </body>
      </html>

```

Notice that paragraphs start with a `<p>` tag and end with a `</p>` tag.

- HTML allows for the representation of headings:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates headings (for chapters, sections, subsections, etc.) -->
4
5  <html lang="en">
6
7      <head>
8          <title>headings</title>
9      </head>
10

```

```
11 <body>
12
13     <h1>One</h1>
14     <p>
15         Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
16         convallis scelerisque quam, vel hendrerit lectus viverra eu. Praesent posuere
17         eget lectus ut faucibus. Etiam eu velit laoreet, gravida lorem in, viverra est.
18         Cras ut purus neque. In porttitor non lorem id lobortis. Mauris gravida metus
19         libero, quis maximus dui porta at. Donec lacinia felis consectetur venenatis
20         scelerisque. Nulla eu nisl sollicitudin, varius velit sit amet, vehicula erat.
21         Curabitur sollicitudin felis sit amet orci mattis, a tempus nulla pulvinar.
22         Aliquam erat volutpat.
23     </p>
24
25     <h2>Two</h2>
26     <p>
27         Mauris ut dui in eros semper hendrerit. Morbi vel elit mi. Sed sit
28         amet ex non quam dignissim dignissim et vel arcu. Pellentesque eget elementum
29         orci. Morbi ac cursus ex. Pellentesque quis turpis blandit orci dapibus semper
30         sed non nunc. Nulla et dolor nec lacus finibus volutpat. Sed non lorem diam.
31         Donec feugiat interdum interdum. Vivamus et justo in enim blandit fermentum vel
32         at elit. Phasellus eu ante vitae ligula varius aliquet. Etiam id posuere nibh.
33     </p>
34
35     <h3>Three</h3>
36     <p>
37         Aenean venenatis convallis ante a rhoncus. Nullam in metus vel diam
38         vehicula tincidunt. Donec lacinia metus sem, sit amet egestas elit blandit sit
39         amet. Nunc egestas sem quis nisl mattis semper. Pellentesque ut magna congue
40         lorem eleifend sodales. Donec tortor tortor, aliquam vitae mollis sed, interdum
41         ut lectus. Mauris non purus quis ipsum lacinia tincidunt.
42     </p>
43
44     <h4>Four</h4>
45     <p>
46         Integer at justo lacinia libero blandit aliquam ut ut dui. Quisque
47         tincidunt facilisis venenatis. Nullam dictum odio quis lorem luctus, vel
48         malesuada dolor luctus. Aenean placerat faucibus enim a facilisis. Maecenas
49         eleifend quis massa sed eleifend. Ut ultricies, dui ac vulputate hendrerit, ex
50         metus iaculis diam, vitae fermentum libero dui et ante. Phasellus suscipit, arcu
51         ut consequat sagittis, massa urna accumsan massa, eu aliquet nulla lorem vitae
52         arcu. Pellentesque rutrum felis et metus porta semper. Nam ac consectetur mauris.
53     </p>
54
55     <h5>Five</h5>
56     <p>
57         Suspendisse rutrum vestibulum odio, sed venenatis purus condimentum
58         sed. Morbi ornare tincidunt augue eu auctor. Vivamus sagittis ac lectus at
59         aliquet. Nulla urna mauris, interdum non nibh in, vehicula porta enim. Donec et
60         posuere sapien. Pellentesque ultrices scelerisque ipsum, vel fermentum nibh
61         tincidunt et. Proin gravida porta ipsum nec scelerisque. Vestibulum fringilla
62         erat at turpis laoreet, nec hendrerit nisi scelerisque.
```

```

36     </p>
37
38     <h6>Six</h6>
39     <p>
40         Sed quis malesuada mi. Nam id purus quis augue sagittis pharetra.
41         Nulla facilisi. Maecenas vel fringilla ante. Cras tristique, arcu sit amet
42         blandit auctor, urna elit ultricies lacus, a malesuada eros dui id massa. Aliquam
43         sem odio, pretium vel cursus eget, scelerisque at urna. Vestibulum posuere a
44         turpis consectetur consectetur. Cras consequat, risus quis tempor egestas, nulla
45         ipsum ornare erat, nec accumsan nibh lorem nec risus. Integer at iaculis lacus.
        Integer congue nunc massa, quis molestie felis pellentesque vestibulum. Nulla
        odio tortor, aliquam nec quam in, ornare aliquet sapien.
46     </p>
47
48     </body>
49
50 </html>

```

Notice that `<h1>`, `<h2>`, and `<h3>` denote different levels of headings.

- We can also create unordered lists within HTML:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates (ordered) lists -->
4
5  <html lang="en">
6      <head>
7          <title>list</title>
8      </head>
9      <body>
10         <ul>
11             <li>foo</li>
12             <li>bar</li>
13             <li>baz</li>
14         </ul>
15     </body>
16 </html>

```

Notice that the `` tag creates an unordered list containing three items.

- We can also create ordered lists within HTML:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates (ordered) lists -->
4
5  <html lang="en">
6      <head>
7          <title>list</title>
8      </head>
9      <body>

```

```
10     <ol>
11         <li>foo</li>
12         <li>bar</li>
13         <li>baz</li>
14     </ol>
15 </body>
16 </html>
```

Notice that the `` tag creates an ordered list containing three items.

- We can also create a table in HTML:

```
1 <!DOCTYPE html>
2
3 <!-- Demonstrates table -->
4
5 <html lang="en">
6     <head>
7         <title>table</title>
8     </head>
9     <body>
10        <table>
11            <tr>
12                <td>1</td>
13                <td>2</td>
14                <td>3</td>
15            </tr>
16            <tr>
17                <td>4</td>
18                <td>5</td>
19                <td>6</td>
20            </tr>
21            <tr>
22                <td>7</td>
23                <td>8</td>
24                <td>9</td>
25            </tr>
26            <tr>
27                <td>*</td>
28                <td>0</td>
29                <td>#</td>
30            </tr>
31        </table>
32    </body>
33 </html>
```

Tables also have tags that open and close each element within. Also, notice the syntax for comments in HTML.

- Images can also be utilized within HTML:

```
1 <!DOCTYPE html>
2
3 <!-- Demonstrates image -->
4
5 <html lang="en">
6   <head>
7     <title>image</title>
8   </head>
9   <body>
10    
11  </body>
12 </html>
```

Notice that `src="bridge.png"` indicates the path where the image file can be located.

- Videos can also be included in HTML:

```
1 <!DOCTYPE html>
2
3 <!-- Demonstrates video -->
4
5 <html lang="en">
6   <head>
7     <title>video</title>
8   </head>
9   <body>
10    <video controls muted>
11      <source src="video.mp4" type="video/mp4">
12    </video>
13  </body>
14 </html>
```

Notice that the `type` attribute designates that this is a video of type `mp4`. Further, notice how `controls` and `muted` are passed to `video`.

- You can also link between various web pages:

```
1 <!DOCTYPE html>
2
3 <!-- Demonstrates link -->
4
5 <html lang="en">
6   <head>
7     <title>link</title>
8   </head>
9   <body>
10    Visit <a href="https://www.harvard.edu">Harvard</a>.
11  </body>
12 </html>
```

Notice that the `<a>` or *anchor* tag is used to make `Harvard` a linkable text.

- Meta tags are used to hold information about the data within the HTML file. Consider the following:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates responsive design -->
4
5  <html lang="en">
6      <head>
7          <meta name="viewport" content="initial-scale=1, width=device-width">
8          <title>meta</title>
9      </head>
10     <body>
11         Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
12         convallis scelerisque quam, vel hendrerit lectus viverra eu. Praesent posuere
13         eget lectus ut faucibus. Etiam eu velit laoreet, gravida lorem in, viverra est.
14         Cras ut purus neque. In porttitor non lorem id lobortis. Mauris gravida metus
15         libero, quis maximus dui porta at. Donec lacinia felis consectetur venenatis
16         scelerisque. Nulla eu nisl sollicitudin, varius velit sit amet, vehicula erat.
17         Curabitur sollicitudin felis sit amet orci mattis, a tempus nulla pulvinar.
18         Aliquam erat volutpat.
19     </body>
20 </html>

```

Notice this set of `meta` attributes makes this page mobile-friendly.

- There are numerous `meta` key-value pairs that you can use:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates Open Graph tags -->
4
5  <html lang="en">
6      <head>
7          <meta property="og:title" content="CS50">
8          <meta property="og:description" content="Introduction to the intellectual
9             enterprises of computer science and the art of programming.">
10         <meta property="og:image" content="cat.jpg">
11         <title>meta</title>
12     </head>
13     <body>
14         ...
15     </body>
</html>

```

Notice that these key value pairs relate to the `title` and `description` of the web page.

- You can also create forms reminiscent of Google's search:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates form -->

```

```

4 <html lang="en">
5   <head>
6     <title>search</title>
7   </head>
8   <body>
9     <form action="https://www.google.com/search" method="get">
10       <input name="q" type="search">
11       <input type="submit" value="Google Search">
12     </form>
13   </body>
14 </html>

```

Notice that a `form` tag opens and provides the attribute of what `action` it will take. The `input` field is included, passing the name `q` and the type as `search`.

- We can make this search better as follows:

```

1 <!DOCTYPE html>
2
3 <!-- Demonstrates additional form attributes -->
4
5 <html lang="en">
6   <head>
7     <title>search</title>
8   </head>
9   <body>
10    <form action="https://www.google.com/search" method="get">
11      <input autocomplete="off" autofocus name="q" placeholder="Query"
12        type="search">
13      <button>Google Search</button>
14    </form>
15  </body>
16 </html>

```

Notice that `autocomplete` is turned `off`. `autofocus` is enabled.

- We've seen just a few of many HTML elements you can add to your site. If you have an idea for something to add to your site that we haven't seen yet (a button, an audio file, etc.) try Googling "X in HTML" to find the right syntax! Similarly, you can use cs50.ai to help you discover more HTML features!

Regular Expressions

- *Regular expressions* or *regexes* are a means by which to ensure that user-provided data fits a specific format.
- We can implement our own registration page that utilizes regexes as follows:

```

1 <!DOCTYPE html>
2
3 <!-- Demonstrates type="email" -->
4

```

```

5 <html lang="en">
6   <head>
7     <title>register</title>
8   </head>
9   <body>
10    <form>
11      <input autocomplete="off" autofocus name="email" placeholder="Email"
12        type="email">
13      <button>Register</button>
14    </form>
15  </body>
16 </html>

```

Notice that the `input` tag includes attributes that designate that this is of type `email`. The browser knows to double-check that input is an email address.

- While the browser uses these built-in attributes to check for an email address, we can add a `pattern` attribute to ensure that only specific data ends up in the email address:

```

1 <!DOCTYPE html>
2
3 <!-- Demonstrates pattern attribute -->
4
5 <html lang="en">
6   <head>
7     <title>register</title>
8   </head>
9   <body>
10    <form>
11      <input autocomplete="off" autofocus name="email" pattern=".+@.+\.edu"
12        placeholder="Email" type="email">
13      <button>Register</button>
14    </form>
15  </body>
16 </html>

```

Notice that the `pattern` attribute is handed a regular expression to denote that the email address must include an `@` symbol and a `.edu`.

- You can learn more about regular expressions from [Mozilla's documentation](#). Further, you can make inquiries to [cs50.ai](#) for hints.

CSS

- `css`, or *cascading style sheet*, is a markup language that allows you to fine-tune the aesthetics of your HTML files.
- CSS is filled with *properties*, which include key-value pairs.
- In your terminal, type `code home.html` and write code as follows:

```

1 <!DOCTYPE html>

```

```

2
3 <!-- Demonstrates inline CSS with P tags -->
4
5 <html lang="en">
6   <head>
7     <title>css</title>
8   </head>
9   <body>
10    <p style="font-size: large; text-align: center;">
11      John Harvard
12    </p>
13    <p style="font-size: medium; text-align: center;">
14      Welcome to my home page!
15    </p>
16    <p style="font-size: small; text-align: center;">
17      Copyright &#169; John Harvard
18    </p>
19   </body>
20 </html>

```

Notice that some `style` attributes are provided to the `<p>` tags. The `font-size` is set to `large`, `medium`, or `small`. Then `text-align` is set to center.

- While correct, the above is not well-designed. We can remove redundancy by modifying our code as follows:

```

1 <!DOCTYPE html>
2
3 <!-- Removes outer DIV -->
4
5 <html lang="en">
6   <head>
7     <title>css</title>
8   </head>
9   <body style="text-align: center;">
10    <div style="font-size: large">
11      John Harvard
12    </div>
13    <div style="font-size: medium">
14      Welcome to my home page!
15    </div>
16    <div style="font-size: small">
17      Copyright &#169; John Harvard
18    </div>
19   </body>
20 </html>

```

Notice that `<div>` tags are used to divide up this HTML file into specific regions. `text-align: center` is invoked on the entire body of the HTML. Because everything inside `body` is a child of `body`, the `center` attribute cascades down to those children.

- It turns out that there are newer semantic tags that are included in HTML. We can modify our code as follows:

```

1  <!DOCTYPE html>
2
3  <!-- Uses semantic tags instead of DIVs -->
4
5  <html lang="en">
6      <head>
7          <title>css</title>
8      </head>
9      <body style="text-align: center">
10         <header style="font-size: large">
11             John Harvard
12         </header>
13         <main style="font-size: medium">
14             Welcome to my home page!
15         </main>
16         <footer style="font-size: small">
17             Copyright &#169; John Harvard
18         </footer>
19     </body>
20 </html>

```

Notice that the `header` and `footer` both have different styles assigned to them.

- This practice of placing the style and information all in the same location is not good practice. We could move the elements of style to the top of the file as follows:

```

1  <!-- Demonstrates class selectors -->
2
3  <html lang="en">
4      <head>
5          <style>
6
7              .centered
8              {
9                  text-align: center;
10             }
11
12              .large
13              {
14                  font-size: large;
15             }
16
17              .medium
18              {
19                  font-size: medium;
20             }
21
22              .small

```

```

23      {
24          font-size: small;
25      }
26
27      </style>
28      <title>css</title>
29  </head>
30  <body class="centered">
31      <header class="large">
32          John Harvard
33      </header>
34      <main class="medium">
35          Welcome to my home page!
36      </main>
37      <footer class="small">
38          Copyright © John Harvard
39      </footer>
40  </body>
41 </html>

```

Notice all the style tags are placed up in the `head` in the `style` tag wrapper. Also notice that we've assigned *classes*, called `centered`, `large`, `medium`, and `small` to our elements, and that we select those classes by placing a dot before the name, as in `.centered`

- It turns out that we can move all our style code into a special file called a *CSS* file. We can create a file called `style.css` and paste our classes there:

```

1  .centered
2  {
3      text-align: center;
4  }
5
6  .large
7  {
8      font-size: large;
9  }
10
11 .medium
12 {
13     font-size: medium;
14 }
15
16 .small
17 {
18     font-size: small;
19 }

```

Notice that this is verbatim what appeared in our HTML file.

- We then can tell the browser where to locate the CSS for this HTML file:

```

1 <!DOCTYPE html>
2
3 <!-- Demonstrates external stylesheets -->
4
5 <html lang="en">
6   <head>
7     <link href="style.css" rel="stylesheet">
8     <title>css</title>
9   </head>
10  <body class="centered">
11    <header class="large">
12      John Harvard
13    </header>
14    <main class="medium">
15      Welcome to my home page!
16    </main>
17    <footer class="small">
18      Copyright © John Harvard
19    </footer>
20  </body>
21 </html>

```

Notice that `style.css` is linked to this HTML file as a stylesheet, telling the browser where to locate the styles we created.

Frameworks

- Similar to third-party libraries we can leverage in Python, there are third-party libraries called *frameworks* that we can utilize with our HTML files.
- Bootstrap* is one of these frameworks that we can use to beautify our HTML and easily perfect design elements such that our pages are more readable.
- Bootstrap can be utilized by adding the following `link` tag in the `head` of your html file:

```

1 <head>
2   <link
3     href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
4     rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN"
5     crossorigin="anonymous">
6   <title>favorites</title>
7 </head>

```

- Consider the following HTML:

```

1 <!DOCTYPE html>
2
3 <!-- Demonstrates table -->
4
5 <html lang="en">

```

```

6   <head>
7     <title>phonebook</title>
8   </head>
9   <body>
10    <table>
11      <thead>
12        <tr>
13          <th>Name</th>
14          <th>Number</th>
15        </tr>
16      </thead>
17      <tbody>
18        <tr>
19          <td>Carter</td>
20          <td>+1-617-495-1000</td>
21        </tr>
22        <tr>
23          <td>David</td>
24          <td>+1-617-495-1000</td>
25        </tr>
26        <tr>
27          <td>John</td>
28          <td>+1-949-468-2750</td>
29        </tr>
30      </tbody>
31    </table>
32  </body>
33 </html>

```

Notice how when looking at a served version of this page, it's quite plain.

- Now consider the following HTML that implements the use of Bootstrap:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates table with Bootstrap -->
4
5  <html lang="en">
6    <head>
7      <link
8        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
9        rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwykc2MPK8M2HN"
10       crossorigin="anonymous">
11      <title>phonebook</title>
12    </head>
13    <body>
14      <table class="table">
15        <thead>

```

```

16         </tr>
17     </thead>
18     <tbody>
19         <tr>
20             <td>Carter</td>
21             <td>+1-617-495-1000</td>
22         </tr>
23         <tr>
24             <td>David</td>
25             <td>+1-949-468-2750</td>
26         </tr>
27     </tbody>
28   </table>
29 </body>
30 </html>

```

Notice how much prettier this website is now.

- Similarly, consider to the following expansion of our search page created earlier:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates layout with Bootstrap -->
4
5  <html lang="en">
6      <head>
7          <link
8              href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
9                  rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwykc2MPK8M2HN"
10                 crossorigin="anonymous">
11          <title>search</title>
12      </head>
13      <body>
14
15          <div class="container-fluid">
16
17              <ul class="m-3 nav">
18                  <li class="nav-item">
19                      <a class="nav-link text-dark"
20                          href="https://about.google/">About</a>
21                  </li>
22                  <li class="nav-item">
23                      <a class="nav-link text-dark"
24                          href="https://store.google.com/">Store</a>
25                  </li>
26                  <li class="nav-item ms-auto">
27                      <a class="nav-link text-dark"
28                          href="https://www.google.com/gmail/">Gmail</a>
29                  </li>
30              <li class="nav-item">

```

```

25          <a class="nav-link text-dark"
26            href="https://www.google.com/imghp">Images</a>
27        </li>
28        <li class="nav-item">
29          <a class="nav-link text-dark"
30            href="https://www.google.com/intl/en/about/products">
31              <svg xmlns="http://www.w3.org/2000/svg" width="16"
32                height="16" fill="currentColor" class="bi bi-grid-3x3-gap-fill" viewBox="0 0 16
33                16">
34                <path d="M1 2a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0
35                1-1 1H2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-h2a1 1 0 0 1 1v2a1 1 0 0 1-1
36                1H7a1 1 0 0 1 1-1V2zm5 0a1 1 0 0 1 1-h2a1 1 0 0 1 1v2a1 1 0 0 1-1
37                1h-2a1 1 0 0 1-1V2zM1 7a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
38                1h-2a1 1 0 0 1 1V2a1 1 0 0 1-1 1H2a1 1 0 0 1-1-V7zm5 0a1 1 0 0 1 1-
39                1h2a1 1 0 0 1 1v2a1 1 0 0 1-1 1H7a1 1 0 0 1-1-V7zm5 0a1 1 0 0 1 1-
40                1h2a1 1 0 0 1 1v2a1 1 0 0 1-1 1h-2a1 1 0 0 1-1-V7zM1 12a1 1 0 0 1 1-1h2a1 1 0
41                0 1 1 1v2a1 1 0 0 1-1 1H2a1 1 0 0 1-1-v-2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
42                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
43                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
44                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
45                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
46                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
47                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
48                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
49                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
50                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
51                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
52                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
53                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
54                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1
55                1h-2a1 1 0 0 1-1V2zm5 0a1 1 0 0 1 1-1h2a1 1 0 0 1 1v2a1 1 0 0 1-1

```

This version of the page is exceedingly stylized, thanks to Bootstrap.

- You can learn more about this in the [Bootstrap Documentation](#).

JavaScript

- JavaScript is another programming language that allows for interactivity within web pages.
- Consider the following implementation of `hello.html` that includes both JavaScript and HTML:

```
1  <!DOCTYPE html>
2
3  <!-- Demonstrates onsubmit -->
4
5  <html lang="en">
6      <head>
7          <script>
8
9              function greet()
10             {
11                 alert('hello, ' + document.querySelector('#name').value);
12             }
13
14         </script>
15         <title>hello</title>
16     </head>
17     <body>
18         <form onsubmit="greet(); return false;">
19             <input autocomplete="off" autofocus id="name" placeholder="Name"
20 type="text">
21             <input type="submit">
22         </form>
23     </body>
24 </html>
```

Notice how this form uses an `onsubmit` property to trigger a `script` found at the top of the file. The script uses `alert` to create an alert pop-up. `#name.value` goes to the textbox on the page and obtains the value typed by the user.

- Generally, it's considered bad design to mix onsubmit and JavaScript. We can advance our code as follows:

```
1  <!DOCTYPE html>
2
3  <!-- Demonstrates DOMContentLoaded -->
4
5  <html lang="en">
6      <head>
7          <script>
8
9              document.addEventListener('DOMContentLoaded', function() {
10                  document.querySelector('form').addEventListener('submit',
11 function(e) {
12                     alert('hello, ' + document.querySelector('#name').value);
13                     e.preventDefault();
14                 }
15             )
16         )
17     </head>
18     <body>
19         <form>
20             <input type="text" id="name" placeholder="Name">
21             <input type="submit" value="Submit">
22         </form>
23     </body>
24 </html>
```

```

13     });
14   });
15
16   </script>
17   <title>hello</title>
18 </head>
19 <body>
20   <form>
21     <input autocomplete="off" autofocus id="name" placeholder="Name"
22       type="text">
23     <input type="submit">
24   </form>
25 </body>
</html>

```

Notice that this version of the code creates an `addEventListener` to listen to the form `submit` being triggered. Notice how `DOMContentLoaded` ensures that the whole page is loaded before executing the JavaScript.

- JavaScript allows you to dynamically read and modify the html document loaded into memory such that the user need not reload to see changes.
- Consider the following HTML:

```

1  <!DOCTYPE html>
2
3  <!-- Demonstrates programmatic changes to style -->
4
5  <html lang="en">
6    <head>
7      <title>background</title>
8    </head>
9    <body>
10      <button id="red">R</button>
11      <button id="green">G</button>
12      <button id="blue">B</button>
13      <script>
14
15        let body = document.querySelector('body');
16        document.querySelector('#red').addEventListener('click', function() {
17          body.style.backgroundColor = 'red';
18        });
19        document.querySelector('#green').addEventListener('click', function()
{
20          body.style.backgroundColor = 'green';
21        });
22        document.querySelector('#blue').addEventListener('click', function()
{
23          body.style.backgroundColor = 'blue';
24        });
25
26      </script>

```

```
27     </body>
28 </html>
```

Notice that JavaScript listens for when a specific button is clicked. Upon such a click, certain style attributes on the page are changed. `body` is defined as the body of the page. Then, an event listener waits for the clicking of one of the buttons. Then, the `body.style.backgroundColor` is changed.

- Similarly, consider the following:

```
1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <script>
6
7              // Toggles visibility of greeting
8              function blink()
9              {
10                  let body = document.querySelector('body');
11                  if (body.style.visibility == 'hidden')
12                  {
13                      body.style.visibility = 'visible';
14                  }
15                  else
16                  {
17                      body.style.visibility = 'hidden';
18                  }
19              }
20
21          // Blink every 500ms
22          window.setInterval(blink, 500);
23
24      </script>
25      <title>blink</title>
26  </head>
27  <body>
28      hello, world
29  </body>
30 </html>
```

This example blinks a text at a set interval. Notice that `window.setInterval` takes in two arguments: A function to be called and a wait period (in milliseconds) between function calls.

- Consider the following:

```
1  <!DOCTYPE html>
2
3  <html lang="en">
4
5      <head>
6          <title>autocomplete</title>
```

```

7   </head>
8
9   <body>
10
11      <input autocomplete="off" autofocus placeholder="Query" type="text">
12
13      <ul></ul>
14
15      <script src="large.js"></script>
16      <script>
17
18          let input = document.querySelector('input');
19          input.addEventListener('keyup', function(event) {
20              let html = '';
21              if (input.value) {
22                  for (word of WORDS) {
23                      if (word.startsWith(input.value)) {
24                          html += `<li>${word}</li>`;
25                      }
26                  }
27              }
28              document.querySelector('ul').innerHTML = html;
29          });
30
31      </script>
32
33      </body>
34  </html>

```

This is a JavaScript implementation of autocomplete. This pulls from a file (not pictured here) called `large.js` that is a list of words.

- Interestingly, we can also geolocate using JavaScript:

```

1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <title>geolocation</title>
6      </head>
7      <body>
8          <script>
9
10         navigator.geolocation.getCurrentPosition(function(position) {
11             document.write(position.coords.latitude + ", " +
position.coords.longitude);
12         });
13
14         </script>
15     </body>
16 </html>

```

Notice that `navigator.geolocation` is used to `getCurrentPosition`. This will not work if your computer or browser does not allow for location tracking.

- The capabilities of JavaScript are many and can be found in the [JavaScript Documentation](#).

Summing Up

In this lesson, you learned how to create your own HTML files, style them, leverage third-party frameworks, and utilize JavaScript. Specifically, we discussed...

- TCP/IP
- DNS
- HTML
- Regular expressions.
- CSS
- Frameworks
- JavaScript

See you next time!

Lecture 9-Flask

- [Welcome!](#)
- [Static to Dynamic](#)
- [Flask](#)
- [Forms](#)
- [Layout](#)
- [POST](#)
- [Frosh IMs](#)
- [Flask and SQL](#)
- [Session](#)
- [Shopping Cart](#)
- [Shows](#)
- [AJAX and APIs](#)
- [JSON](#)
- [Summing Up](#)

Welcome!

- In previous weeks, you have learned numerous programming languages, techniques, and strategies.

- Indeed, this class has been far less of a *C class* or *Python class* and far more of a *programming class*, such that you can go on to follow future trends.
- In these past several weeks, you have learned *how to learn* about programming.
- Today, we will be moving from HTML and CSS into combining HTML, CSS, SQL, Python, and JavaScript so you can create your own web applications.

Static to Dynamic

- Up until this point, all HTML you saw was pre-written and static.
- In the past, when you visited a page, the browser downloaded an HTML page, and you were able to view it. These are considered *static* pages, in that what programmed in the HTML is exactly what the user sees and downloads to their internet browser.
- Dynamic pages refer to the ability of Python and similar languages to create HTML files on-the-fly. Accordingly, you can have web pages that are generated by options selected by your user.
- You have used `http-server` in the past to serve your web pages. Today, we are going to utilize a new server that can parse out a web address and perform actions based on the URL provided.

Flask

- Flask* is a third-party library that allows you to host web applications using the Flask framework, or a micro-framework, within Python.
- You can run flask by executing `flask run` in your terminal window in [cs50.dev](#).
- To do so, you will need a file called `app.py` and a folder called `templates`.
- To get started, create a folder called `templates` and create a file called `index.html` with the following code:

```

1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <meta name="viewport" content="initial-scale=1, width=device-width">
6          <title>hello</title>
7      </head>
8      <body>
9          hello, {{ name }}
10     </body>
11 </html>
12

```

Notice the double `{{ name }}` that is a placeholder for something that will be later provided by our Flask server.

- Then, in the same folder that the `templates` folder appears, create a file called `app.py` and add the following code:

```

1 # Uses request.args.get
2
3 from flask import Flask, render_template, request
4
5 app = Flask(__name__)
6
7
8 @app.route("/")
9 def index():
10     name = request.args.get("name", "world")
11     return render_template("index.html", name=name)

```

- Notice that this code defines `app` as the Flask application. Then, it defines the `/` route of `app` as returning the contents of `index.html` with the argument of `name`. By default, the `request.args.get` function will look for the `name` being provided by the user. If no name is provided, it will default to `world`.
- `@app.route` is otherwise known as a decorator.
- Finally, add a final file in the same folder as `app.py` called `requirements.txt` that has only a single line of code:

Notice only `Flask` appears in this file.

- You can run this file by typing `flask run` in the terminal window. If Flask does not run, ensure that your syntax is correct in each of the files above. Further, if Flask will not run, make sure your files are organized as follows:

```

1 /templates
2     index.html
3 app.py
4 requirements.txt

```

- Once you get it running, you will be prompted to click a link. Once you navigate to that webpage, try adding `?name=[Your Name]` to the base URL in your browser's URL bar.

Forms

- Improving upon our program, we know that most users will not type arguments into the address bar. Instead, programmers rely upon users to fill out forms on web pages. Accordingly, we can modify `index.html` as follows:

```

1 <!DOCTYPE html>
2
3 <html lang="en">
4     <head>
5         <meta name="viewport" content="initial-scale=1, width=device-width">
6         <title>hello</title>
7     </head>
8     <body>
9         <form action="/greet" method="get">

```

```

10         <input autocomplete="off" autofocus name="name" placeholder="Name"
11             type="text">
12             <button type="submit">Greet</button>
13         </form>
14     </body>
15 </html>

```

Notice that a form is now created that takes the user's name and then passes it off to a route called `/greet`.

- Further, we can change `app.py` as follows:

```

1 # Adds a form, second route
2
3 from flask import Flask, render_template, request
4
5 app = Flask(__name__)
6
7
8 @app.route("/")
9 def index():
10     return render_template("index.html")
11
12
13 @app.route("/greet")
14 def greet():
15     return render_template("greet.html", name=request.args.get("name", "world"))

```

Notice that the default path will display a form for the user to input their name. The `/greet` route will pass the `name` to that web page.

- To finalize this implementation, you will need another template for `greet.html` in the `templates` folder as follows:

```

1 <!DOCTYPE html>
2
3 <html lang="en">
4     <head>
5         <meta name="viewport" content="initial-scale=1, width=device-width">
6         <title>hello</title>
7     </head>
8     <body>
9         hello, {{ name }}
10    </body>
11 </html>

```

Notice that this route will now render the greeting to the user, followed by their name.

Layout

- Both of our web pages, `index.html` and `greet.html`, have much of the same data. Wouldn't it be nice to allow the body to be unique, but copy the same layout from page to page?
- First, create a new template called `layout.html` and write code as follows:

```

1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <meta name="viewport" content="initial-scale=1, width=device-width">
6          <title>hello</title>
7      </head>
8      <body>
9          {% block body %}{% endblock %}
10     </body>
11 </html>
```

Notice that the `{% block body %}{% endblock %}` allows for the insertion of other code from other HTML files.

- Then, modify your `index.html` as follows:

```

1  {% extends "layout.html" %}
2
3  {% block body %}
4
5      <form action="/greet" method="get">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              <button type="submit">Greet</button>
9      </form>
10  {% endblock %}
```

Notice that the line `{% extends "layout.html" %}` tells the server where to get the layout of this page. Then, the `{% block body %}{% endblock %}` tells what code to be inserted into `layout.html`.

- Finally, change `greet.html` as follows:

```

1  {% extends "layout.html" %}
2
3  {% block body %}
4      hello, {{ name }}
5  {% endblock %}
```

Notice how this code is shorter and more compact.

POST

- You can imagine scenarios where it is not safe to utilize `get`, as usernames and passwords would show up in the URL.

- We can utilize the method `post` to help with this problem by modifying `app.py` as follows:

```

1 # Switches to POST
2
3 from flask import Flask, render_template, request
4
5 app = Flask(__name__)
6
7
8 @app.route("/")
9 def index():
10     return render_template("index.html")
11
12
13 @app.route("/greet", methods=["POST"])
14 def greet():
15     return render_template("greet.html", name=request.form.get("name", "world"))

```

Notice that `POST` is added to the `/greet` route, and that we use `request.form.get` rather than `request.args.get`.

- This tells the server to look *deeper* in the virtual envelope and not reveal the items in `post` in the URL.
- Still, this code can be advanced further by utilizing a single route for both `get` and `post`. To do this, modify `app.py` as follows:

```

1 # Uses a single route
2
3 from flask import Flask, render_template, request
4
5 app = Flask(__name__)
6
7
8 @app.route("/", methods=["GET", "POST"])
9 def index():
10     if request.method == "POST":
11         return render_template("greet.html", name=request.form.get("name",
12 "world"))
12     return render_template("index.html")

```

Notice that both `get` and `post` are done in a single routing. However, `request.method` is utilized to properly route based upon the type of routing requested by the user.

- Accordingly, you can modify your `index.html` as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}

4

5      <form action="/" method="post">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              <button type="submit">Greet</button>
9
10  {% endblock %}

```

Notice that the form `action` is changed.

- Still, there is a bug still in this code. With our new implementation, when someone types in no name into the form, `Hello,` is displayed without a name. We can improve our code by editing `app.py` as follows:

```

1  # Moves default value to template
2
3  from flask import Flask, render_template, request
4
5  app = Flask(__name__)
6
7
8  @app.route("/", methods=[ "GET", "POST" ])
9  def index():
10      if request.method == "POST":
11          return render_template("greet.html", name=request.form.get("name"))
12      return render_template("index.html")

```

Notice that `name=request.form.get("name"))` is changed.

- Finally, change `greet.html` as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}

4

5      hello, {% if name %}{{ name }}{% else %}world{% endif %}

6
7  {% endblock %}

```

Notice how `hello,` is changed to allow for a default output when no name is identified.

- As we've been changing many files, you may wish to compare your final code with [our final code](#).

Frosh IMs

- Frosh IMs or *froshims* is a web application that allows students to register for intramural sports.

- Close all your `hello` related windows and create a folder by typing `mkdir froshims` in the terminal window. Then, type `cd froshims` to browse to this folder. Within, create a directory called `templates` by typing `mkdir templates`. Finally, type `code app.py` and write code as follows:

```

1 # Implements a registration form using a select menu, validating sport server-
2 # side
3
4 from flask import Flask, render_template, request
5
6 app = Flask(__name__)
7
8 SPORTS = [
9     "Basketball",
10    "Soccer",
11    "Ultimate Frisbee"
12]
13
14 @app.route("/")
15 def index():
16     return render_template("index.html", sports=SPORTS)
17
18
19 @app.route("/register", methods=["POST"])
20 def register():
21
22     # Validate submission
23     if not request.form.get("name") or request.form.get("sport") not in SPORTS:
24         return render_template("failure.html")
25
26     # Confirm registration
27     return render_template("success.html")

```

Notice that a `failure` option is provided, such that a failure message will be displayed to the user if the `name` or `sport` field is not properly filled out.

- Next, create a file in the `templates` folder called `index.html` by typing `code templates/index.html` and write code as follows:

```

1 {% extends "layout.html" %}
2
3 {% block body %}
4     <h1>Register</h1>
5     <form action="/register" method="post">
6         <input autocomplete="off" autofocus name="name" placeholder="Name"
7             type="text">
8         <select name="sport">
9             <option disabled selected value="">Sport</option>
10            {% for sport in sports %}
11                <option value="{{ sport }}>{{ sport }}</option>
12            {% endfor %}

```

```
12     </select>
13     <button type="submit">Register</button>
14   </form>
15 {%- endblock %}
```

- Next, create a file called `layout.html` by typing `code templates/layout.html` and write code as follows:

```
1  <!DOCTYPE html>
2
3  <html lang="en">
4    <head>
5      <meta name="viewport" content="initial-scale=1, width=device-width">
6      <title>froshims</title>
7    </head>
8    <body>
9      {% block body %}{% endblock %}
10   </body>
11 </html>
```

- Fourth, create a file in templates called `success.html` as follows:

```
1  {% extends "layout.html" %}
2
3  {% block body %}
4    You are registered!
5  {% endblock %}
```

- Finally, create a file in templates called `failure.html` as follows:

```
1  {% extends "layout.html" %}
2
3  {% block body %}
4    You are not registered!
5  {% endblock %}
```

- You can imagine how we might want to see the various registration options using radio buttons. We can improve `index.html` as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}
4      <h1>Register</h1>
5      <form action="/register" method="post">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              {% for sport in sports %}
9                  <input name="sport" type="radio" value="{{ sport }}> {{ sport }}
10             {% endfor %}
11             <button type="submit">Register</button>
12         </form>
13     {% endblock %}

```

- Executing `flask run` you can see how the interface has now changed.
- We can further improve upon our program by enabling checkboxes. Modify `index.html` as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}
4      <h1>Register</h1>
5      <form action="/register" method="post">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              {% for sport in sports %}
9                  <input name="sport" type="checkbox" value="{{ sport }}> {{ sport }}
10             {% endfor %}
11             <button type="submit">Register</button>
12         </form>
13     {% endblock %}

```

Notice that `type` is changed to `checkbox`.

- To implement this, we will need to modify `app.py`:

```

1  # Implements a registration form using checkboxes

2

3  from flask import Flask, render_template, request

4

5  app = Flask(__name__)

6

7  SPORTS = [
8      "Basketball",
9      "Soccer",
10     "Ultimate Frisbee"
11 ]
12
13
14 @app.route("/")
15 def index():

```

```

16     return render_template("index.html", sports=SPORTS)
17
18
19 @app.route("/register", methods=["POST"])
20 def register():
21
22     # Validate submission
23     if not request.form.get("name"):
24         return render_template("failure.html")
25     for sport in request.form.getlist("sport"):
26         if sport not in SPORTS:
27             return render_template("failure.html")
28
29     # Confirm registration
30     return render_template("success.html")

```

Notice how `for sport in` allows iteration through all the sports selected by the user.

- You can imagine how we might want to accept the registration of many different registrants. We can improve `app.py` as follows:

```

1  # Implements a registration form, storing registrants in a dictionary, with error
2  # messages
3
4  from flask import Flask, redirect, render_template, request
5
6  app = Flask(__name__)
7
8  REGISTRANTS = {}
9
10 SPORTS = [
11     "Basketball",
12     "Soccer",
13     "Ultimate Frisbee"
14 ]
15
16 @app.route("/")
17 def index():
18     return render_template("index.html", sports=SPORTS)
19
20
21 @app.route("/register", methods=["POST"])
22 def register():
23
24     # Validate name
25     name = request.form.get("name")
26     if not name:
27         return render_template("error.html", message="Missing name")
28
29     # Validate sport
30     sport = request.form.get("sport")

```

```

31     if not sport:
32         return render_template("error.html", message="Missing sport")
33     if sport not in SPORTS:
34         return render_template("error.html", message="Invalid sport")
35
36     # Remember registrant
37     REGISTRANTS[name] = sport
38
39     # Confirm registration
40     return redirect("/registrants")
41
42
43 @app.route("/registrants")
44 def registrants():
45     return render_template("registrants.html", registrants=REGISTRANTS)

```

Notice that a dictionary called `REGISTRANTS` is used to log the `sport` selected by `REGISTRANTS[name]`. Also, notice that `registrants=REGISTRANTS` passes the dictionary on to this template.

- Additionally, we can implement `error.html`:

```

1  {% extends "layout.html" %}
2
3  {% block body %}
4      <h1>Error</h1>
5      <p>{{ message }}</p>
6      
7  {% endblock %}

```

- Further, create a new template called `registrants.html` as follows:

```

1  {% extends "layout.html" %}
2
3  {% block body %}
4      <h1>Registrants</h1>
5      <table>
6          <thead>
7              <tr>
8                  <th>Name</th>
9                  <th>Sport</th>
10             </tr>
11         </thead>
12         <tbody>
13             {% for name in registrants %}
14                 <tr>
15                     <td>{{ name }}</td>
16                     <td>{{ registrants[name] }}</td>
17                 </tr>
18             {% endfor %}
19         </tbody>
20     </table>

```

```
21 | {%- endblock %}
```

Notice that `{% for name in registrants %}...{% endfor %}` will iterate through each of the registrants. Very powerful to be able to iterate on a dynamic web page!

- You now have a web application! However, there are some security flaws! Because everything is client-side, an adversary could change the HTML and *hack* a website. Further, this data will not persist if the server is shut down. Could there be some way we could have our data persist even when the server restarts?

Flask and SQL

- Just as we have seen how Python can interface with a SQL database, we can combine the power of Flask, Python, and SQL to create a web application where data will persist!
- To implement this, you will need to take a number of steps.
- First, modify `requirements.txt` as follows:
- Modify `index.html` as follows:

```
1  {% extends "layout.html" %}

2

3  {% block body %}
4      <h1>Register</h1>
5      <form action="/register" method="post">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              {% for sport in sports %}
9                  <input name="sport" type="radio" value="{{ sport }}> {{ sport }}
10                 {% endfor %}
11                 <button type="submit">Register</button>
12             </form>
13     {% endblock %}
```

- Modify `layout.html` as follows:

```
1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <meta name="viewport" content="initial-scale=1, width=device-width">
6          <title>froshims</title>
7      </head>
8      <body>
9          {% block body %}{% endblock %}
10     </body>
11 </html>
```

- Ensure `failure.html` appears as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}
4      You are not registered!
5  {% endblock %}

```

- Modify `registrants.html` to appear as follows:

```

1  {% extends "layout.html" %}

2

3  {% block body %}
4      <h1>Registrants</h1>
5      <table>
6          <thead>
7              <tr>
8                  <th>Name</th>
9                  <th>Sport</th>
10                 <th></th>
11             </tr>
12         </thead>
13         <tbody>
14             {% for registrant in registrants %}
15                 <tr>
16                     <td>{{ registrant.name }}</td>
17                     <td>{{ registrant.sport }}</td>
18                     <td>
19                         <form action="/deregister" method="post">
20                             <input name="id" type="hidden" value="{{
21                             registrant.id }}">
22                             <button type="submit">Deregister</button>
23                         </form>
24                     </td>
25                 </tr>
26             {% endfor %}
27         </tbody>
28     {% endblock %}

```

Notice that a hidden value `registrant.id` is included such that it's possible to use this `id` later in `app.py`

- Finally, modify `app.py` as follows:

```

1  # Implements a registration form, storing registrants in a SQLite database, with
2  # support for deregistration
3
4  from cs50 import SQL
5  from flask import Flask, redirect, render_template, request
6
7  app = Flask(__name__)

```

```

8 db = SQL("sqlite:///froshims.db")
9
10 SPORTS = [
11     "Basketball",
12     "Soccer",
13     "Ultimate Frisbee"
14 ]
15
16
17 @app.route("/")
18 def index():
19     return render_template("index.html", sports=SPORTS)
20
21
22 @app.route("/deregister", methods=["POST"])
23 def deregister():
24
25     # Forget registrant
26     id = request.form.get("id")
27     if id:
28         db.execute("DELETE FROM registrants WHERE id = ?", id)
29     return redirect("/registrants")
30
31
32 @app.route("/register", methods=["POST"])
33 def register():
34
35     # Validate submission
36     name = request.form.get("name")
37     sport = request.form.get("sport")
38     if not name or sport not in SPORTS:
39         return render_template("failure.html")
40
41     # Remember registrant
42     db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", name, sport)
43
44     # Confirm registration
45     return redirect("/registrants")
46
47
48 @app.route("/registrants")
49 def registrants():
50     registrants = db.execute("SELECT * FROM registrants")
51     return render_template("registrants.html", registrants=registrants)

```

Notice that the `cs50` library is utilized. A route is included for `register` for the `post` method. This route will take the name and sport taken from the registration form and execute a SQL query to add the `name` and the `sport` to the `registrants` table. The `deregister` routes to a SQL query that will grab the user's `id` and utilize that information to deregister this individual.

- You can read more in the [Flask documentation](#).

Session

- While the above code is useful from an administrative standpoint, where a back-office administrator could add and remove individuals from the database, one can imagine how this code is not safe to implement on a public server.
- For one, bad actors could make decisions on behalf of other users by hitting the deregister button – effectively deleting their recorded answer from the server.
- Web services like Google use login credentials to ensure users only have access to the right data.
- We can actually implement this itself using *cookies*. Cookies are small files that are stored on your computer, such that your computer can communicate with the server and effectively say, “I’m an authorized user that has already logged in.” This authorization through this cookie is called a *session*.
- In the simplest form, we can implement this by creating a folder called `login` and then adding the following files.
- First, create a file called `requirements.txt` that reads as follows:

Notice that in addition to `Flask`, we also include `Flask-Session`, which is required to support login sessions.

- Second, in a `templates` folder, create a file called `layout.html` that appears as follows:

```
1 <!DOCTYPE html>
2
3 <html lang="en">
4   <head>
5     <meta name="viewport" content="initial-scale=1, width=device-width">
6     <title>login</title>
7   </head>
8   <body>
9     {% block body %}{% endblock %}
10  </body>
11 </html>
```

Notice this provides a very simple layout with a title and a body.

- Third, create a file in the `templates` folder called `index.html` that appears as follows:

```
1 {% extends "layout.html" %}
2
3 {% block body %}
4
5   {% if name %}
6     You are logged in as {{ name }}. <a href="/logout">Log out</a>.
7   {% else %}
8     You are not logged in. <a href="/login">Log in</a>.
9   {% endif %}
10
11 {% endblock %}
```

Notice that this file looks to see if `session["name"]` exists (elaborated further in `app.py` below). If it does, it will display a welcome message. If not, it will recommend you browse to a page to log in.

- Fourth, create a file called `login.html` and add the following code:

```
1  {% extends "layout.html" %}

2

3  {% block body %}

4

5      <form action="/login" method="post">
6          <input autocomplete="off" autofocus name="name" placeholder="Name"
7              type="text">
8              <button type="submit">Log In</button>
9
10  {% endblock %}
```

Notice this is the layout of a basic login page.

- Finally, create a file called `app.py` and write code as follows:

```
1  from flask import Flask, redirect, render_template, request, session
2  from flask_session import Session
3
4  # Configure app
5  app = Flask(__name__)
6
7  # Configure session
8  app.config["SESSION_PERMANENT"] = False
9  app.config["SESSION_TYPE"] = "filesystem"
10 Session(app)
11
12
13 @app.route("/")
14 def index():
15     return render_template("index.html", name=session.get("name"))
16
17
18 @app.route("/login", methods=[ "GET", "POST"])
19 def login():
20     if request.method == "POST":
21         session[ "name" ] = request.form.get( "name" )
22         return redirect("/")
23     return render_template("login.html")
24
25
26 @app.route("/logout")
27 def logout():
28     session.clear()
29     return redirect("/")
```

Notice the modified *imports* at the top of the file, including `session`, which will allow for you to support sessions. Most important, notice how `session["name"]` is used in the `login` and `logout` routes. The `login` route will assign the login name provided and assign it to `session["name"]`. However, in the `logout` route, the logging out is implemented by clearing the value of `session`.

- You can read more about sessions in the [Flask documentation](#).

Shopping Cart

- Moving on to a final example of utilizing Flask's ability to enable a session.
- We examined the following code for `store` in `app.py`. The following code was shown:

```
1  from cs50 import SQL
2  from flask import Flask, redirect, render_template, request, session
3  from flask_session import Session
4
5  # Configure app
6  app = Flask(__name__)
7
8  # Connect to database
9  db = SQL("sqlite:///store.db")
10
11 # Configure session
12 app.config["SESSION_PERMANENT"] = False
13 app.config["SESSION_TYPE"] = "filesystem"
14 Session(app)
15
16
17 @app.route("/")
18 def index():
19     books = db.execute("SELECT * FROM books")
20     return render_template("books.html", books=books)
21
22
23 @app.route("/cart", methods=[ "GET", "POST"])
24 def cart():
25
26     # Ensure cart exists
27     if "cart" not in session:
28         session["cart"] = []
29
30     # POST
31     if request.method == "POST":
32         book_id = request.form.get("id")
33         if book_id:
34             session["cart"].append(book_id)
35         return redirect("/cart")
36
37     # GET
38     books = db.execute("SELECT * FROM books WHERE id IN (?)", session["cart"])
39     return render_template("cart.html", books=books)
```

Notice that `cart` is implemented using a list. Items can be added to this list using the `Add to Cart` buttons in `books.html`. When clicking such a button, the `post` method is invoked, where the `id` of the item is appended to the `cart`. When viewing the cart, invoking the `get` method, SQL is executed to display a list of the books in the cart.

- You can see the rest of the files that power this `flask` implementation in the [source code](#).

Shows

- We looked at a pre-designed program called `shows`, in `app.py`:

```
1 # Searches for shows using LIKE
2
3 from cs50 import SQL
4 from flask import Flask, render_template, request
5
6 app = Flask(__name__)
7
8 db = SQL("sqlite:///shows.db")
9
10
11 @app.route("/")
12 def index():
13     return render_template("index.html")
14
15
16 @app.route("/search")
17 def search():
18     shows = db.execute("SELECT * FROM shows WHERE title LIKE ?",
19                         "%" + request.args.get("q") + "%")
20     return render_template("search.html", shows=shows)
```

Notice how the `search` route allows for a way by which to search for a `show`. This search looks for titles `LIKE` the one provided by the user.

- You can see the rest of the files of this implementation in the [source code](#).

AJAX and APIs

- An *application program interface* or *API* is a series of specifications that allow you to interface with another service. For example, we could utilize IMDB's API to interface with their database. We might even integrate APIs for handling specific types of data downloadable from a server.
- Improving upon `shows`, looking at an improvement of `app.py`, we saw the following:

```
1 # Searches for shows using Ajax
2
3 from cs50 import SQL
4 from flask import Flask, render_template, request
5
6 app = Flask(__name__)
7
```

```

8 db = SQL("sqlite:///shows.db")
9
10
11 @app.route("/")
12 def index():
13     return render_template("index.html")
14
15
16 @app.route("/search")
17 def search():
18     q = request.args.get("q")
19     if q:
20         shows = db.execute("SELECT * FROM shows WHERE title LIKE ? LIMIT 50", "%" +
21             + q + "%")
22     else:
23         shows = []
24     return render_template("search.html", shows=shows)

```

Notice that the `search` route executes a SQL query.

- Looking at `search.html`, you'll notice that it is very simple:

```

1  {% for show in shows %}
2      <li>{{ show["title"] }}</li>
3  {% endfor %}

```

Notice that it provides a bulleted list.

- Finally, looking at `index.html`, notice that *AJAX* code is utilized to power the search:

```

1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <meta name="viewport" content="initial-scale=1, width=device-width">
6          <title>shows</title>
7      </head>
8      <body>
9
10         <input autocomplete="off" autofocus placeholder="Query" type="search">
11
12         <ul></ul>
13
14         <script>
15
16             let input = document.querySelector('input');
17             input.addEventListener('input', async function() {
18                 let response = await fetch('/search?q=' + input.value);
19                 let shows = await response.text();
20                 document.querySelector('ul').innerHTML = shows;
21             });
22

```

```
23     </script>
24
25     </body>
26 </html>
```

Notice an event listener is utilized to dynamically query the server to provide a list that matches the title provided. This will locate the `ul` tag in the HTML and modify the web page accordingly to include the list of the matches.

- You can read more in the [AJAX documentation](#).

JSON

- *JavaScript Object Notation* or *JSON* is text file of dictionaries with keys and values. This is a raw, computer-friendly way to get lots of data.
- JSON is a very useful way of getting back data from the server.
- You can see this in action in the `index.html` we examined together:

```
1  <!DOCTYPE html>
2
3  <html lang="en">
4      <head>
5          <meta name="viewport" content="initial-scale=1, width=device-width">
6          <title>shows</title>
7      </head>
8      <body>
9
10         <input autocomplete="off" autofocus placeholder="Query" type="text">
11
12         <ul></ul>
13
14     <script>
15
16         let input = document.querySelector('input');
17         input.addEventListener('input', async function() {
18             let response = await fetch('/search?q=' + input.value);
19             let shows = await response.json();
20             let html = '';
21             for (let id in shows) {
22                 let title = shows[id].title.replace('<', '&lt;').replace('>', '&gt;');
23                 html += '<li>' + title + '</li>';
24             }
25             document.querySelector('ul').innerHTML = html;
26         });
27
28     </script>
29
30     </body>
31 </html>
```

While the above may be somewhat cryptic, it provides a starting point for you to research JSON on your own to see how it can be implemented in your own web applications.

- Further, we examined `app.py` to see how the JSON response is obtained:

```
1 # Searches for shows using Ajax with JSON
2
3 from cs50 import SQL
4 from flask import Flask, jsonify, render_template, request
5
6 app = Flask(__name__)
7
8 db = SQL("sqlite:///shows.db")
9
10
11 @app.route("/")
12 def index():
13     return render_template("index.html")
14
15
16 @app.route("/search")
17 def search():
18     q = request.args.get("q")
19     if q:
20         shows = db.execute("SELECT * FROM shows WHERE title LIKE ? LIMIT 50", "%" +
21             + q + "%")
22     else:
23         shows = []
24     return jsonify(shows)
```

Notice how `jsonify` is used to convert the result into a readable format acceptable by contemporary web applications.

- You can read more in the [JSON documentation](#).

Summing Up

In this lesson, you learned how to utilize Python, SQL, and Flask to create web applications. Specifically, we discussed...

- GET
- POST
- Flask
- Session
- APIs
- AJAX
- JSON

See you next time for our final lecture!

Lecture 10-Cybersecurity

Cybersecurity

- [Recap](#)
- [Looking Ahead](#)
- [Cybersecurity](#)
- [Passwords](#)
- [Phone Security](#)
- [Password Managers](#)
- [Two-factor Authentication](#)
- [Hashing](#)
- [Cryptography](#)
- [Passkeys](#)
- [Encryption](#)
- [Deletion](#)
- [Summing Up](#)

Recap

- Over these past ten weeks, you have been drinking from the proverbial firehose.
- While in this course, you learned how to program many programming languages; indeed, our great hope is that you *learned how to program* in all: Regardless of the programming language.
- Further, we hope that you *learned how to solve problems* above all else.

Looking Ahead

- As you journey from the work of this course to the world outside CS50, you may want to take a number of steps to prepare.
- To be able to execute commands on the terminal, much like you did on [CS50.dev](#), you can install command-line tools on your [Mac](#) or [PC](#).
- You can learn more about [Git](#).
- You can [download](#) and [learn](#) about VS Code.
- You can host a website using [GitHub](#) or [Netlify](#).
- You can host a web app using [AWS](#), [Azure](#), or [Google Cloud](#).
- You can ask questions in relevant online communities.
- You can ask questions using AI-based tools like [OpenAI](#) and [GitHub Copilot](#).
- You can take any of our other CS50 courses.

- You can join one of our many [communities](#).

Cybersecurity

- Today will be a high-level overview of some of the cybersecurity-related topics.
- Cybersecurity is understanding how our data is *secure* or *not secure*.

Passwords

- One cybersecurity concern relates to our passwords.
- Passwords are one method used to secure data online.
- There are common passwords that people use:

1	1. 123456
2	2. admin
3	3. 12345678
4	4. 123456789
5	5. 1234
6	6. 12345
7	7. password
8	8. 123
9	9. Aa123456
10	10. 1234567890

- If you have one of the passwords above, most likely, millions of people have the same password as you!
- Adversaries in the world will start with this list.
- Bad guys can also guess most of the heuristics you use to add symbols to your password.
- Adversaries can use *brute-force attacks*, using a dictionary of passwords to simply try every possible password.
- Your password is likely not as secure as you think it is.

Phone Security

- Many phones are secured by a four-digit code.
- The most simple form of attack would be to brute-force attempt all possible passwords.
- There are 10,000 possible passwords when using a four-digit code.
- If it takes one guess per second, it will take 10,000 seconds to crack the password.
- However, if a programmer creates a program to generate all possible codes, the time required would be minimal. Consider the following code in Python:

```
1 from string import digits
2 from itertools import product
3
4 for passcode in product(digits, repeat=4):
5     print(passcode)
```

- It should be quite disconcerting that the code above could take only a few seconds (at most!) to discover your password.
- We could improve our security by switching to a four-letter password. This would result in 7,311,616 possible passwords.
- Including uppercase and lowercase characters would create over 78 million possibilities.
- Consider how we could modify your code to discover these passwords:

```
1 from string import ascii_letters
2 from itertools import product
3
4 for passcode in product(ascii_letters, repeat=4):
5     print(passcode)
```

- We could even add the ability to look at all possible eight-digit passwords with letters, numbers, and punctuations:

```
1 from string import ascii_letters, digits, punctuation
2 from itertools import product
3
4 for passcode in product(ascii_letters + digits + punctuation, repeat=8):
5     print(passcode)
```

- Expanding to eight characters, including upper and lowercase letters, numbers, and symbols, would result in 6,095,689,385,410,816 possible combinations.
- In the digital world, you simply want your password to be better than other peoples' passwords such that others would be attacked far before you—as you are a much less convenient target.
- A downside of using such a long password is the downside of having to remember it.
- Accordingly, there are other defenses that could be employed to slow down an attacker. For example, some phone manufacturers lock out those who guess a password incorrectly.
- Security is about finding a “sweet spot” between the trade-offs of enhanced security while maintaining convenience.

Password Managers

- Password managers can be used to create very challenging passwords and remember them for you.
- The probability of a password secured by a password manager being broken is very, very low.
- You'd hope that such password managers are secure. However, if one gains access to your password manager, they would have access to all your passwords.

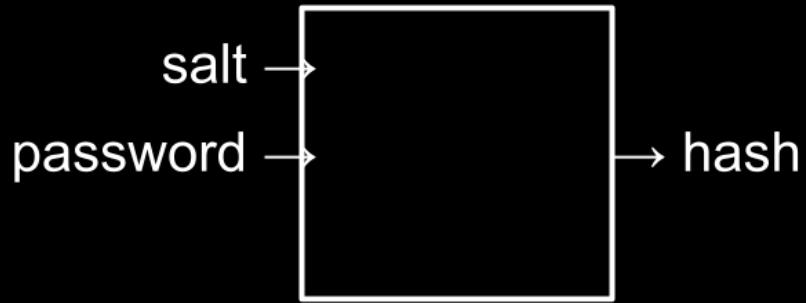
- In the end, you are far less likely to be at risk by those you live with—and much more likely to be at risk by the billions of other people on the internet.
- As mentioned prior, you can make a decision based on a balance between security and convenience.

Two-factor Authentication

- Adding another means by which you must authenticate adds further security. However, there is a human cost as you might not have access to your second factor.
- These are implemented as one-time passwords of sorts that are sent to an email, device, or phone number.
- Always, security policies attempt to balance the needs of security and human convenience.

Hashing

- Your account information and other sensitive data should not be stored as raw text in an online database.
- If a database becomes compromised and all credentials are stored in plain text, credentials for other services at other websites are likely also compromised.
- Hence, hashing algorithms, as discussed earlier in this course, are used to store only hashed values of passwords.
- One-way hashing allows online services to actually *never* store the original password typed by the user: Only the hashed value of these passwords. Accordingly, if there is a breach, only the hashed value will be known.
- *Rainbow tables* are huge dictionaries that adversaries use to attempt to pre-hash possible passwords as a means by which to attempt to break the hash algorithm.
- As an additional process to heightened security, programmers may sometimes introduce *salting* where it becomes unlikely that multiple users may have the same hash value to represent their passwords. You can imagine this as follows:

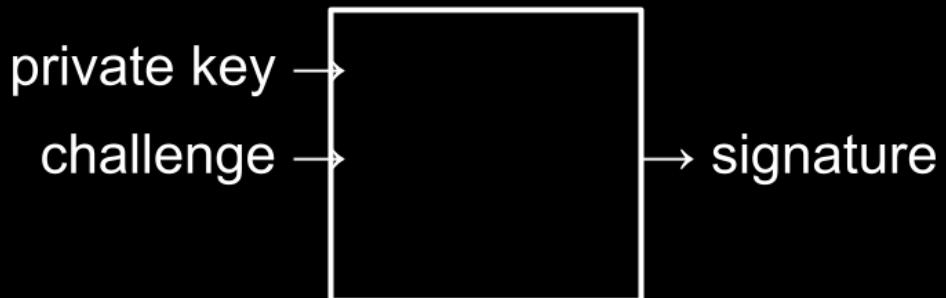


Cryptography

- Similar to hashing, a cipher algorithm can use a *public key* and text to create ciphertext.
- In turn, a *private key* and the ciphertext can be fed to the algorithm to decrypt the text.

Passkeys

- Passkeys are a new technology only emerging in the most recent months.
- Through private keys and a challenge being fed to an algorithm, websites can authenticate you through the unique signature created by your device.



- Hence, passwords and usernames may soon become obsolete.

Encryption

- Encryption is a way by which data is obscured such that only the sender and intended receiver can be read.
 - Early in this course, we learned a very simple algorithm to “shift” the text by one or more characters as a rudimentary form of encryption.
 - *End-to-end encryption* is a way by which encrypting and decrypting happen on the same system without a middleman. This prevents the middleman or a malicious actor from being able to snoop on your data. Zoom and Apple Messages can both utilize end-to-end encryption.

Deletion

- Trashing a file on your computer or emptying the trash can does not actually delete the actual bits of the file on your computer.
 - Instead, remnants of the files are left.

- *Secure deletion* is where the remnants of those files are turned into zeros and ones.
 - Still, some remnants may remain because of what is rendered inaccessible by the operating system.
 - *Full-disk encryption* allows your entire hard drive to be encrypted. Thus, your deleted files are less accessible to adversaries.
 - Considering encryption, it's this same technology that adversaries use to create *ransomware* that can, quite literally, hold your hard drive for ransom.

Summing Up

- Use a password manager.

- Use two-factor authentication.
- Use (end-to-end) encryption.