CS31: Introduction to Computer Systems

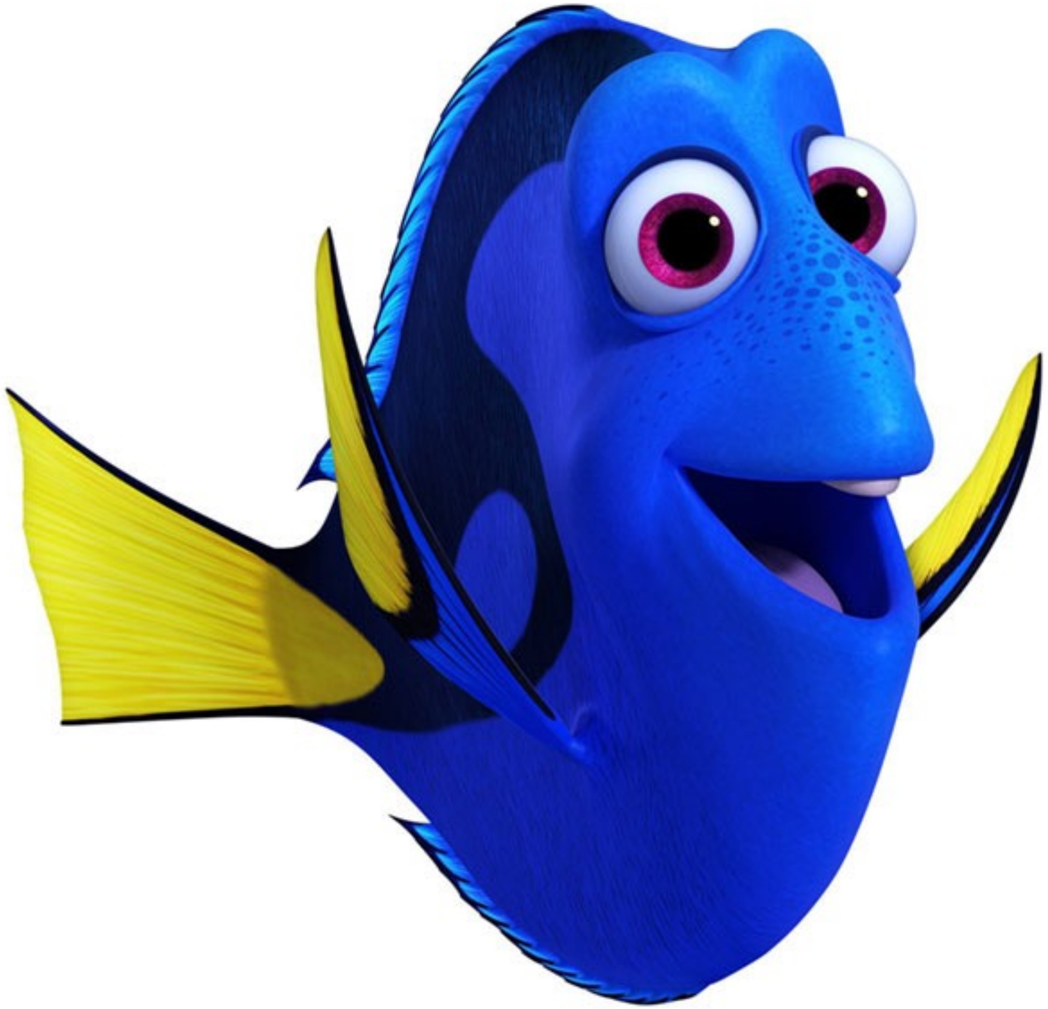
Week 1, Class 2

Introduction to C Programming

01/25/24

Dr. Sukrit Venkatagiri
Swarthmore College

Where are we?



C — programming language

↓ *compiled*

x86 Assembly — instruction set architecture

↓ *assembled*

Binary — logic / bits

↓

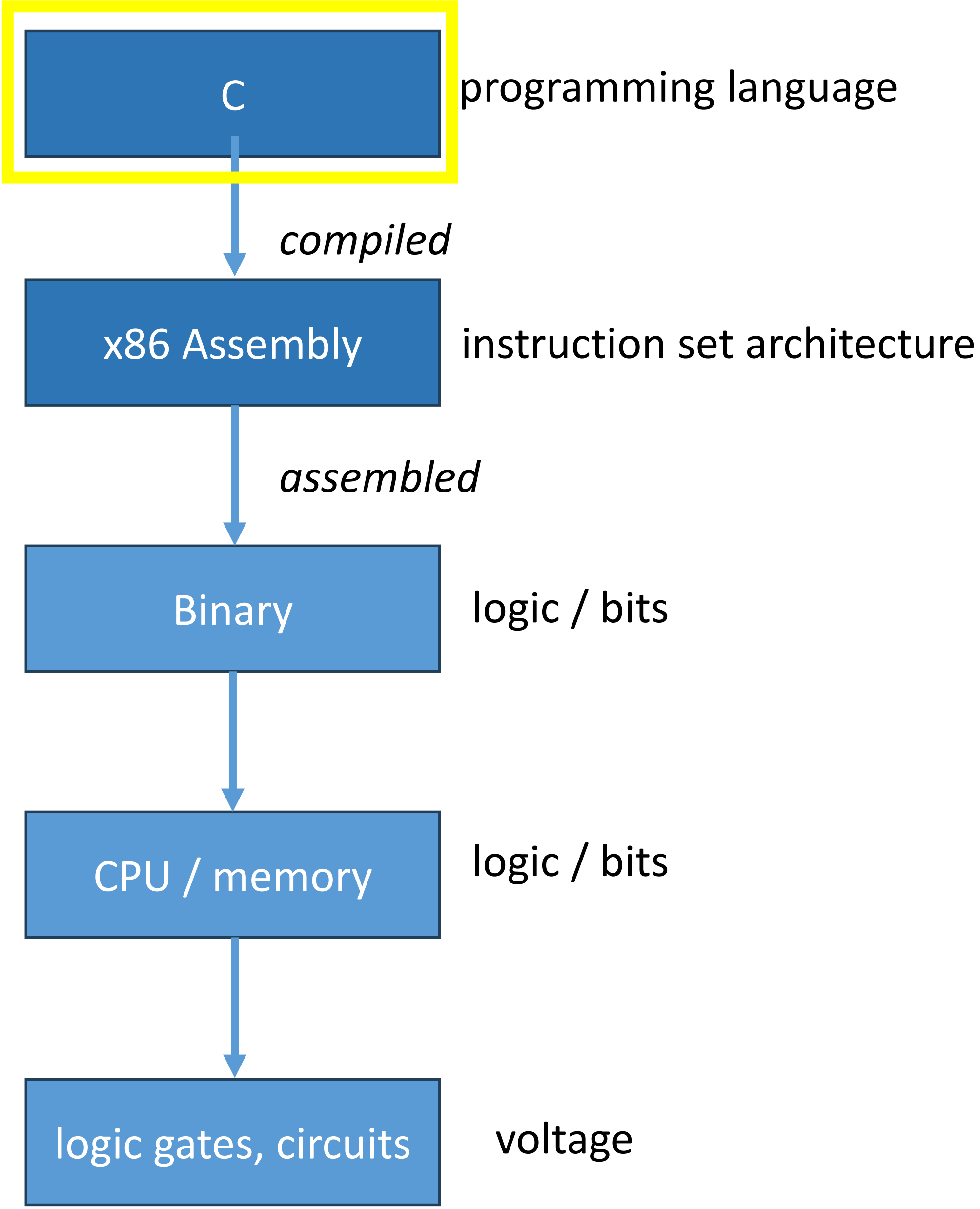
CPU / memory — logic / bits

↓

logic gates, circuits — voltage

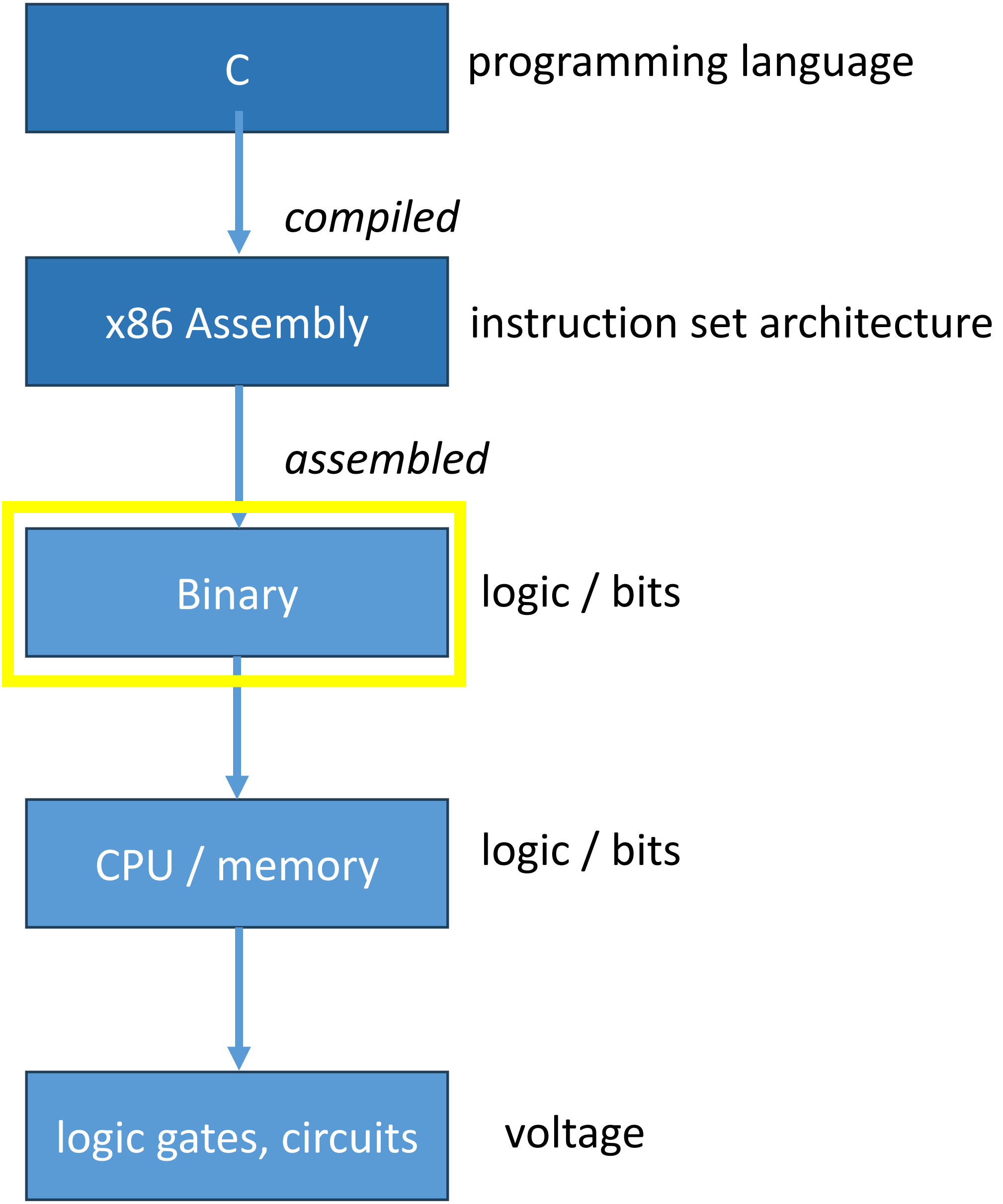
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	”
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	”
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	”
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	”
13	Threading	pthreads Game of Life
14	Threading	”



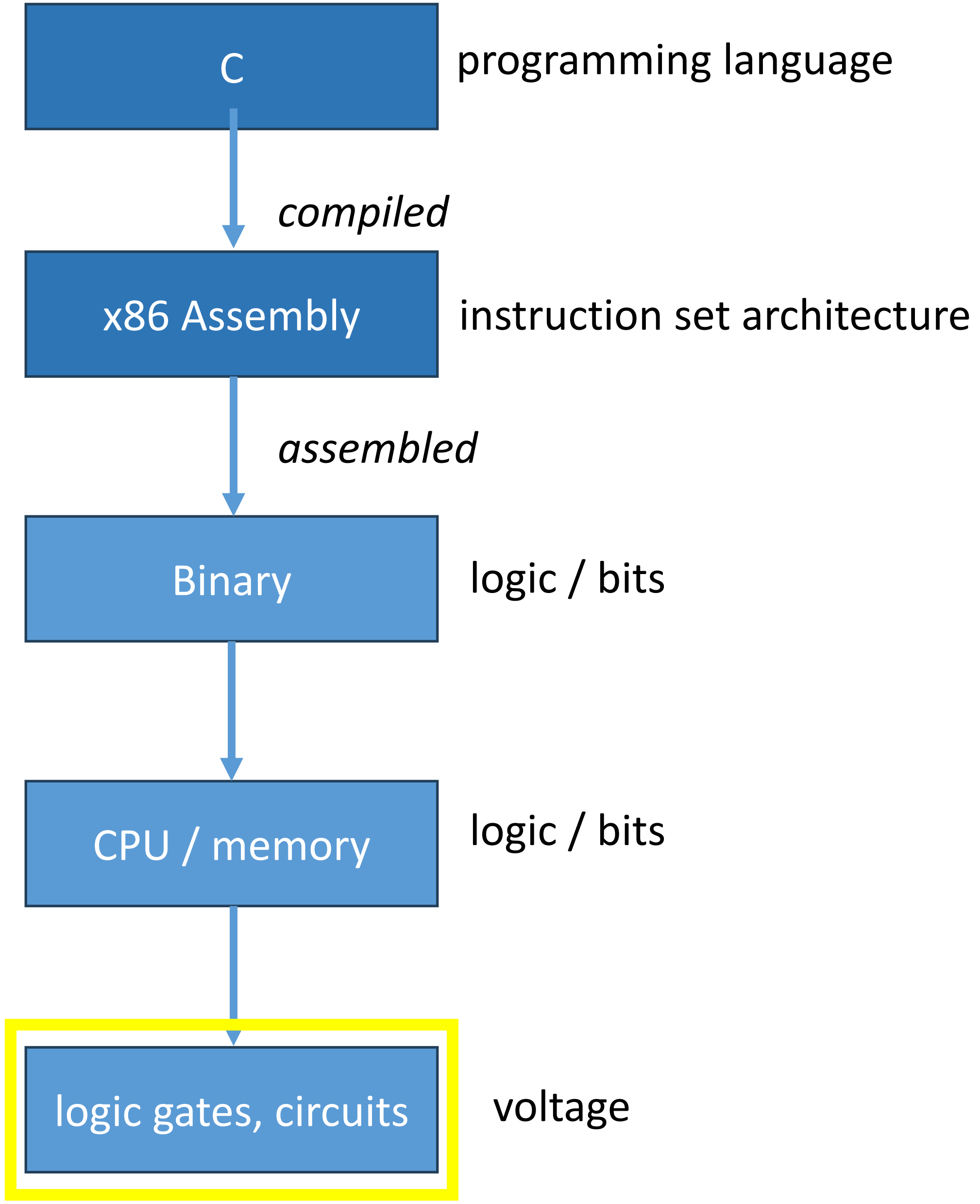
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	”
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	”
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	“
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	“
13	Threading	pthreads Game of Life
14	Threading	“



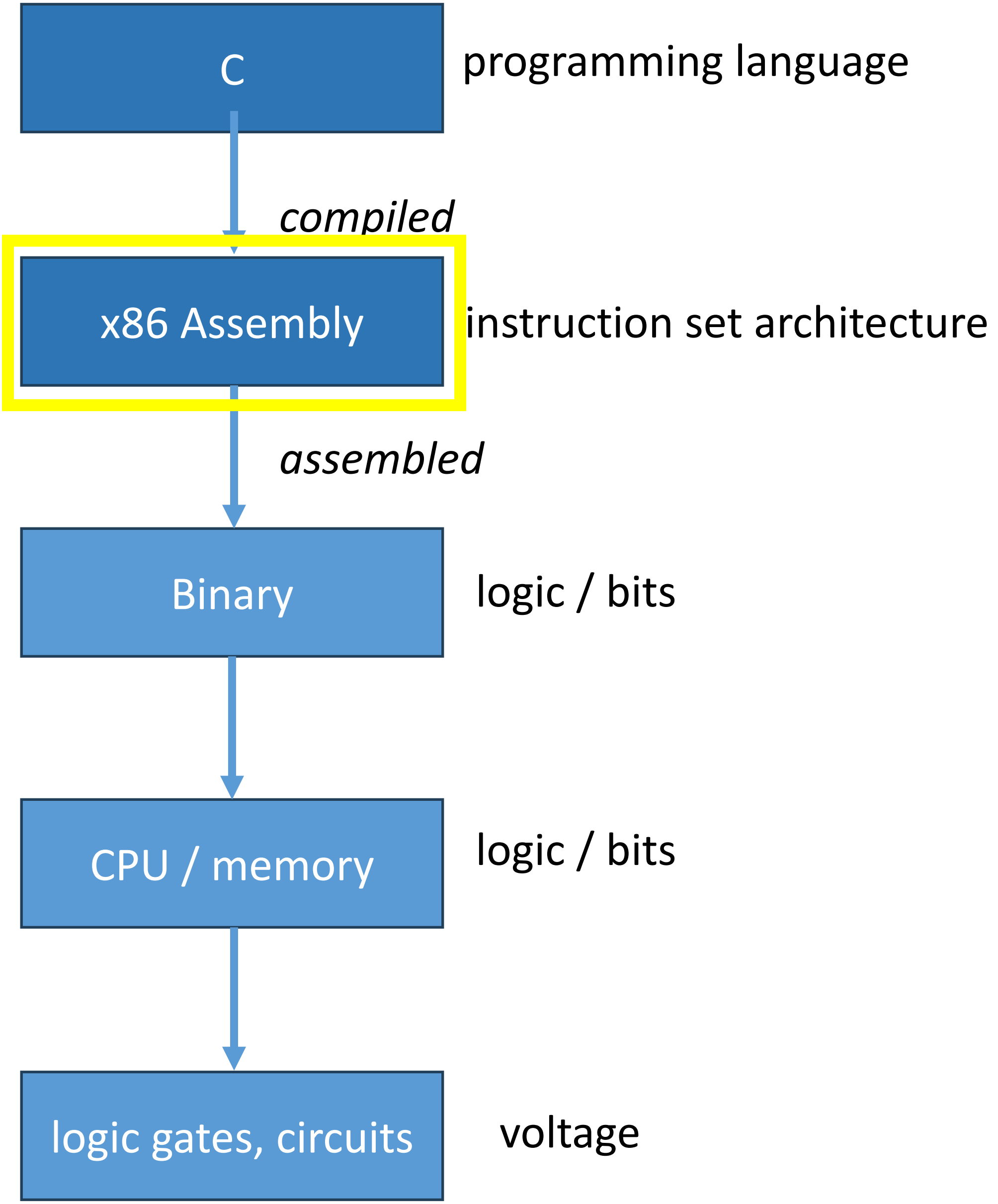
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	"
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	"
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	"
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	"
13	Threading	pthreads Game of Life
14	Threading	"



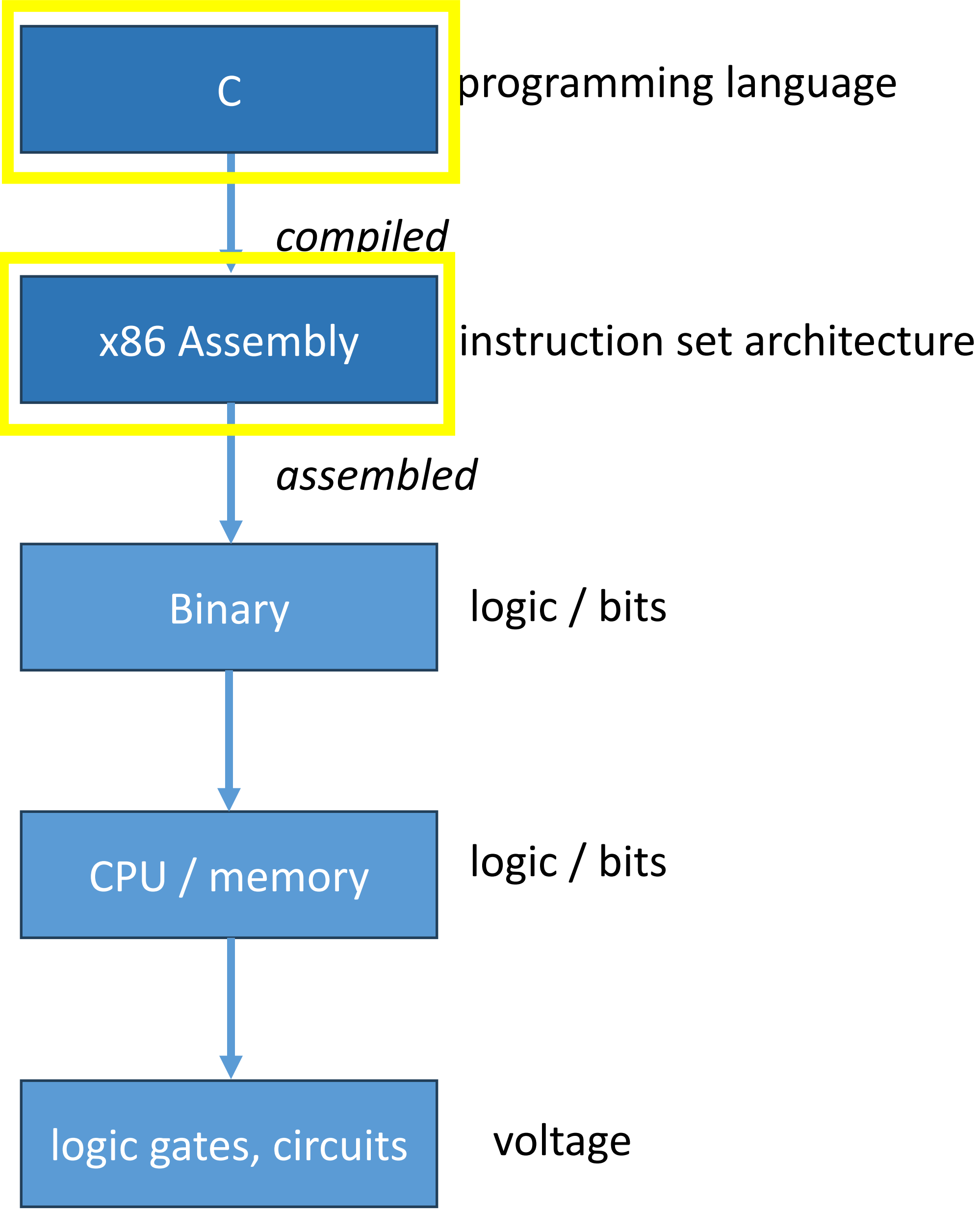
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	”
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	”
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	”
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	”
13	Threading	pthreads Game of Life
14	Threading	”



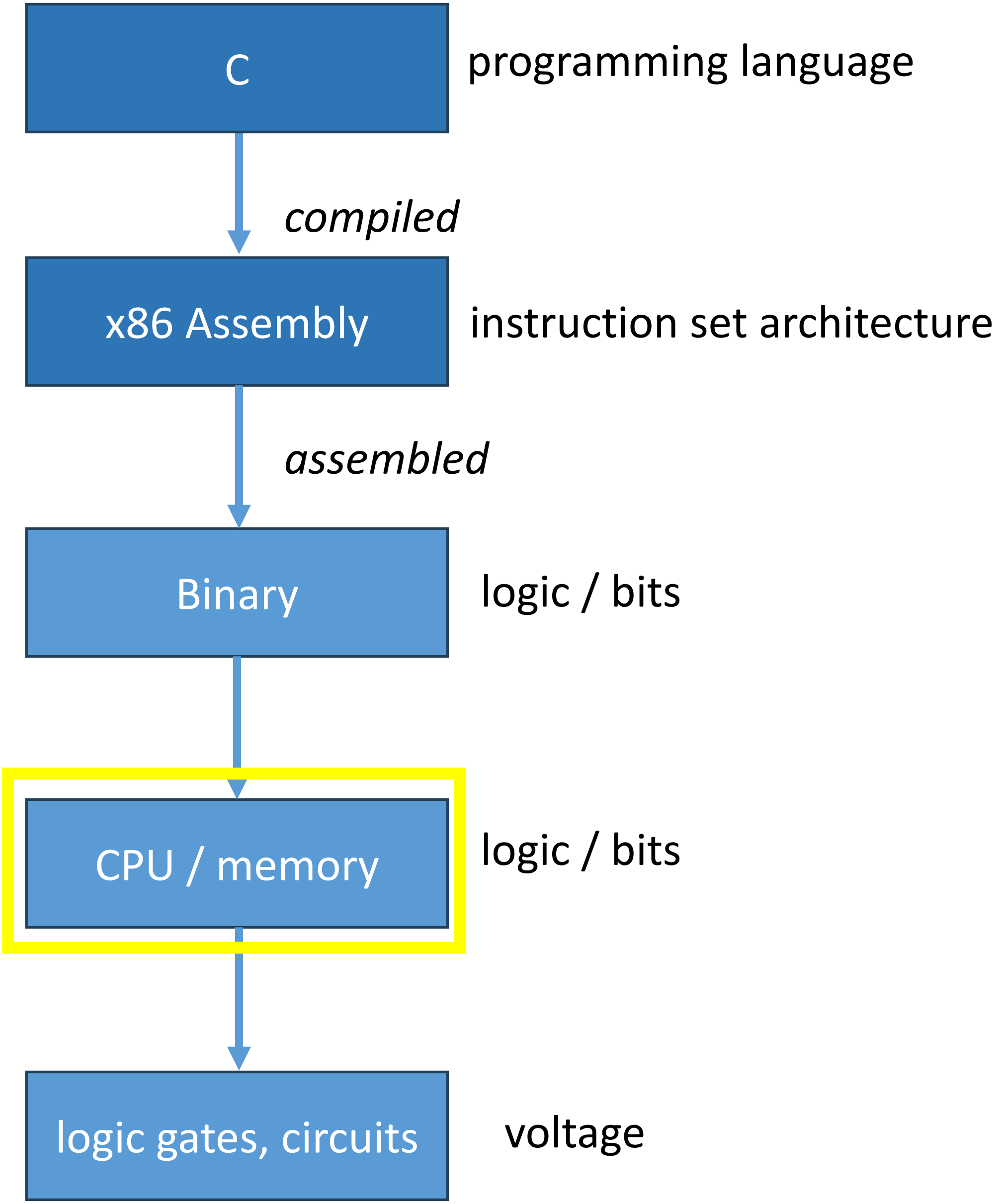
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	"
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	"
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	"
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	"
13	Threading	pthreads Game of Life
14	Threading	"



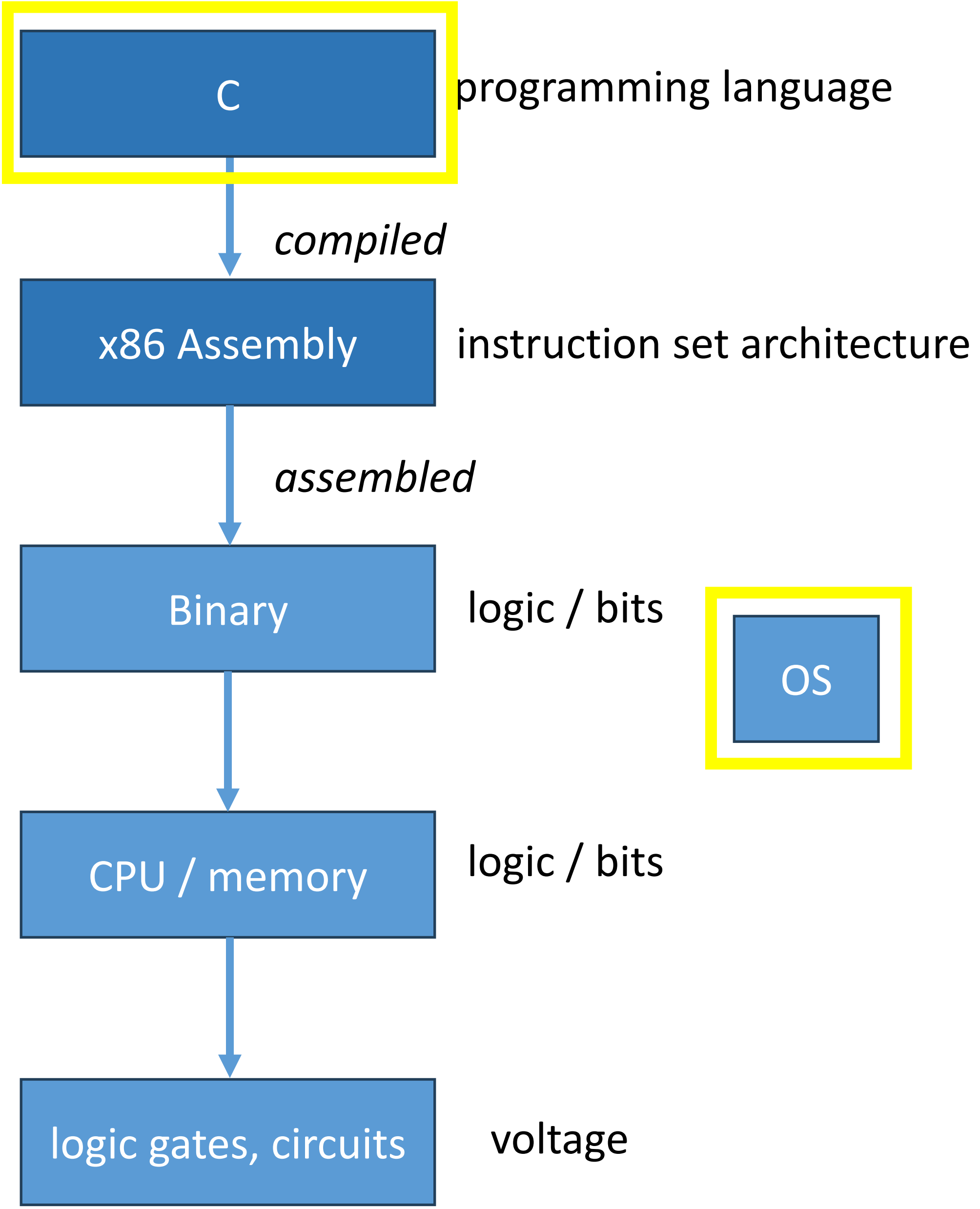
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	"
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	"
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	"
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	"
13	Threading	pthreads Game of Life
14	Threading	"



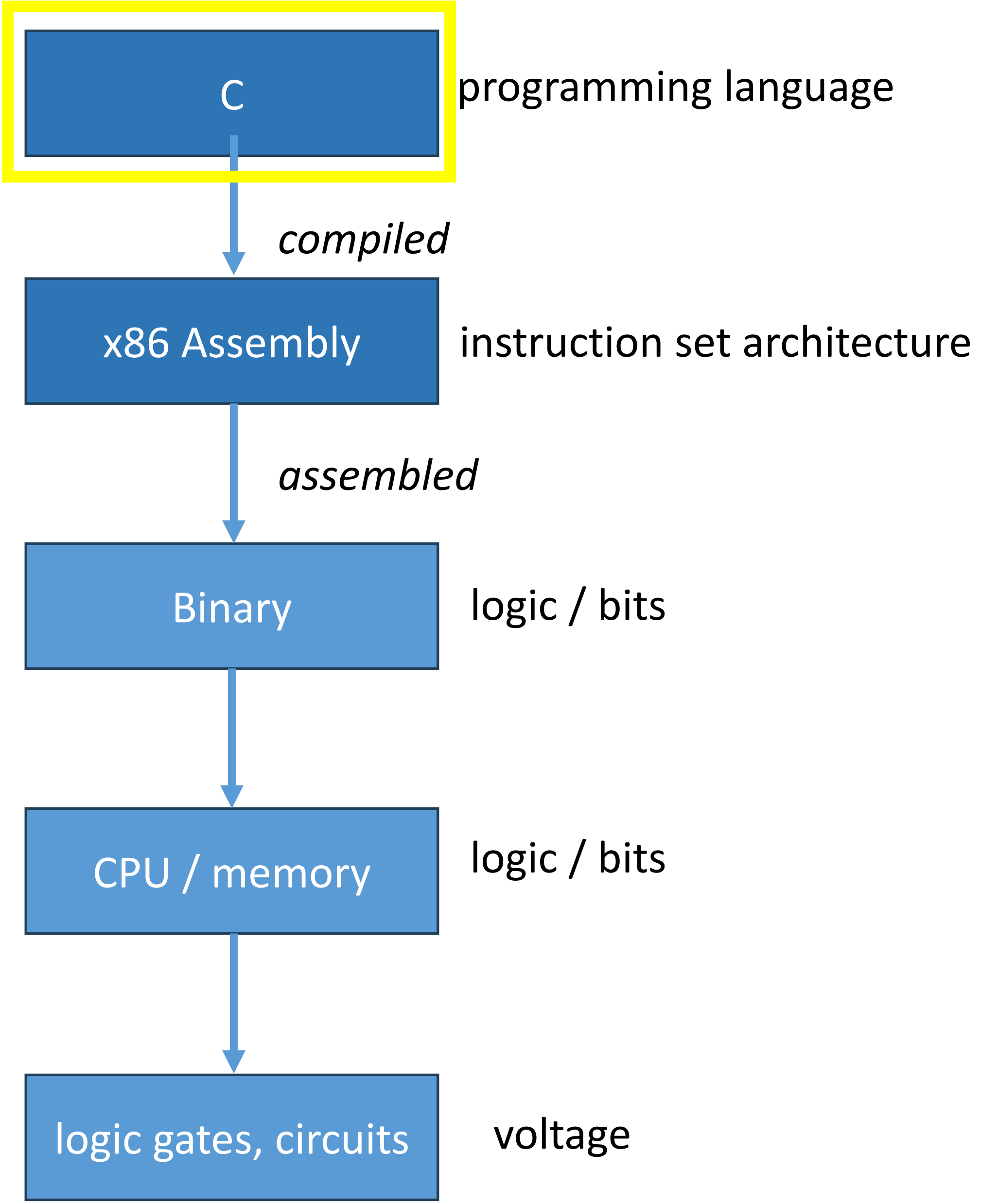
Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	"
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	"
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	"
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	"
13	Threading	pthreads Game of Life
14	Threading	"



Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	”
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	”
	Spring Break	
8	Storage and Memory Hierarchy	Game of Life
9	Caching	“
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	“
13	Threading	pthreads Game of Life
14	Threading	“



CS31: Introduction to Computer Systems

Week 1, Class 2

Introduction to C Programming

01/25/24

Dr. Sukrit Venkatagiri
Swarthmore College

Agenda

- Basics of C programming
  - Comments, variables, print statements, loops, conditionals, etc.
  - NOT the focus of this course
  - Ask questions if you have them!

- Comparison of C vs. Python
  - Data organization and strings
  - Functions

The First "Computers": Women

FUN FACT

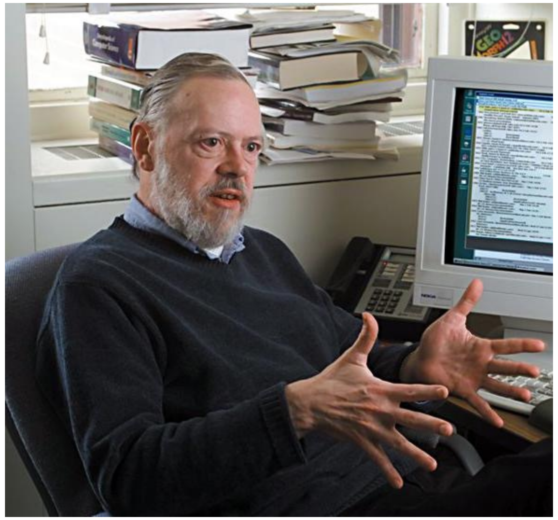
ENIAC was developed 10 mi from here, at UPenn

History

When Computer Coding Was a 'Woman's' Job | HISTORY

Computer programming used to be a 'pink ghetto'—so it was underpaid and undervalued.

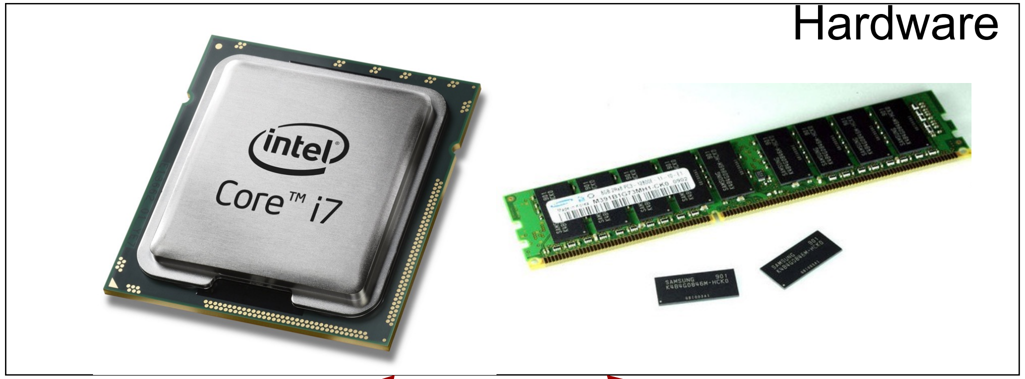
What is C?



Dennis Ritchie
worked at Bell LABORATORIES, INC.

first transistor, solar cell, compilers,
C, C++, Unix, deep learning, + more!

Machine /
Hardware



C → Unix

C was created for systems programming back in 1972.

C was created to write Unix.

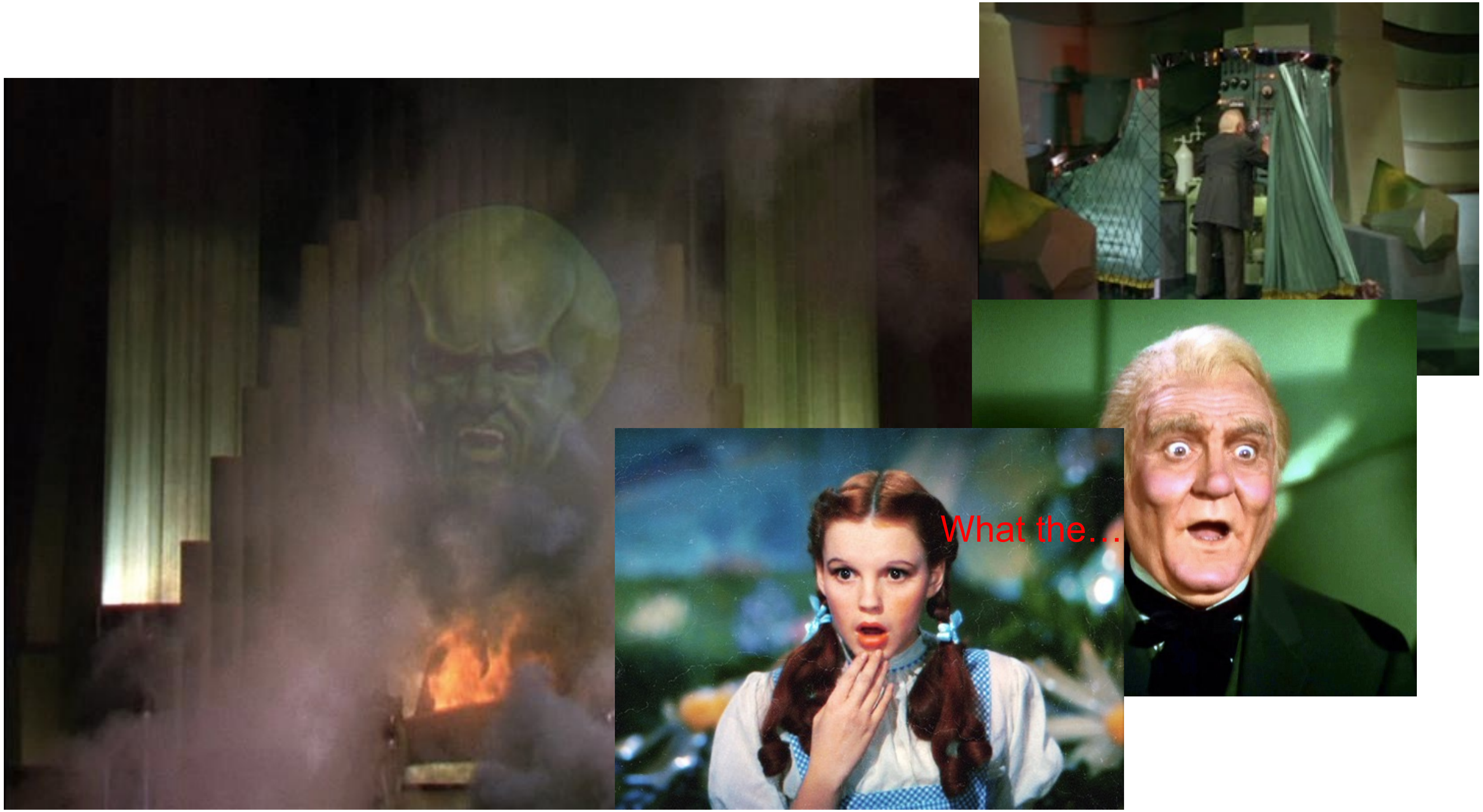
Why C in this course?

Have you watched the Wizard of Oz?

What was going on behind the curtains?

More than what you would think!

The mystery revealed!



Python versus C: Paradigms

Python and C follow different programming paradigms.

- C:
  - is procedure-oriented
  - breaks down to functions
- Python:
  - follows an object-oriented paradigm (as do C++ and Java)
  - allows Python to break down objects and methods

So, the point(er) is….?

- Programming languages are tools
  - Python is one language and it does its job well
  - C is another language and it does its job well

- Pick the right tool for the job
  - C is a good language to explore how the system works under-the-hood.
  - C is the Language of Systems Programmers: Fast running OS code that exposes the details of the hardware is really important!

- It's the right tool for the job we need to accomplish in this course!

Hello World

<u>Python</u>	<u>C</u>

`# hello world`
`import math`
 	
`def main():`
`    print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

Hello World

Python	C

<pre># hello world 	
import math
 	
def main():
    print “hello world”
 	
main()</pre>
	<pre>// hello world
	#include <stdio.h>
	
	int main( ) {
	  printf(“hello world\n”);
	  return 0;
	}</pre>

#: single line comment	//: single line comment

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`    print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	//: single line comment
import libname: include Python libraries	#include<libname>: include C libraries

Hello World

Python	C

`# hello world`
`import math`
 	
`def main():`
`    print "hello world"`
 	
`main()`
	`// hello world`
	`#include <stdio.h>`
	
	`int main( ) {`
	`  printf("hello world\n");`
	`  return 0;`
	`}`

#: single line comment	//: single line comment
import libname: include Python libraries	#include<libname>: include C libraries
Blocks: indentation	Blocks: { } (indent for readability)

To <u>Blank Space</u> or Not to Blank Space

- Python cares about how your program is formatted. Spacing has meaning.

- <u>C compiler does NOT care</u>. Spacing is ignored.
  - This includes spaces, tabs, new lines, etc.
  - Good practice (for your own sanity):
    - Put each statement on a separate line.
    - Keep indentation consistent within blocks.

Hello World

Python

```python
# hello world
import math

def main():
   print "hello world"

main()
```

C

```c
// hello world
#include <stdio.h>

int main( ) {
  printf("hello world\n");
  return 0;
}
```

Python	C
#: single line comment	//: single line comment
import libname: include Python lib.	#include<libname>: include C libraries
Blocks: indentation	Blocks: { } (indent for readability)
print: statement to printout string	printf: function to print out format string
statement: each on separate line	statement: each ends with ;
def main(): : the main function definition	int main( ) : the main function definition (int specifies the return type of main)

Types

- Everything is stored as bits.
- *Type* tells us how to interpret those bits.
- “What type of data is it?”
  - integer, floating point, text, etc.

Type Matters!

- No self-identifying data
  - Looking at a sequence of bits doesn’t tell you what they mean
  - Could be signed, unsigned integer
  - Could be floating-point number
  - Could be part of a string
- The machine interprets what those bits mean!

Types in C

- All variables have an explicit type!

- You (programmer) must declare variable types.
  - Where: at the beginning of a block, before use.
  - How: <variable type> <variable name>;

- Examples:

int humidity;	float temperature;
humidity = 20;	temperature = 32.5

Numerical Type Comparison

Integers (int)

- Example:

```
int humidity;
humidity = 20;
```

- Only represents integers
- Small range, high precision
- Faster arithmetic
- (Maybe) less space required

Floating Point (float, double)

- Example:

```
float temperature;
temperature = 32.5;
```

- Represents fractional values
- Large range, less precision
- Slower arithmetic

I need a variable to store a number, which type should I use? <u>Use the one that fits your specific need best…</u>

Operators: consider the type

- **Arithmetic**: **+, -, *, /, %** (numeric type operands)

  /: operation and result type depends on operand types:

  - Two `int` operands: `int` division truncates the result → 3/2 is 1
  - One or two float or double operands: floating-point division → 3.0/2 is 1.5

  %: mod operator: (only int or unsigned types)

  - Gives you the (integer) remainder of division

    13 % 2 is 1 27 % 3 is 0

Operators: consider the type

- **Shorthand operators:**
  - var **=** var **op** expr; ⟶ var **op=** expr;
    x += 4 is equivalent to x = x + 4

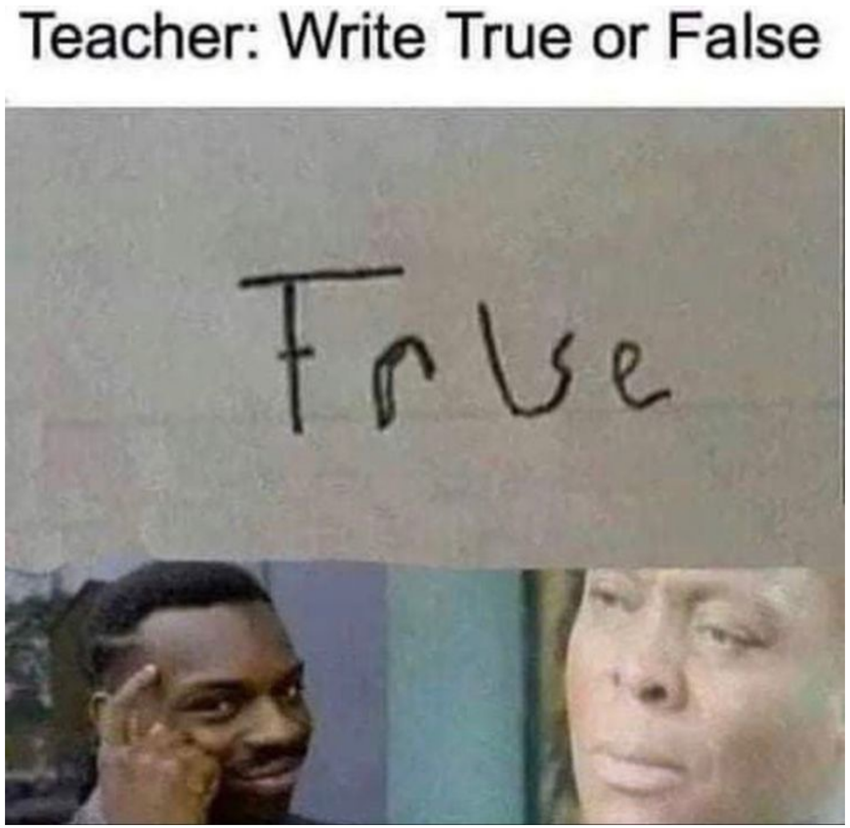
    int y = 4;
    y *= 2; ⟵ **What is the value of y?**

  - var = var+1; ⟶ var**++;**
    var = var+1; ⟶ var**--**;

  - x++ is same as x = x + 1 x-- is same as x = x -1;
  - ++x and –x are **different** from x++ and x– (we'll talk about this later)

Boolean (true/false) values in C

- There is no “boolean” type in C!
- Instead, **integer expressions** used in conditional statements are interpreted as true or false
- ***Norm*: Zero (0) is false, any non-zero value is true**
- Questions?
- “Which non-zero value does it use?”



Operators: consider the type

- **Relational** (operands any type, result integer “boolean”):
  - **<, <=, >, >=, ==, !=**

  - 6 != (4+2) is 0 (false)
  - 6 > 3 some non-zero value (we don’t care which one) (true)

- **Logical** (operands int “boolean”, result integer “boolean”):
  - **!** (not): !6 is 0 (false)
  - **&&** (and): 8 && 0 is 0 (false)
  - **||** (or): 8 || 0 is non-zero (true)

Conditional Statements

Basic if statement:

```
If (<boolean expr>) {
  if-true-body
}
```

With optional else:

```
if (<boolean expr>) {
  if-true-body
} else {
  else body(expr-false)
}
```

Chaining if-else if

```
if (<boolean expr1>) {
  if-expr1-true-body
} else if (<bool expr2>){
  else-if-expr2-true-body
   (expr1 false)
}
...
} else if (<bool exprN>){
  else-if-exprN-true-body
}
```

With optional else:

```
if (<boolean expr1>) {
  if-expr1-true-body
} else if (<bool expr2>){
  else-if-expr2-true-body
}
...
} else if (<bool exprN>){
  else-if-exprN-true-body
} else {
  else body
   (all exprX's false)
}
```

Very similar to Python, just remember { } are blocks

While Loops

- Basically identical to Python while loops:

```
while (<boolean expr>) {
    while-expr-true-body
}
```

```
x = 20;
while (x < 100) {
  y = y + x;
  x += 4;     //  x = x + 4;
}
<next stmt after loop>;
```

```
x = 20;
while (1) {   // while true
  y = y + x;
  x += 4;
  if (x >= 100) {
     break;   // break out of loop
  }
}
<next stmt after loop>;
```

For loops: different than Python’s

```
for (<initialize>; <condition>; <step>) {
   for-loop-body-statements
}
<next stmt after loop>;
```

1. Evaluate <inititialize> one time, when first eval **for** statement
2. Evaluate <condition>, if it is false, drop out of the loop (<next stmt after loop>)
3. Evaluate the statements in the for loop body
4. Evaluate <step>
5. Goto step (2)

```
for (i = 1; i <= 10; i++) {  // example for loop
   printf(“%d\n”, i*i);
}
```

`printf` function

- Similar to Python’s formatted print statement, with a few differences:
  - C: need to explicitly print end-of-line character (`\n`)
  - C: **string and char are different types**
    `‘a’`: in Python is a string, in C is a (single) **char**
    `“a”`: in Python is a string, in C is a **string**

```
Python:   print  “%d %s\t %f”  % (6, “hello”, 3.4)
     C:   printf(“%d %s\t %f\n”,  6, “hello”, 3.4);

          printf(<format string>, <values list>);
```

%d	int placeholder (-13)
%f or %g	float or double placeholder (9.6)
%c	char placeholder (‘a’)
%s	string placeholder (“hello there”)
\t \n	tab character, new line character

Data Collections in C

- Many complex data types out there (CS 35)

- C has a few simple ones built-in:
  - Arrays
  - Structures (`struct`)
  - Strings (arrays of characters)

- Often combined in practice, e.g.:
  - An array of structs
  - A struct containing strings

Arrays

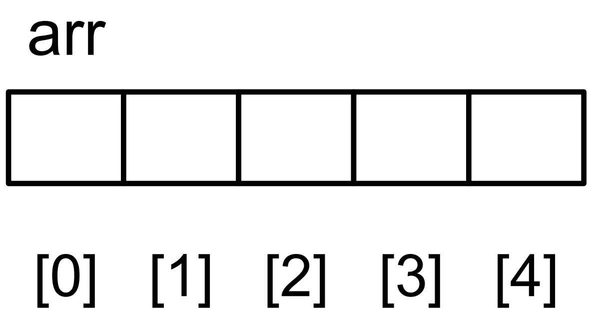
- C's support for collections of values
  - Array buckets store a **single** type of value
  - Specify max capacity (num buckets) when you declare an array variable (single memory chunk)

```
<type> <var_name>[<num buckets>];
 int arr[5];   // an array of 5 integers
float rates[40]; // an array of 40 floats
```

Arrays

- C's support for collections of values
- Often accessed via a loop:

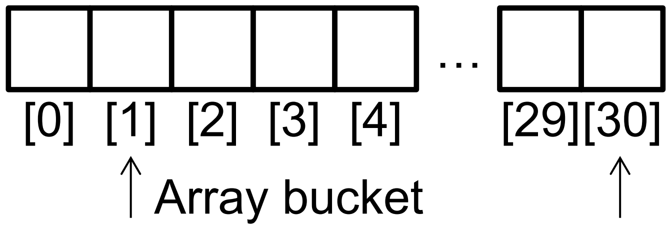
```
int arr[5];  // an array of 5 integers
float rates[40]; // an array of 40 floats
for (i=0; i < 5; i++) {
   arr[i] = i;
   rates[i] = (arr[i]*1.5)/4;
}
```



Get/Set value using brackets [] to index into array.

Array Characteristics

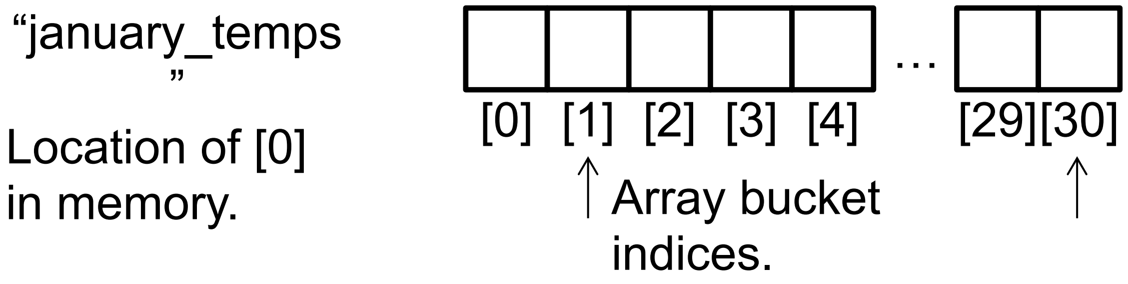
```
int january_temps[31];  // Daily high temps
```



- Indices start at 0! Why?
- Array variable name means, to the compiler, the beginning of the memory chunk. (The memory address)
  - january_temps” (without brackets!) Location of january_temps[0] in memory.
  - Keep this in mind, we’ll return to it soon (functions).

Array Characteristics

```
int january_temps[31];  // Daily high temps
```



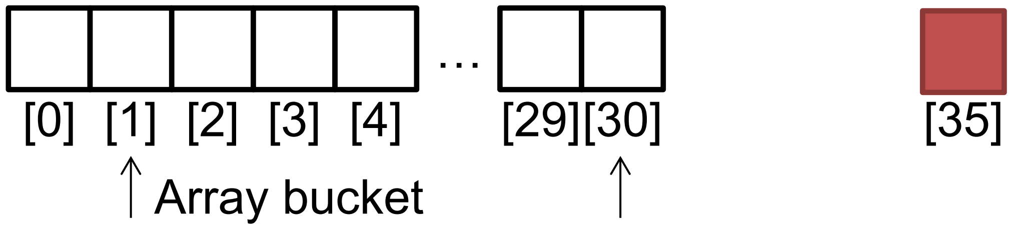
- Indices start at 0! Why?
- The index refers to an offset from the start of the array
  - e.g., [3] means "three integers forward from the starting address"

Array Characteristics

```
int january_temps[31];  // Daily high temps
```

"january_temps"

Location of [0] in memory.



Array bucket indices.

- Asking for january_temps[35]?

<u>C does NOT do bounds checking</u>.

- Python: error
- <u>C: "Sure! I don't care .." <ominous silence while bad things happen></u>

Your TODO List

- **Now:** Submit partner survey
- **Now:** Buy an iClicker
- **Before lab tomorrow:** Complete Lab 0
- **By 11:59pm Thursday:** Lab 1 is due
- **By 11:59pm Friday:** Complete HW1, submit to gradescope
- **The next 13 weeks:** Read the readings before class

Characters and Strings

- A character (type `char`) is numerical value that holds one letter.

  `char my_letter = ‘w’;`   // Note: single quotes

- What is the numerical value?
  – printf(“%d   %c”, my_letter, my_letter);
  – Would print: 119   w

- Why is ‘w’ equal to 119?
  – American Standard Code for Information Interchange (ASCII) standard says so.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Characters and Strings

$ man ascii

119 = w

Characters and Strings

- A character (type `char`) is numerical value that holds one letter.
- A string is a memory block containing characters, one after another…

Hmm, suppose we used printf and %s to print name.

- Examples:

  char food[6] = “Pizza”;

How does it know where the string ends and other memory begins?

name	P	i	z	z	a	(Other memory)
	[0]	[1]	[2]	[3]	[4]	

0 is the “Null character”

Special stuff over here in the lower values.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

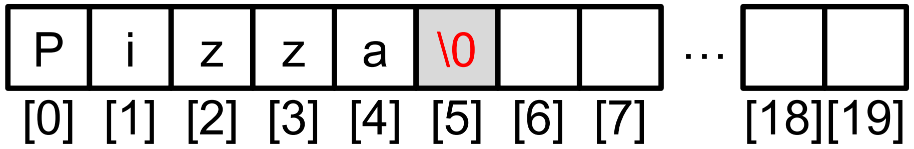
Characters and Strings

$ man ascii

Characters and Strings

- A character (type `char`) is numerical value that holds one letter.
- A string is a memory block containing characters, one after another, with a null terminator (numerical 0) at the end.
- Examples:

```
char name[20] = “Pizza”;
```



Strings in C

- C String library functions: `#include <string.h>`
  - Common functions (strlen, strcpy, etc.) make strings easier
  - Less friendly than Python strings

- More on strings later, in labs.

- For now, remember about strings:
  - Allocate enough space for null terminator!
  - If you're modifying a character array (string), don't forget to set the null terminator!
  - If you see crazy, unpredictable behavior with strings, check these two things!

structs

- Treat a collection of values as a single type:
  - C is not an object oriented language, no classes
  - A `struct` is like just the data part of a class

- Rules:
  1. Define a new struct type outside of any function
  2. Declare variables of the new struct type
  3. Use dot notation to access the field values of a struct variable

Struct Example

Suppose we want to represent <u>a *student* type.</u>

```
struct student {
    char name[20];
    int grad_year;
    float gpa;
};
// Variable bob is of type struct student
struct student bob;
// Set name (string) with strcpy()
strcpy(bob.name, "Robert Paulson");
bob.grad_year = 2019;
bob.gpa = 3.1;

printf("Name: %s, year: %d, GPA: %f", bob.name, bob.grad_year, bob.gpa);
```

Arrays of Structs

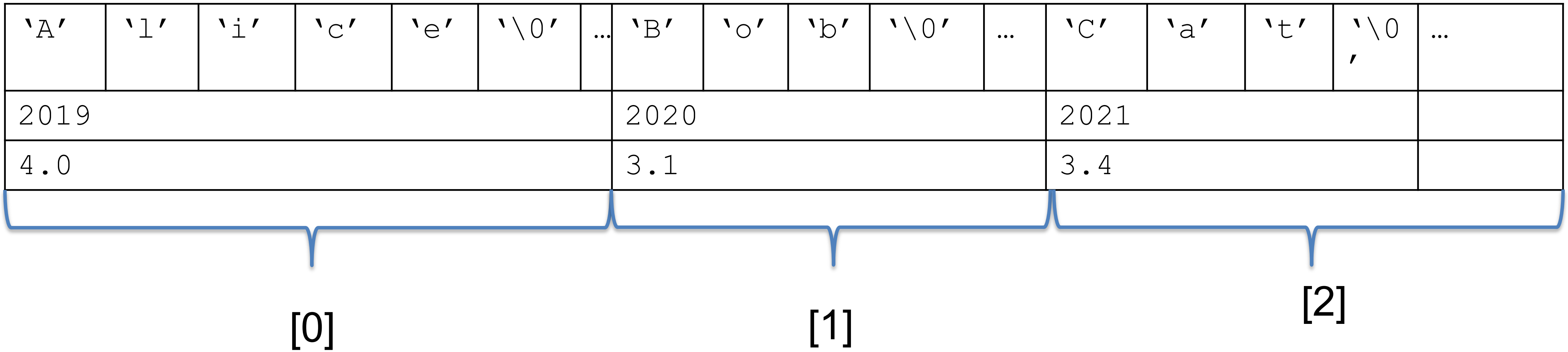
```
struct student {
   char name[20];
   int grad_year;
   float gpa;
};
//create an array of struct students!
struct student classroom[50];

strcpy(classroom[0].name, "Alice");
classroom[0].grad_year = 2014
classroom[0].gpa = 4.0;

// With a loop, create an army of Alice clones!
int i;
for (i = 0; i < 50; i++) {
   strcpy(classroom[i].name, "Alice");
   classroom[i].grad_year = 2014;
   classroom[i].gpa = 4.0;
}
```

Arrays of Structs

```
struct student classroom[50];

strcpy(classroom[0].name, “Alice”);
classroom[0].grad_year = 2019;
classroom[0].gpa = 4.0;

strcpy(classroom[1].name, “Bob”);
classroom[1].grad_year = 2020;
classroom[1].gpa = 3.1

strcpy(classroom[2].name, “Cat”);
classroom[2].grad_year = 2021;
classroom[2].gpa = 3.4
```

Struct: Layout in Memory

`classroom:`



Functions: Specifying Types

Need to specify the return type of the function, and the type of each parameter:

```
<return type> <func name> ( <param list> ) {
    // declare local variables first
    // then function statements
    return <expression>;
}
```

```
// my_function takes 2 int values and returns an int
int my_function(int x, int y) {
  int result;
  result = x;
  if(y > x) {
    result = y+5;
  }
  return result*2;
}
```

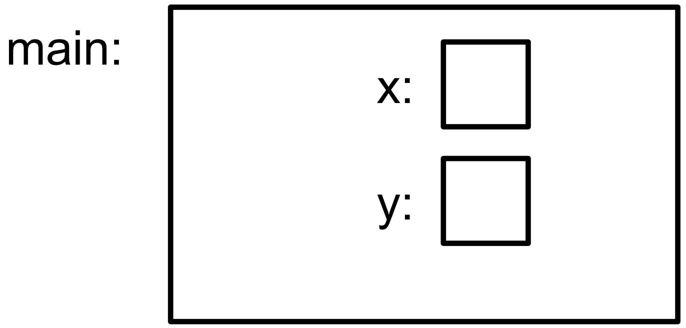
Compiler will yell at you if you try to pass the wrong type!

Function Arguments

Arguments are **passed by value**

- The function gets a separate <u>copy</u> of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf(“%d, %d”, x, y);
}
```



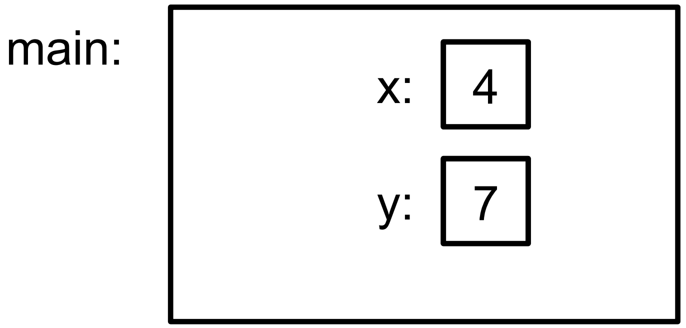
Stack

Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf(“%d, %d”, x, y);
}
```



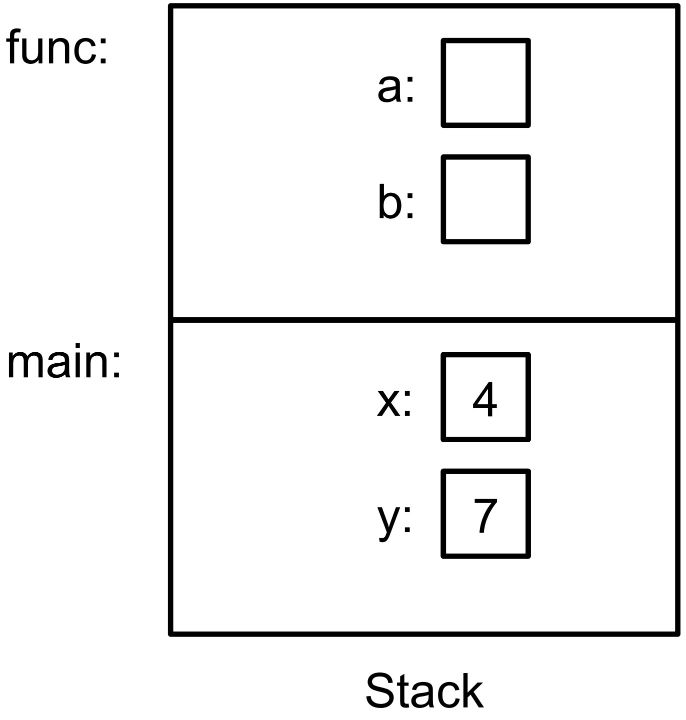
Stack

Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

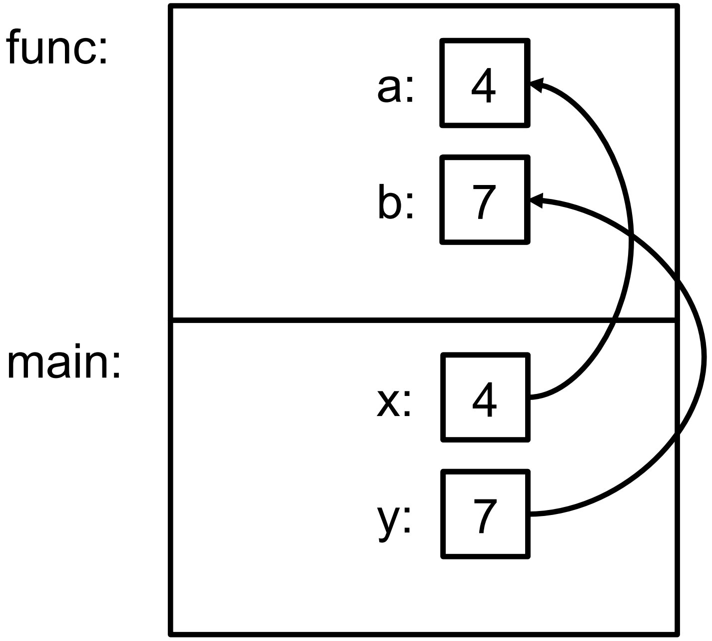


Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



Stack

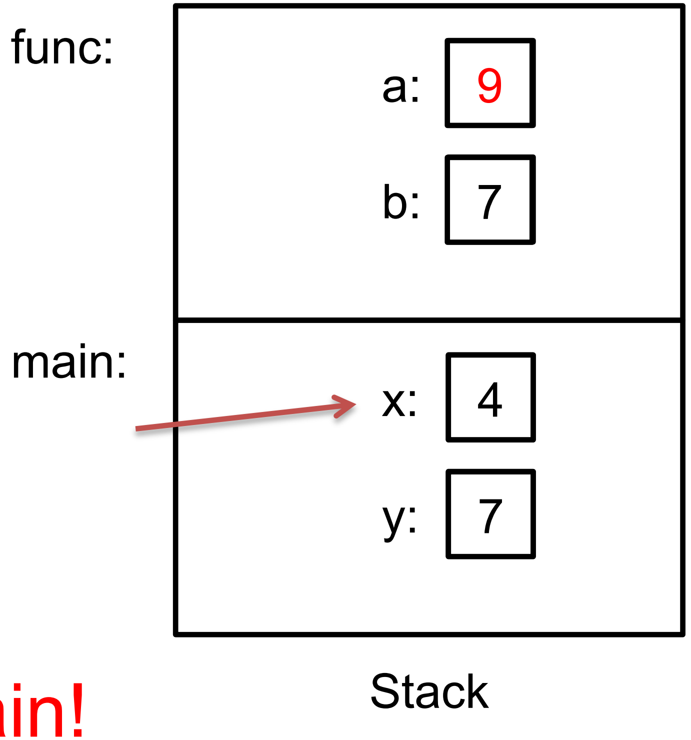
Function Arguments

Arguments are **passed by value**

- The function gets a separate <u>copy</u> of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

Note: This doesn't change!



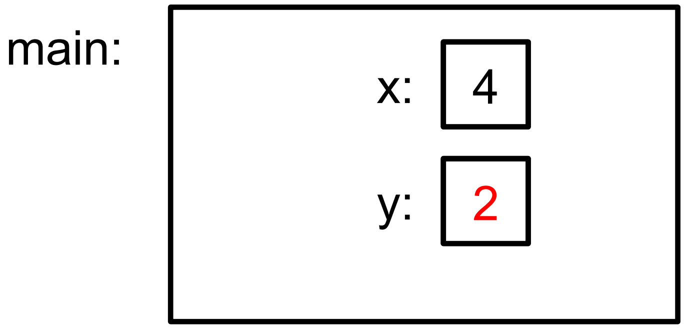
<u>No impact on values in main!</u>

Function Arguments

Arguments are **passed by value**

- The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



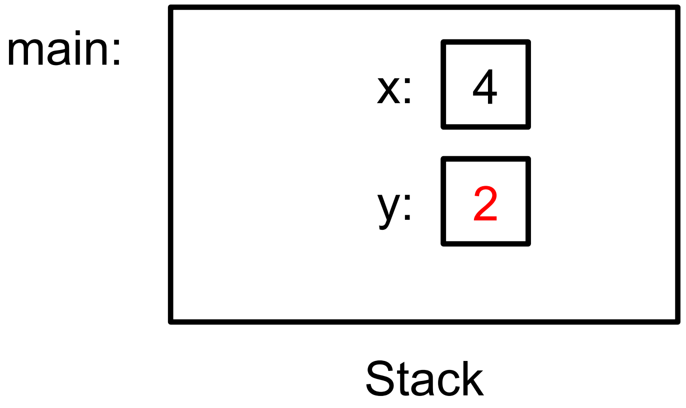
Stack

Function Arguments

Arguments are **passed by value**

- The function gets a separate <u>copy</u> of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main() {
    // declare two integers
    int x, y;
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```



<u>Output: 4, 2</u>

Fear not!

- Don’t worry, I don’t expect you to have mastered C
- It’s a skill you’ll pick up as you go
- We’ll revisit these topics when necessary

- When in doubt: solve the problem in logically, use a whiteboard, whatever else!
  - Translate to C later
  - Eventually, you’ll start to “think in C”

Up next…

- Bits, Bytes, Binary (data representation)