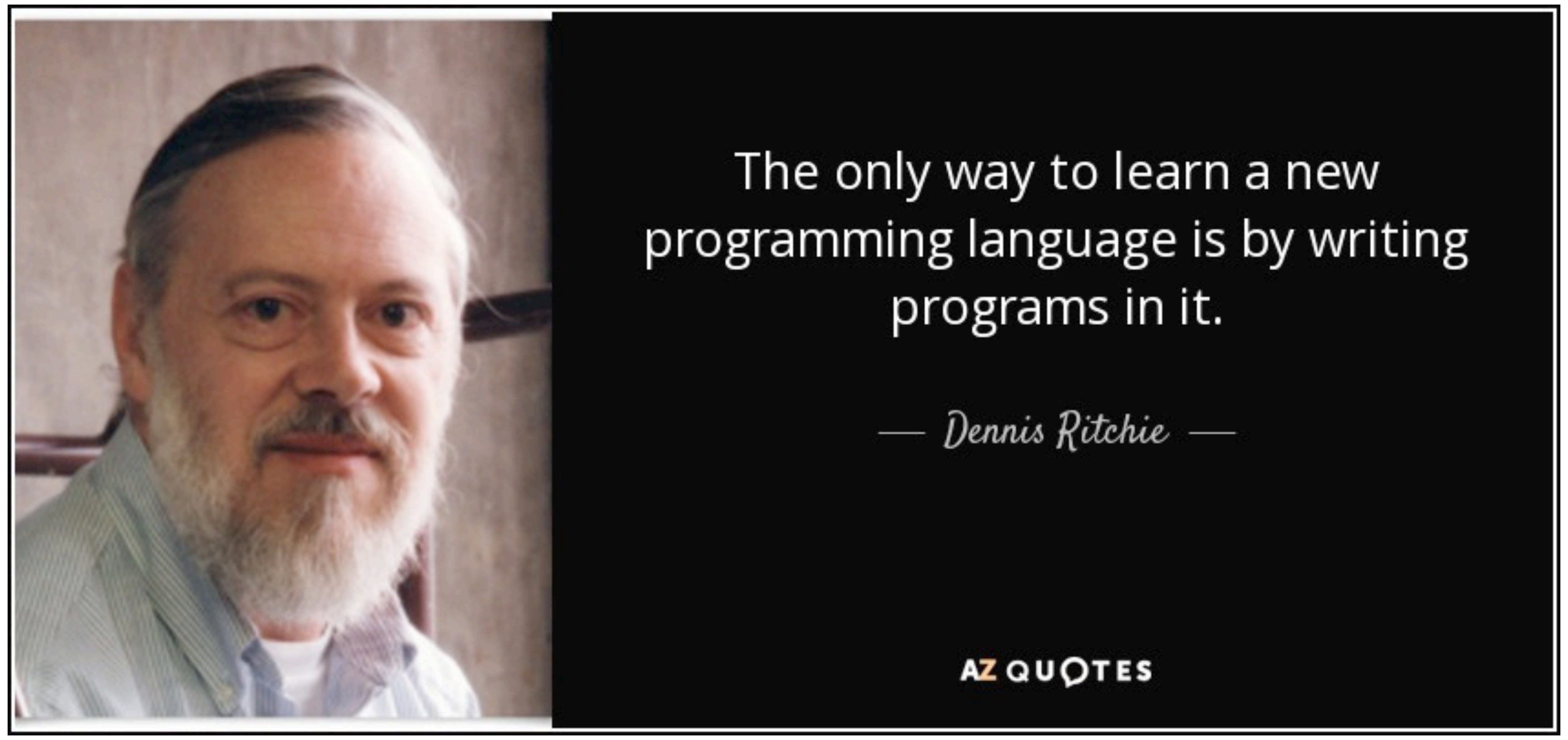
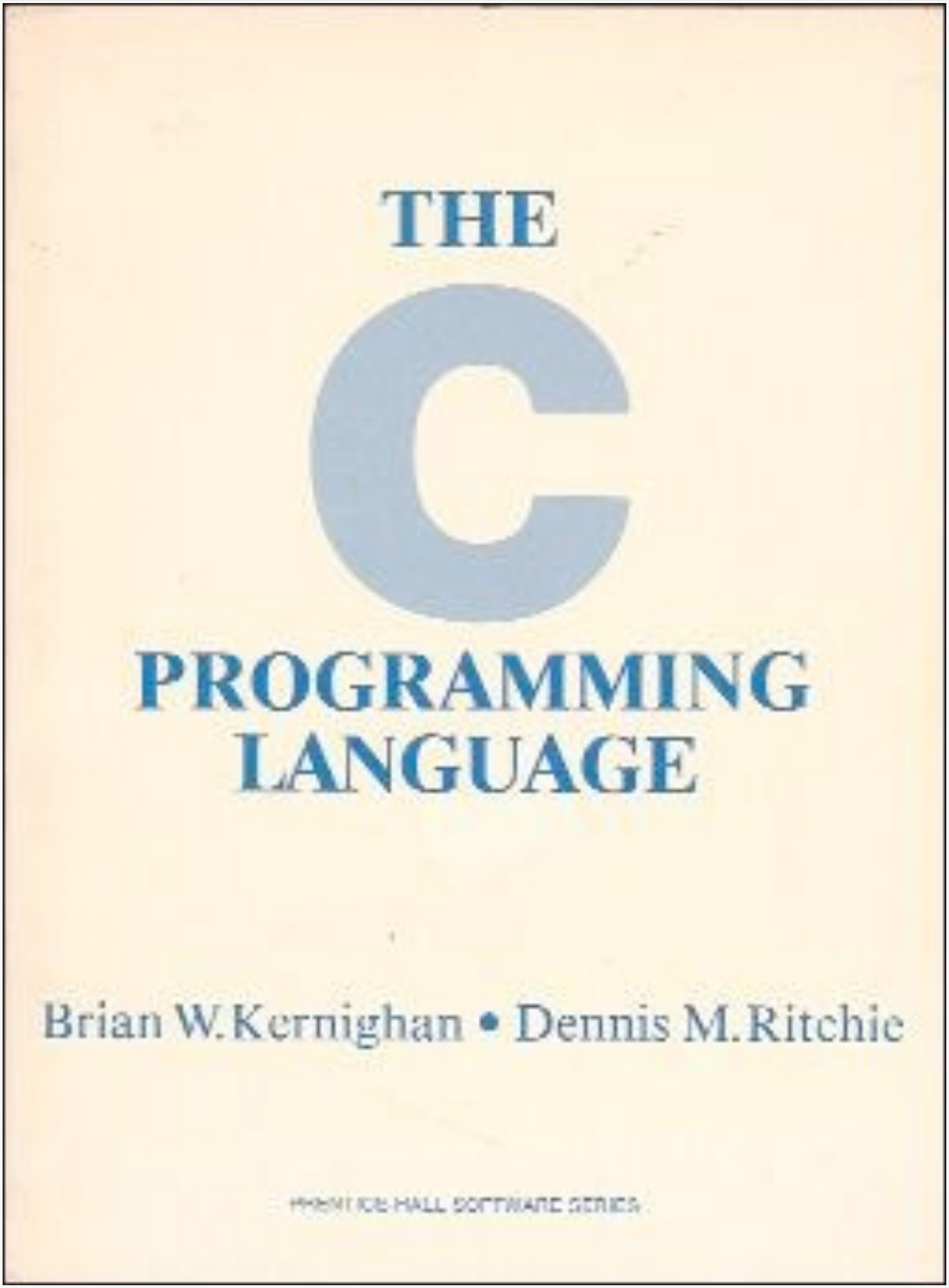
Introduction to C



The only way to learn a new programming language is by writing programs in it.

— Dennis Ritchie —



THE

C

PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE-HALL SOFTWARE SERIES

1st Edition: 1978

Administrivia:

- Reminder, check that you are signed up on Gradescope, Piazza & GitHub
  - GitHub allows us to see & check the progress on projects
    - Defaults also keep you from screwing up and making your archives for the class public
- Lab 0 released, due Monday the 24th
  - Gets you set up with the infrastructure necessary for the class
- Project 1 to be released shortly
  - Trying a new project this semester: **`snek`**

Administrivia 2: Notes on Project 1...

- Project 1 is remarkably ***subtle***
  - Designed to cover ***alot*** of C semantics
- Items include
  - Pointers and structures
  - Strings
  - File I/O
  - Memory allocation and deallocation, including dynamically growing arrays
  - Pointers to functions
- But you don't actually need to write a lot of code! A little more than 100 lines of code

Signed and Unsigned Integers

- C, C++, and Java have *signed integers*, e.g., 7, -255:

```
int x, y, z;
```

- C, C++ also have *unsigned integers*, which are usually used for memory addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent
  0 to 2^{32}-1 (4,294,967,295)

Unsigned Integers: 32-bit example

0000 0000 0000 0000 0000 0000 0000 0000_2 = 0_{10}

0000 0000 0000 0000 0000 0000 0000 0001_2 = 1_{10}

0000 0000 0000 0000 0000 0000 0000 0010_2 = 2_{10}

... ...

0111 1111 1111 1111 1111 1111 1111 1101_2 = $2{,}147{,}483{,}645_{10}$

0111 1111 1111 1111 1111 1111 1111 1110_2 = $2{,}147{,}483{,}646_{10}$

0111 1111 1111 1111 1111 1111 1111 1111_2 = $2{,}147{,}483{,}647_{10}$

1000 0000 0000 0000 0000 0000 0000 0000_2 = $2{,}147{,}483{,}648_{10}$

1000 0000 0000 0000 0000 0000 0000 0001_2 = $2{,}147{,}483{,}649_{10}$

1000 0000 0000 0000 0000 0000 0000 0010_2 = $2{,}147{,}483{,}650_{10}$

... ...

1111 1111 1111 1111 1111 1111 1111 1101_2 = $4{,}294{,}967{,}293_{10}$

1111 1111 1111 1111 1111 1111 1111 1110_2 = $4{,}294{,}967{,}294_{10}$

1111 1111 1111 1111 1111 1111 1111 1111_2 = $4{,}294{,}967{,}295_{10}$

In digital systems everything *stored, communicated, and manipulated* is done using bits...

- A bit can represent one of two possible things: 0 or 1
- But what those things ***are*** is up to how you want to interpret them: the default is just the number 0 or the number 1
  - But it can also be "False" or "True", or depending on context say, "green" or "purple"
- Likewise, a collection of ***N*** bits can represent one of $\boldsymbol{2^N}$ possible things
  - So far we have talked about representing integers, but could represent pixel values, sound samples, floating-point numbers, …

How collections of bits are treated is dependent on the context (*PL types help define the context*)

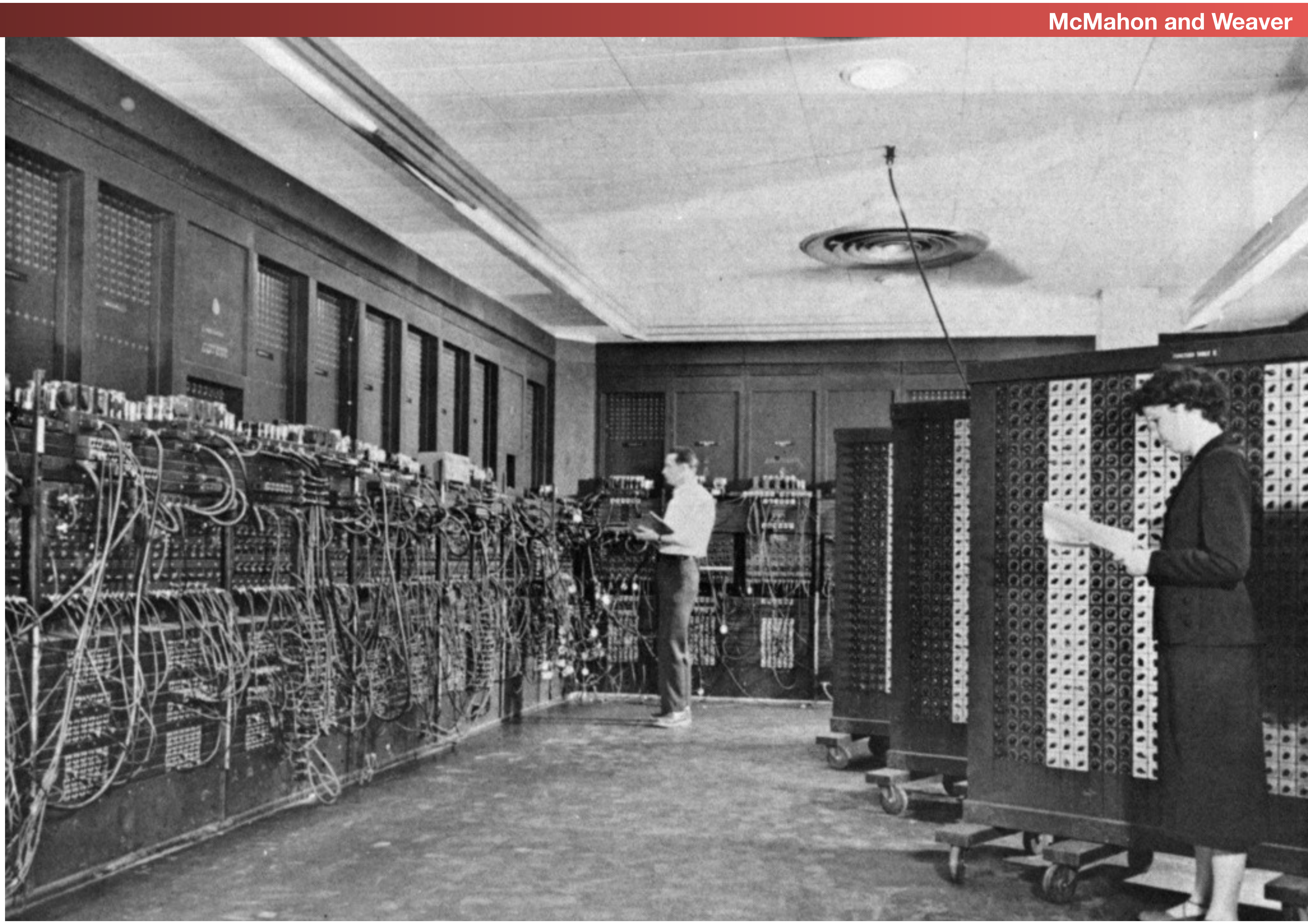
- Say we have a collection of 32 bits...
- We can treat it as a single unsigned number
  - 0 to 2^{32}-1
- Or a single signed number in two's complement
  - -2^{31} to 2^{31}-1
- Or even as a 16 bit unsigned number, followed by an 10 bit signed number, followed by 6 true/false bits
  - So a number from 0 to 2^{16}-1, followed by a number from -2^9 to 2^9-1, followed by 6 true/false bits
  - In the end, taken together, its still representing a single instance out of 2^{32} distinct things

Agenda

- Computer Organization
- Compile vs. Interpret
- C vs Java
- Arrays and Pointers (perhaps)

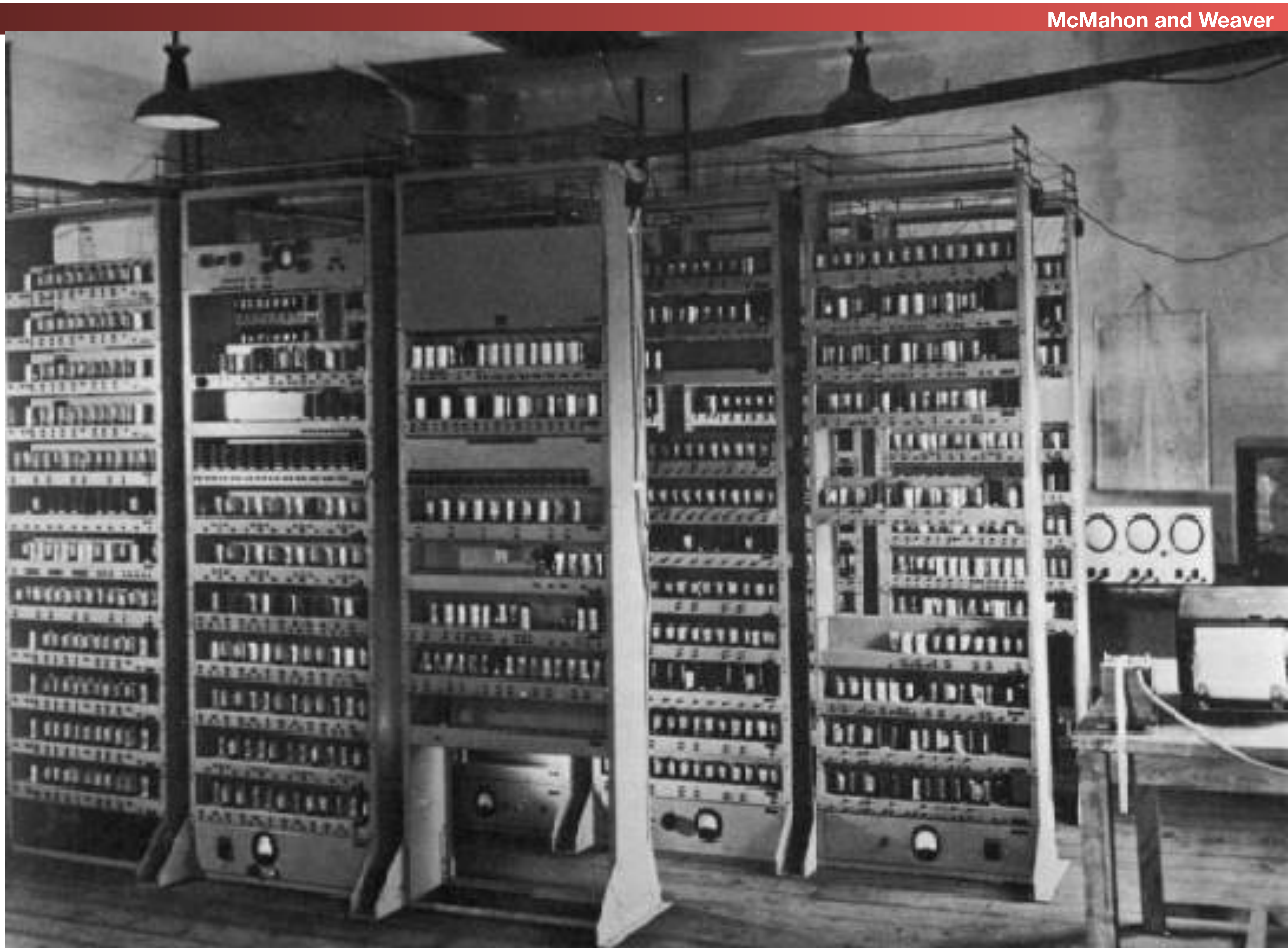
ENIAC (U.Penn., 1946)
First Electronic General-Purpose Computer

- Blazingly fast (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches
  - At that time & before, "computer" mostly referred to ***people*** who did calculations

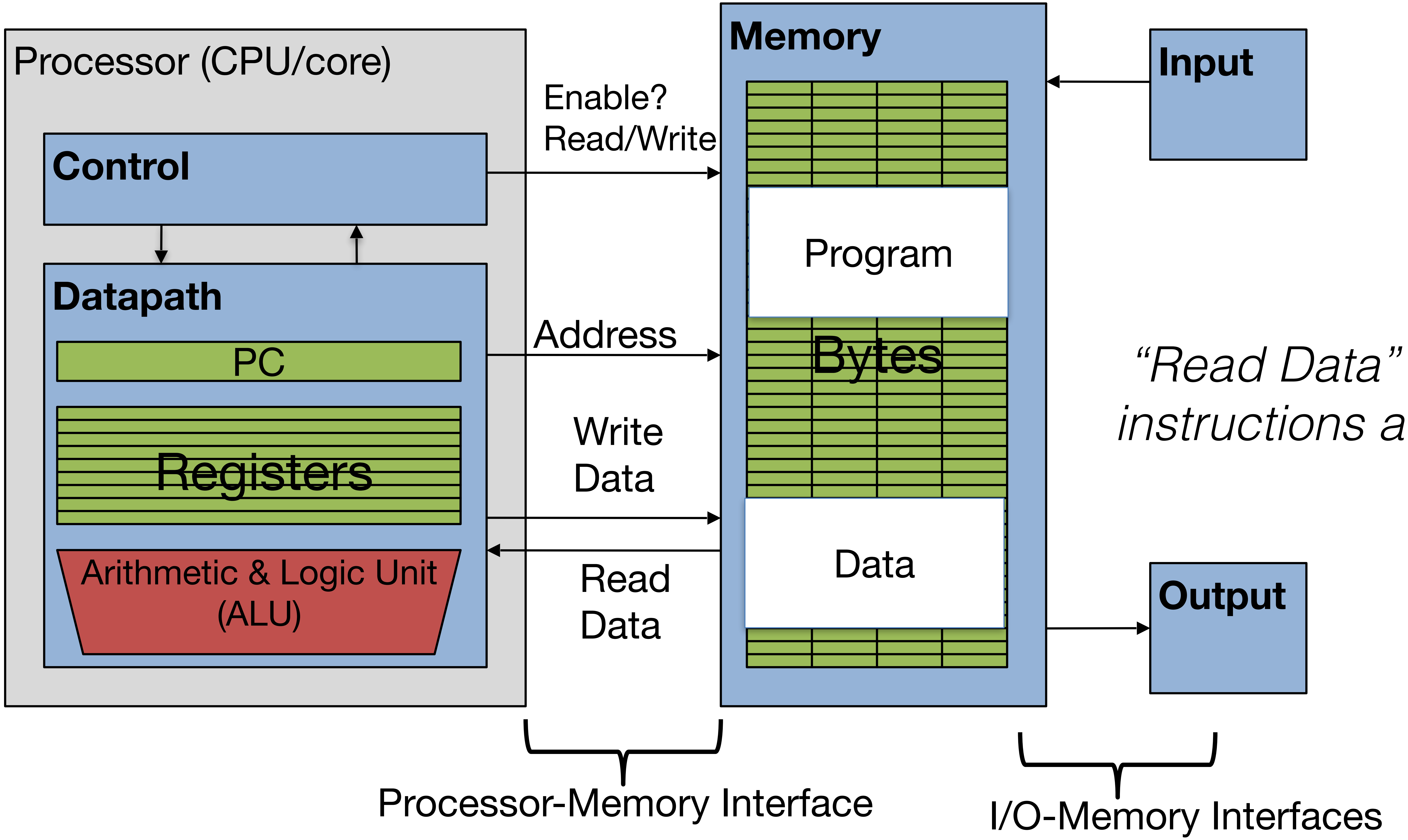


EDSAC (Cambridge, 1949)
First General ***Stored-Program*** Computer

- 35-bit binary 2’s complement words
- Programs held as numbers in memory
  - *This is the revolution:* It isn't just programmable, but the program is just the same type of data that the computer computes on: Bits are not just the numbers being manipulated, ***but the instructions on how to manipulate the numbers***!



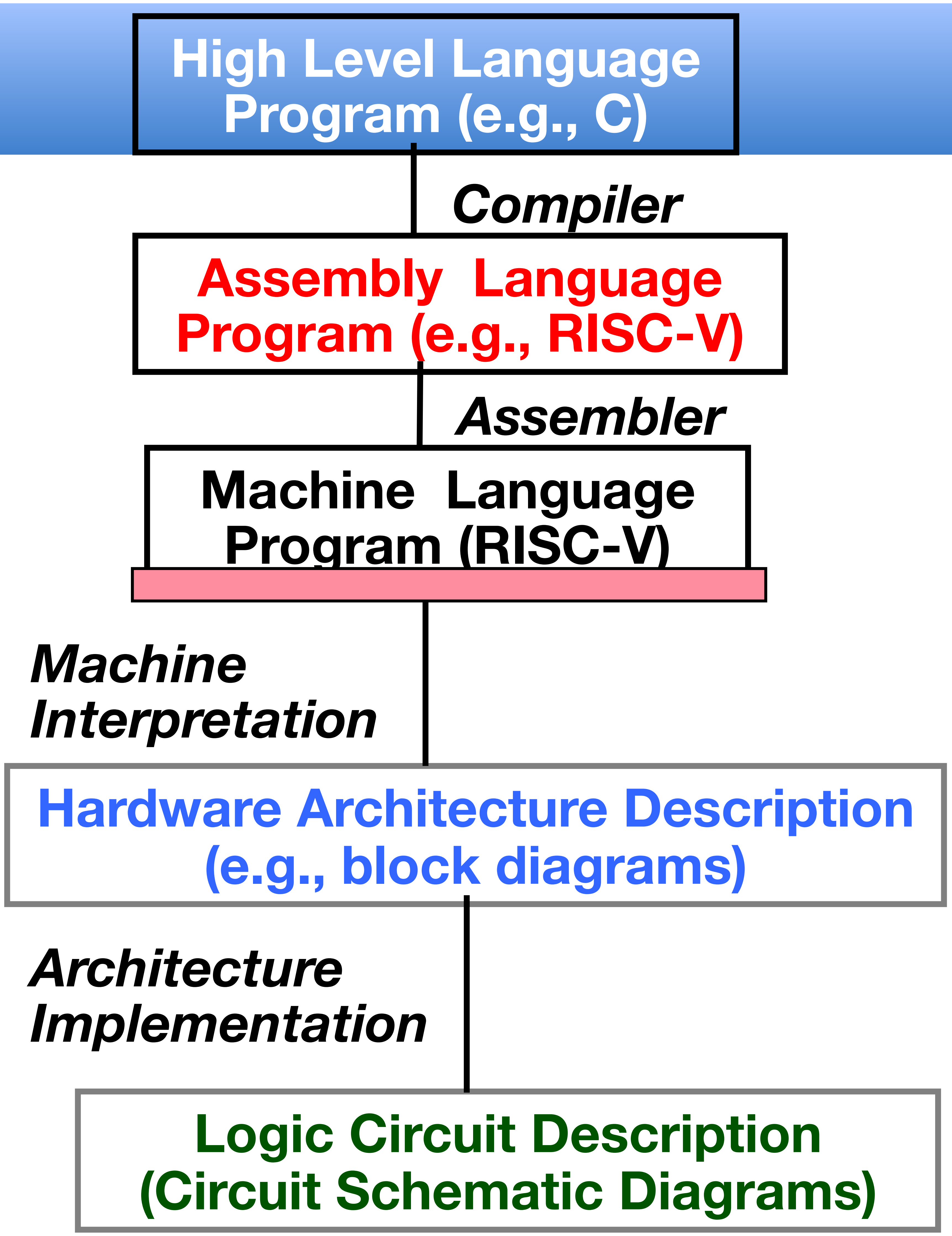
Components of a Computer



"Read Data" is both instructions and data

Great Idea: Levels of Representation/Interpretation

```
lw    t0, t2, 0
lw    t1, t2, 4
sw    t1, t2, 0
sw    t0, t2, 4
```



High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

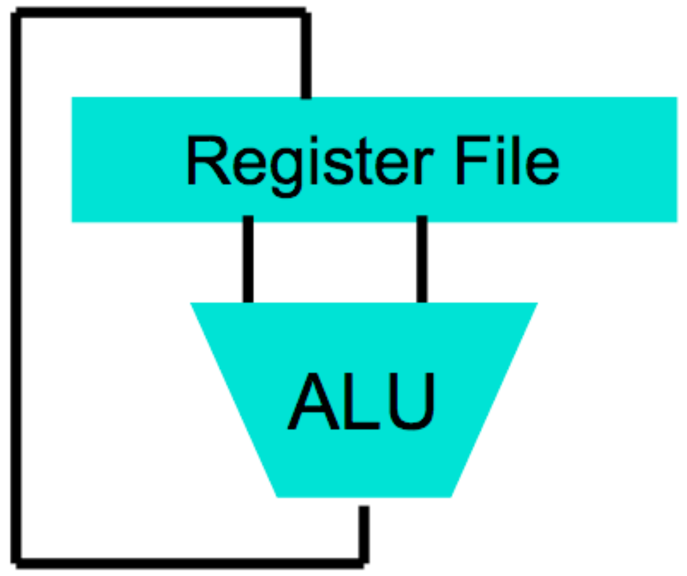
Logic Circuit Description (Circuit Schematic Diagrams)

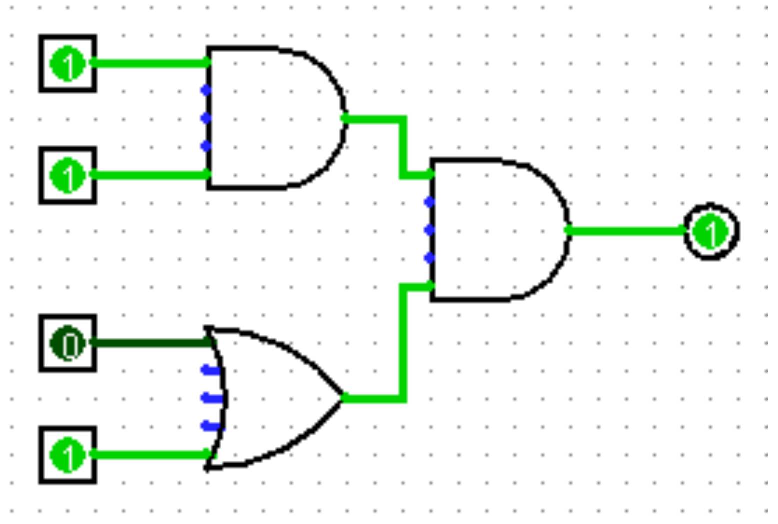
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here!

Anything can be represented as a *number*, i.e., data or instructions

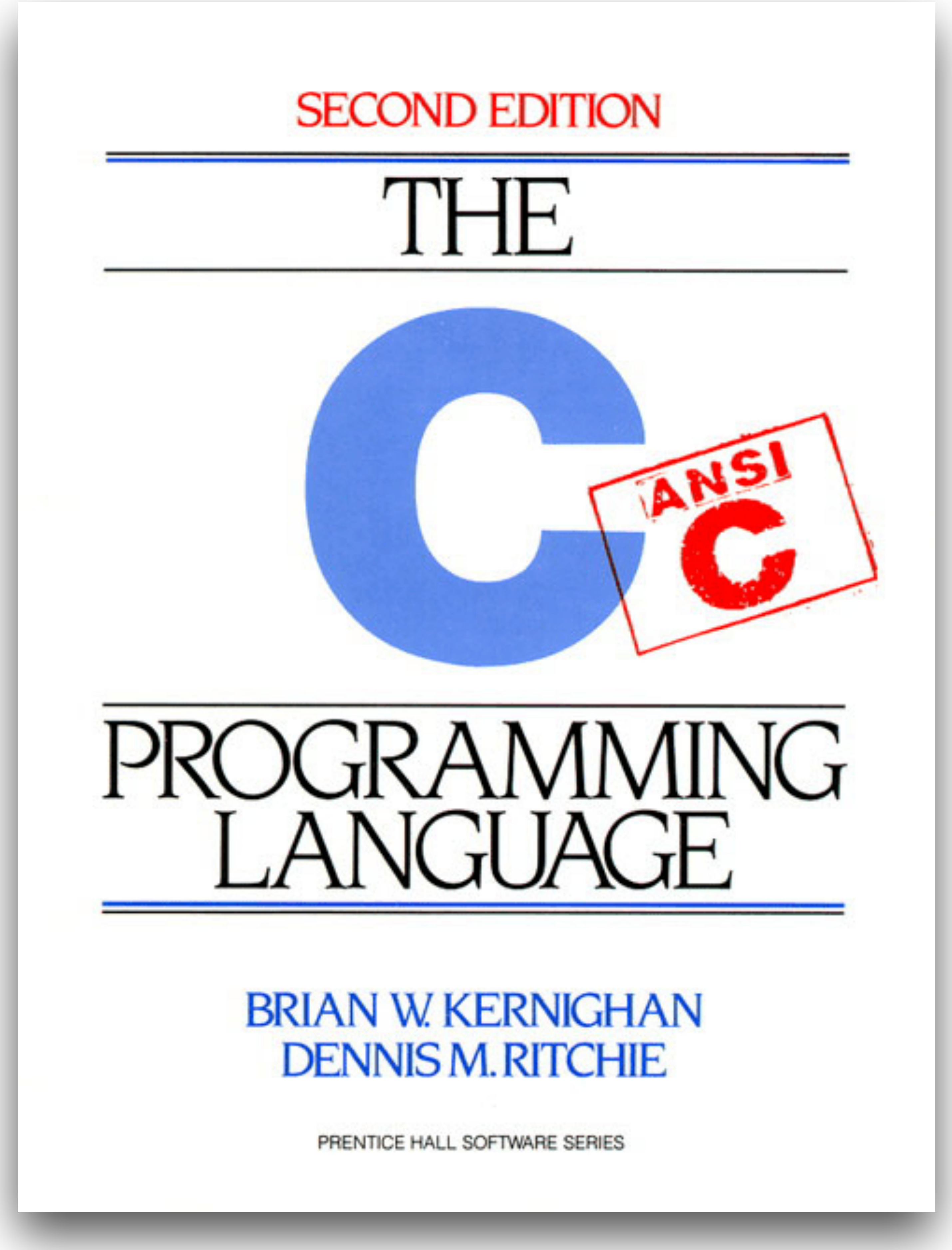
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Introduction to C
“The Universal Assembly Language”

- Class pre-req included classes teaching Java
  - “Some” experience is required before CS61C
    - C++ or Java OK
- Python used in two labs
- C used for everything else "high" level
- Almost all low level assembly is RISC-V
  - But Project 4 may require touching some x86 intrinsics…



“K&R”

Intro to C

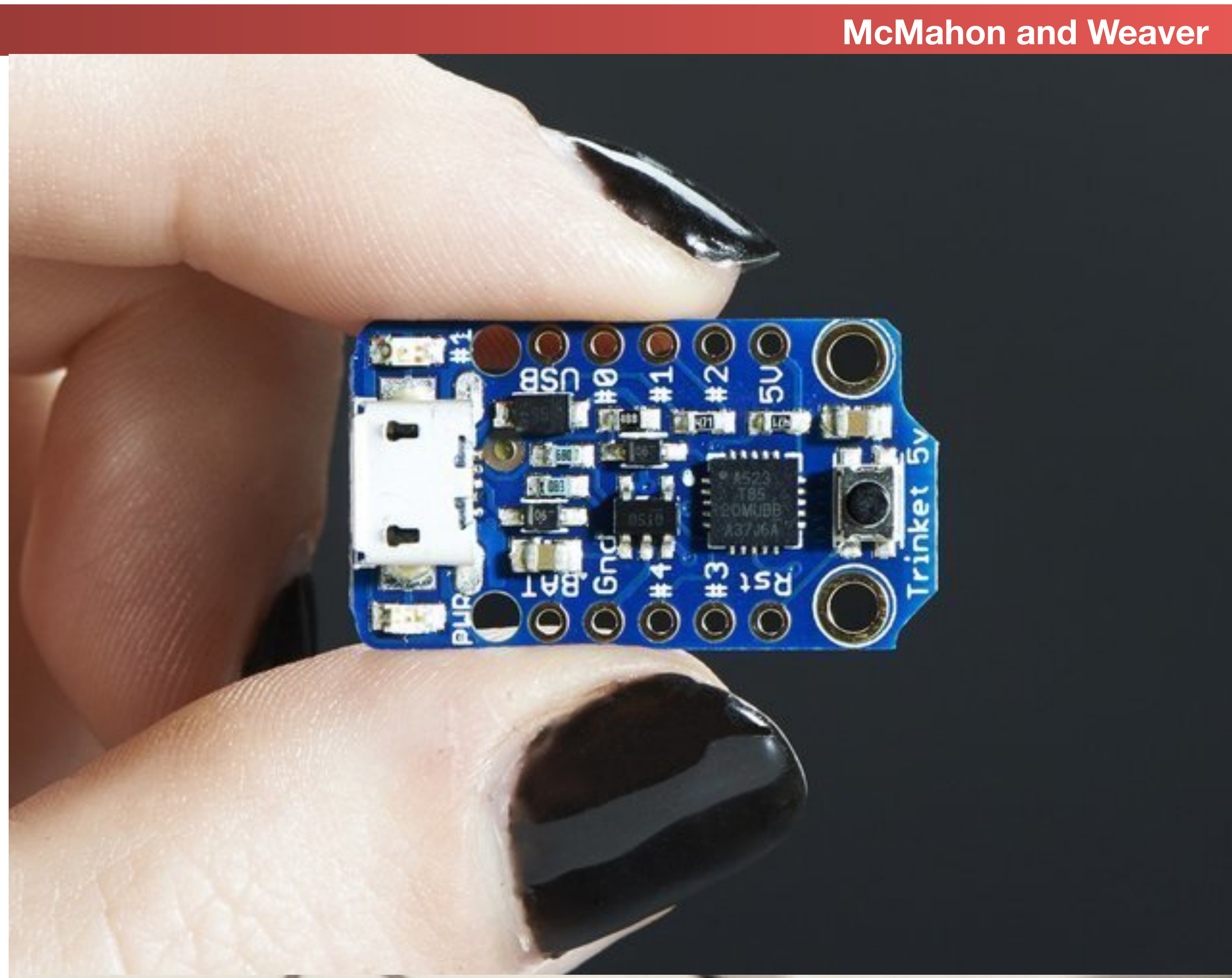
- *C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*
  - Kernighan and Ritchie
- Enabled first operating system not written in assembly language: *UNIX* - A portable OS!

Intro to C

- Why C?: *we can write programs that allow us to exploit underlying features of the architecture (memory management, special instructions) and do it in a* portable *way (C compilers universally available for all existing processor architectures)*
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!
  - It's popularity is mainly because of momentum:
    - Most Operating Systems (Linux, Windows) written in C (or C++),
    - as are world's most popular databases, including Oracle Database, MySQL, MS SQL Server, and PostgreSQL.
  - However, if you are starting a new project where performance matters consider either Go or Rust
    - Rust, "***C-but-safe***": By the time your C is (theoretically) correct with all the necessary checks it should be no faster than Rust.
    - Go, "***Concurrency***": Actually able to do practical concurrent programming to take advantage of modern multi-core microprocessors.

Recommendations on using C

- Use C/C++/Objective C if...
  1. You are starting from an existing C code base
  2. Or, you are targeting a ***very*** small computer
     - E.G. Adafruit "trinket": 16 MHz processor,
       8 kB of Flash, 512 B of SRAM, 512 B of EEPROM
     - KL-02: 2mm x 2mm containing a 32b ARM at 48 MHz,
       32 kB FLASH, 4 KB of SRAM
  3. Or, you are learning how things really ***work***
     - This class, CS162, etc...
- Otherwise, don't...
  - If you can tolerate GC pauses, go (aka golang) is really nice
    - Or C#, Java, Scala, Swift, etc...
  - If you can't, there is rust...

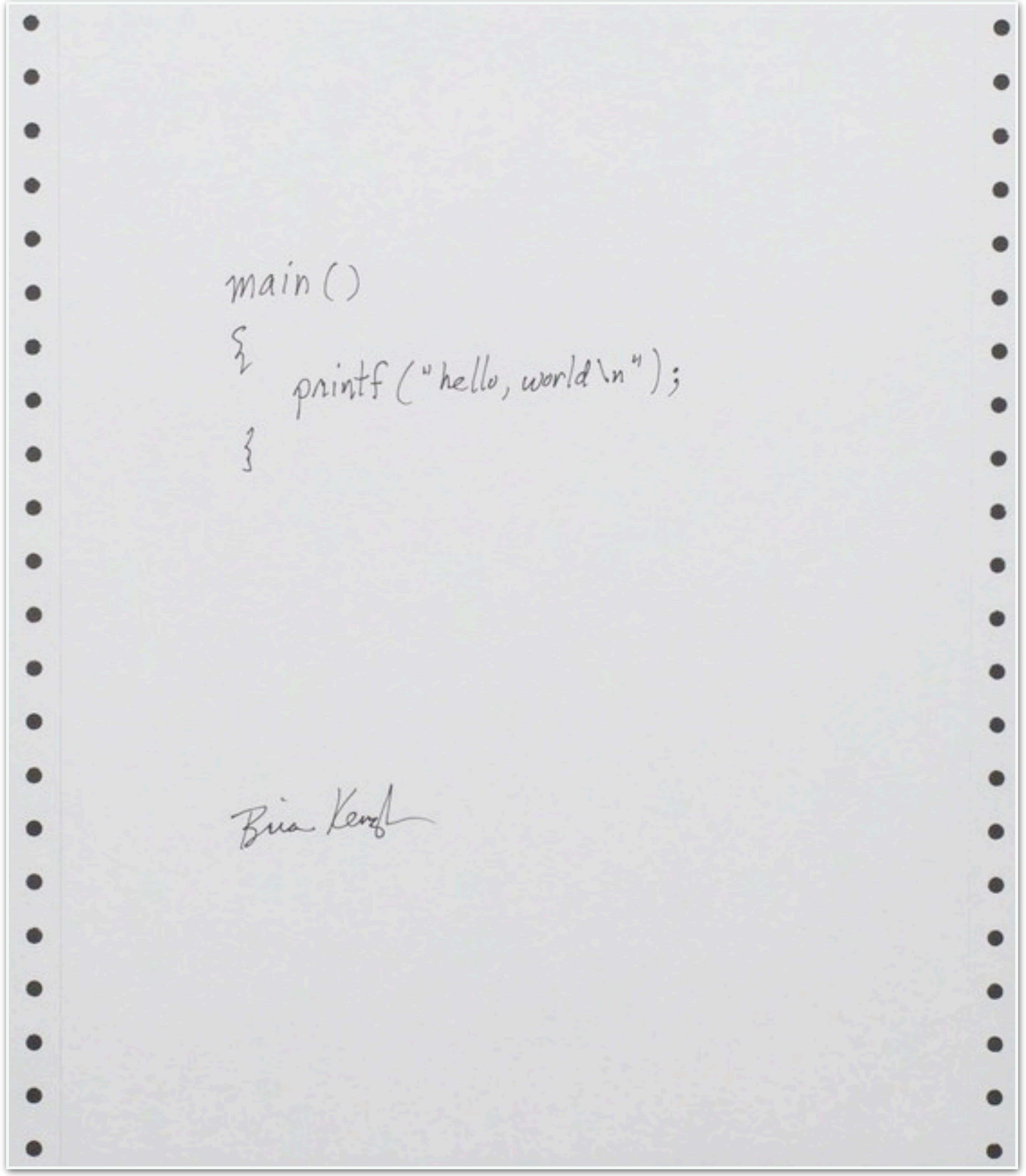


Disclaimer

- You will not learn how to fully code in C in these lectures! You’ll still need your C reference for this course
  - K&R is just about a ***must-have***
  - Useful Reference: “JAVA in a Nutshell,” O’Reilly
    - Chapter 2, “How Java Differs from C”
    - http://oreilly.com/catalog/javanut/excerpt/index.html
  - Brian Harvey’s helpful transition notes
    - On CS61C class website: pages 3-19
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management
  - Key security concept: All of the above are ***unsafe***: If your program contains an error in these areas it might not crash immediately but instead leave the program in an inconsistent (and often exploitable) state
  - Rife with "undefined behavior": Compiler-speak for 🤷‍♂️

Agenda

- Computer Organization
- Compile vs. Interpret
- C vs Java

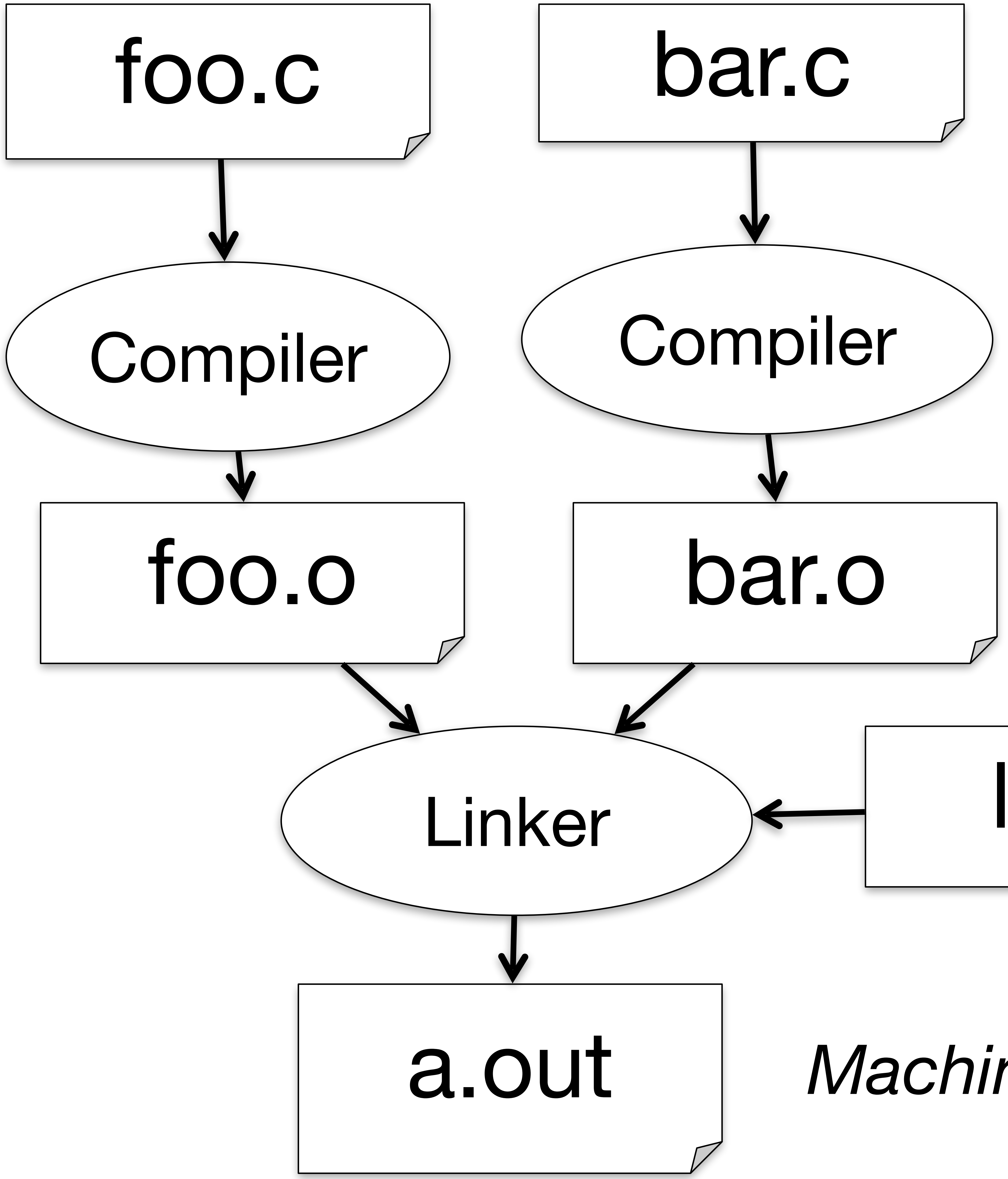


```
main()
{
    printf("hello, world\n");
}
```

Compilation: Overview

- C compilers map C programs directly into architecture-specific *machine code* (string of 1s and 0s)
  - The processor directly executes the machine code (the job of the hardware)
  - Unlike *Java*, which converts to architecture-independent “bytecode” which are interpeted by “virtual machine” and/or converted by a just-in-time compiler (JIT) to machine code.
  - Unlike *Python* environments, which converts to a byte code at runtime
    - These differ mainly in exactly when your program is converted to low-level machine instructions (“levels of interpretation”)
- For C, generally a two part process of compiling *source* files (.c) to *object* files (.o), then linking the .o files into executables;

C Compilation Simplified Overview (more later in course)



C source files (text)

Compiler/assembler combined here

Machine code object files

Pre-built object file libraries

Machine code executable file

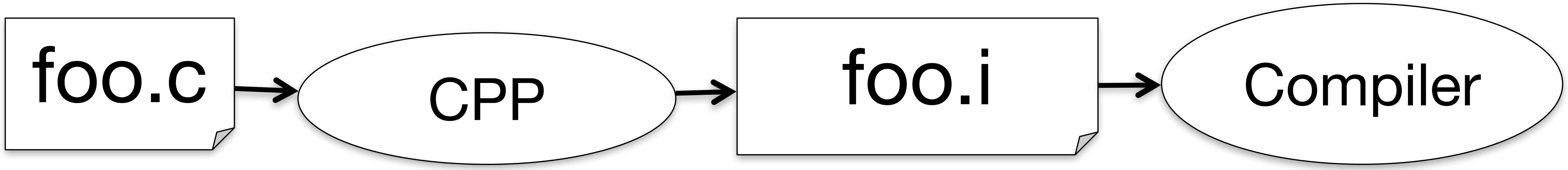
Compilation: Advantages

- Excellent run-time performance: generally much faster than Python or Java for comparable code (because it optimizes for a given architecture)
  - But these days, a lot of performance is in libraries:
    Plenty of people do scientific computation in ***python!?!***, because they have optimized libraries usually written in C and 99% of the execution takes place in the libraries
- Reasonable compilation time: only modified files are recompiled
  - Generally handled by the ***Makefile*** or larger build system

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. x86 vs. RISC-V) and the operating system (e.g., Windows vs. Linux vs. MacOS)
  - And even library versions under Linux. Linux is so bad we came up with "containers", that effectively ship around whole miniature OS images just to run single programs
- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture
- “Change → Compile → Run [repeat]” iteration cycle can be slow during development
  - but `make` only rebuilds changed pieces, and can do compiles in parallel on multiple cores (`make -j X`)
    - linker is sequential though → Amdahl’s Law

C Pre-Processor (CPP)



- C source files first pass through “macro preprocessor”, CPP, before compiler sees code
- CPP commands begin with “#”

  `#include “file.h”` /* ***Inserts*** file.h into output */

  `#include <stdio.h>` /* Looks for file in standard location */

  `#define M_PI (3.14159)` /* Define constant */

  `#if/#endif /* Conditional inclusion of text */`
- CPP replaces comments with a single space
- Use –save-temps option to gcc to see result of preprocessing
  - Full documentation at: http://gcc.gnu.org/onlinedocs/cpp/

CPP Macros:
A Warning...

- You often see C preprocessor macros defined to create small "functions"
  - But they aren't actual functions, instead it just changes the ***text*** of the program
  - In fact, all **`#include`** does is ***copy*** that file into the current file and replace arguments
  - Example:

```
#define twox(x) (x + x)…
twox(3);    ⇒ (3 + 3);
```

  - Could lead to interesting errors with macros

```
twox(y++); ⇒ (y++ + y++);
```

C vs. Java

	C	Java
Type of Language	Procedure Oriented	Object Oriented
Programming Unit	Function	Class = Abstract Data Type
Compilation	gcc hello.c creates machine language code	javac Hello.java creates Java virtual machine language bytecode
Execution	a.out loads and executes program	java Hello interprets bytecodes

hello, world	`#include<stdio.h>`
	`int main(void) {`
	`   printf("Hello\n");`
	`   return 0;`
	`}`
		`public class HelloWorld {`
		`   public static void main(String[] args) {`
		`     System.out.println("Hello");`
		`   }`
		`}`

Storage	Manual (malloc, free)	New allocates & initializes, Automatic (garbage collection) frees

From http://www.cs.princeton.edu/introcs/faq/c2java.html

C vs. Java

	C	Java
Comments	`/*` … `*/`	`/*` … `*/` or `//` … end of line
Constants	`#define`, `const`	`final`
Preprocessor	Yes	No
Variable declaration	At beginning of a block	Before you use it
Variable naming conventions	`sum_of_squares`	`sumOfSquares`
Accessing a library	`#include <stdio.h>`	`import java.io.File;`

From http://www.cs.princeton.edu/introcs/faq/c2java.html

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Example
`int`	Integer Numbers (including negatives) At least 16 bits, can be larger	`0, 78, -217, 0x7337`
`unsigned int`	Unsigned Integers	`0, 6, 35102`
`float`	Floating point decimal	`0.0, 3.14159, 6.02e23`
`double`	Equal or higher precision floating point	`0.0, 3.14159, 6.02e23`
`char`	Single character	`‘a’, ‘D’, ‘\n’`
`long`	Longer int, Size >= sizeof(int), at least 32b	`0, 78, -217, 301720971`
`long long`	Even longer int, size >= sizeof(long), at least 64b	`31705192721092512`

Integers: Python vs. Java vs. C

- C: **`int`** should be integer type that target processor works with most efficiently
- Only guarantee:

sizeof(**`long long`**) ≥ sizeof(**`long`**) ≥ sizeof(**`int`**) ≥ sizeof(**`short`**)

  - Also, **`short`** >= 16 bits, **`long`** >= 32 bits
  - All could be 64 bits

Language	sizeof(int)
Python	>=32 bits (plain ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

Specific Sized Numbers

- C only guarantees minimum and relative size of "**`int`**" "**`short`**" etc...
  - But sometimes you need to know the exact width of something
- **`{u|}int{#}_t`**
  - Whether or not it is unsigned
  - Integer
  - number of bits (8, 16, 32, 64)
- **`uint8_t`** is an unsigned 8-bit integer
- **`int64_t`** is a signed 64-bit integer
  - All these are defined in an auxiliary header file **`stdint.h`** rather than in the language

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
const double the_law = 2.99792458e8;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

Typed Functions in C

- You need to declare the return type of a function when you declare it
  - Plus the types of any arguments
- You also need to declare functions before they are used
  - Usually in a separate header file, eg:

    ```
    int number_of_people();
    float dollars_and_cents();
    int sum(int x, int y);
    ```
- **`void`** type means "This returns nothing"

```
int number_of_people ()
{
  return 3;
}


float dollars_and_cents ()
{
  return 10.33;
}


int sum ( int x, int y)
{
  return x + y;
}
```

Structs in C

- Structs are structured groups of variables, e.g.,

```
typedef struct {
  int length_in_seconds;
  int year_recorded;
} Song;

Song song1;
song1.length_in_seconds =  213;
song1.year_recorded      = 1994;

Song song2;
song2.length_in_seconds =  248;
song2.year_recorded      = 1988;
```

Dot notation: **`x.y = value`**

A First C Program: Hello World

Original C:

```
main()
{
  printf("Hello World\n");
}
```

ANSI Standard C:

```
#include <stdio.h>
/* main's return type is an
   integer */
int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

C Syntax: `main`

- When C program starts
  - C executable **`a.out`** is loaded into memory by operating system (OS)
  - OS sets up stack, then calls into C runtime library,
  - Runtime first initializes memory and other libraries,
  - then calls your procedure named **`main()`**
- We’ll see how to retrieve command-line arguments in **`main()`** later…

A Second C Program: Compute Table of Sines

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

int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("\nCompute a table of the
sine function\n\n");
    /* obtain pi once for all       */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n",
            pi);
    printf("angle     Sine \n");
```

```c
 angle_degree = 0;
 /* initial angle value */
 /* scan over angle     */
 while (angle_degree <= 360)
 /* loop until angle_degree > 360 */
    {
       angle_radian = pi*
             angle_degree/180.0;
       value = sin(angle_radian);
       printf (" %3d      %f \n ",
              angle_degree, value);
       angle_degree += 10;
       /* increment the loop index */
    }
 return 0;
}
```

Second C Program Sample Output

```
Compute a table of the sine function

Value of PI = 3.141593

angle      Sine
   0     0.000000
  10     0.173648
  20     0.342020
  30     0.500000
  40     0.642788
  50     0.766044
  60     0.866025
  70     0.939693
  80     0.984808
  90     1.000000
 100     0.984808
 110     0.939693
 120     0.866025
 130     0.766044
 140     0.642788
....
```

C Syntax: Variable Declarations

- Similar to Java, but with a few minor but important differences
  - All variable declarations must appear before they are used
  - All must be at the beginning of a block.
  - A variable may be initialized in its declaration;
    if not, it holds garbage! (the contents are undefined)
- Examples of declarations:
  - Correct: **`{ int a = 0, b = 10; ...`**
  - Incorrect: **`for (int i = 0; i < 10; i++) { ...`**

Newer C standards are more flexible about this, especially in for loops

An Important Note: Undefined Behavior…

- A lot of C has “Undefined Behavior”
  - The language definition basically says "We don't know what will happen, nor care for that matter"
  - This means it is often un***predictable*** behavior
    - It will run one way on one compiler and computer…
    - But some other way on another
    - Or even just be different each time the program is executed! ***EVEN ON THE SAME INPUT!***
- Often contributes to “heisenbugs”
  - Bugs that seem random/hard to reproduce
  - (In contrast to “bohrbugs” which are deterministic)

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs (shows Java's legacy) in terms of control flow
  - A statement can be a `{}` of code or just a standalone statement
- if-else
  - `if (expression) statement`
    - `if (x == 0) y++;`
    - `if (x == 0) {y++;}`
    - `if (x == 0) {y++; j = j + y;}`
  - `if (expression) statement1 else statement2`
    - There is an ambiguity in a series of if/else if/else if you don't use `{}`s, so always use `{}`s to block the code
    - In fact, it is a bad C habit to not always have the statement in `{}`s, it has resulted in some amusing errors...
- while
  - `while (expression) statement /* Evaluates expression at loop start */`
  - `do statement while (expression); /* Evaluates expression at loop end */`

C Syntax : Control Flow (2/2)

- for

  -
    ```
    for (initialize; check; update) statement
    /* Check is evaluated at the start of the loop like while */
    ```
- switch

  -
    ```
    switch (expression){
      case const1:    statements
      case const2:    statements
      default:        statements
    }
    ```
  - `break; /* need to break out of case */`
  - Note: until you do a break statement things keep executing in the switch statement
  - You can also use **`break`** to exit a loop, but it is bad for to do so
- C also has **`goto`**
  - But it can result in spectacularly bad code if you use it, so don't!
    Makes your code hard to understand, debug, and modify.

C Syntax: True or False

- What evaluates to *FALSE* in C?
  - 0 (integer)
  - NULL (a special kind of pointer that is also 0: more on this later)
  - ***No explicit Boolean type in old-school C***
    - Often you see `#define bool (int)`
    - Then `#define false 0`
    - Alternative approach: include a header file `#include <stdbool.h>` to provide a boolean type
  - Basically anything where all the bits are 0 is false
- What evaluates to *TRUE* in C?
  - ***Anything*** that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true

C and Java operators nearly identical

- arithmetic: `+, -, *, /, %`
- assignment: `=`
- augmented assignment: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- bitwise logic: `~, &, |, ^`
- bitwise shifts: `<<, >>`
- boolean logic: `!, &&, ||`
- equality testing: `==, !=`
- subexpression grouping: `()`
- order relations: `<, <=, >, >=`
- increment and decrement: `++` and `--`
- member selection: `.`, `->`
  - This is slightly different than Java because there are both structures and pointers to structures, more later
- conditional evaluation: `? :`

Our Tip of the Day… Valgrind

- Valgrind turns most unsafe "heisenbugs" into "bohrbugs"
  - It adds almost all the checks that Java does but C does not
  - The result is your program ***immediately*** crashes where you make a mistake
  - It is installed on the lab machines
- Nick's scars from his 60C experience:
  - First C project, spent an entire day tracing down a fault...
    - Program would crash in a memory allocation in a **`printf`** statement only when I had a lot of input and in unpredictable ways
  - That turned out to be a <= instead of a < in initializing an array in a completely different part of the program!

Agenda

- Pointers
- Arrays in C

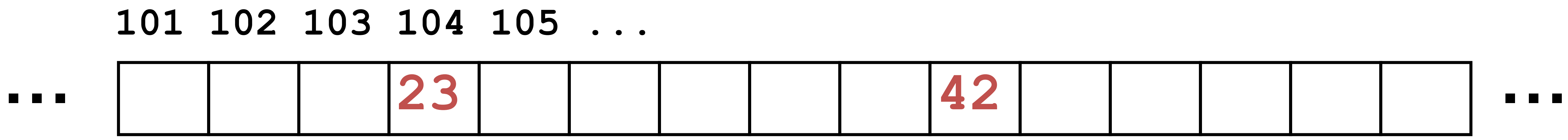
Remember What We Said Earlier About Buckets of Bits?

- C's memory model is that conceptually there is simply one ***huge*** bucket of bits
  - Arranged in bytes
- Each byte has an ***address***
  - Starting at 0 and going up to the maximum value (0xFFFFFFFF on a 32b architecture)
    - 32b architecture means the # of bits in the address
- We commonly think in terms of "words"
  - Least significant bits of the address are the offset within the word
  - Word size is 32b for a 32b architecture, 64b for a 64b architecture:
    A word is big enough to hold an ***address***

0xFFFFFFFC	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF8	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF4	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF0	xxxx	xxxx	xxxx	xxxx
0xFFFFFFEC	xxxx	xxxx	xxxx	xxxx
. . .	. . .	. . .	. . .	. . .
0x14	xxxx	xxxx	xxxx	xxxx
0x10	xxxx	xxxx	xxxx	xxxx
0x0C	xxxx	xxxx	xxxx	xxxx
0x08	xxxx	xxxx	xxxx	xxxx
0x04	xxxx	xxxx	xxxx	xxxx
0x00	xxxx	xxxx	xxxx	xxxx

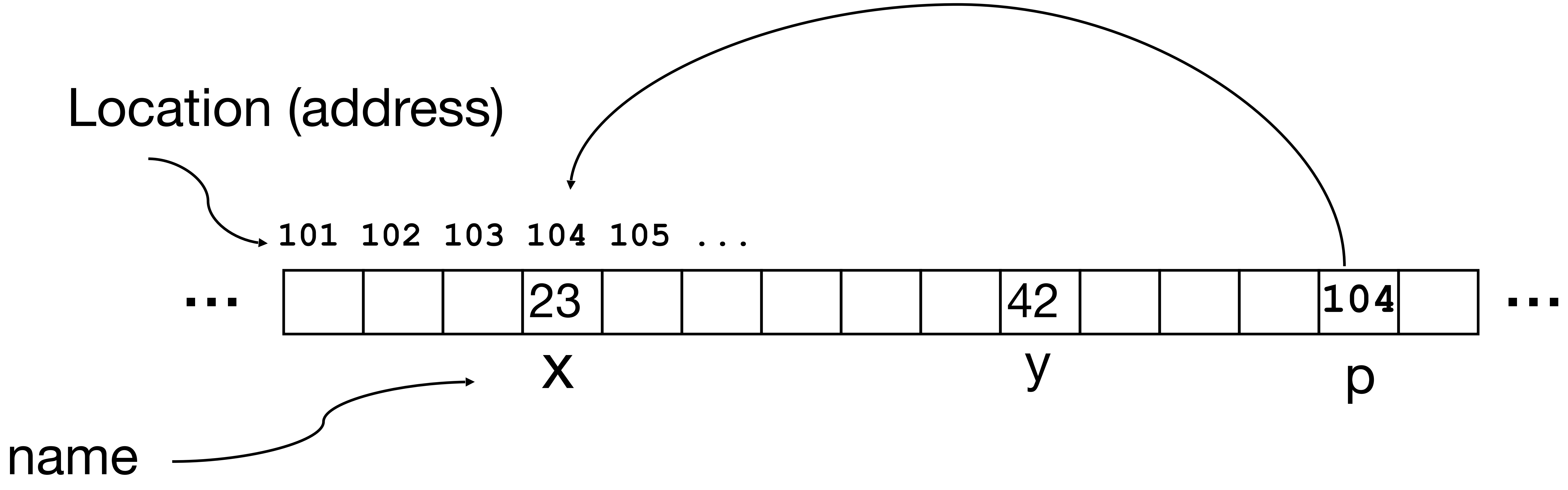
Address vs. Value

- Consider memory to be a ***single*** huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - For addresses do we use signed or unsigned numbers? Negative address?!
    - Answer: Signed
- Don’t confuse the address referring to a memory location with the value stored there



Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



Pointer Syntax

- `int *p;`
  - Tells compiler that variable p is address of an `int`
- `p = &y;`
  - Tells compiler to assign address of `y` to `p`
  - `&` called the "address operator" in this context
- `z = *p;`
  - Tells compiler to assign the value at address in `p` to `z`
  - `*` called the "dereference operator" in this context
- **`*p = 64;`**
  - Tells the compiler to assign the value 64 into the memory location pointed to by **`p`**

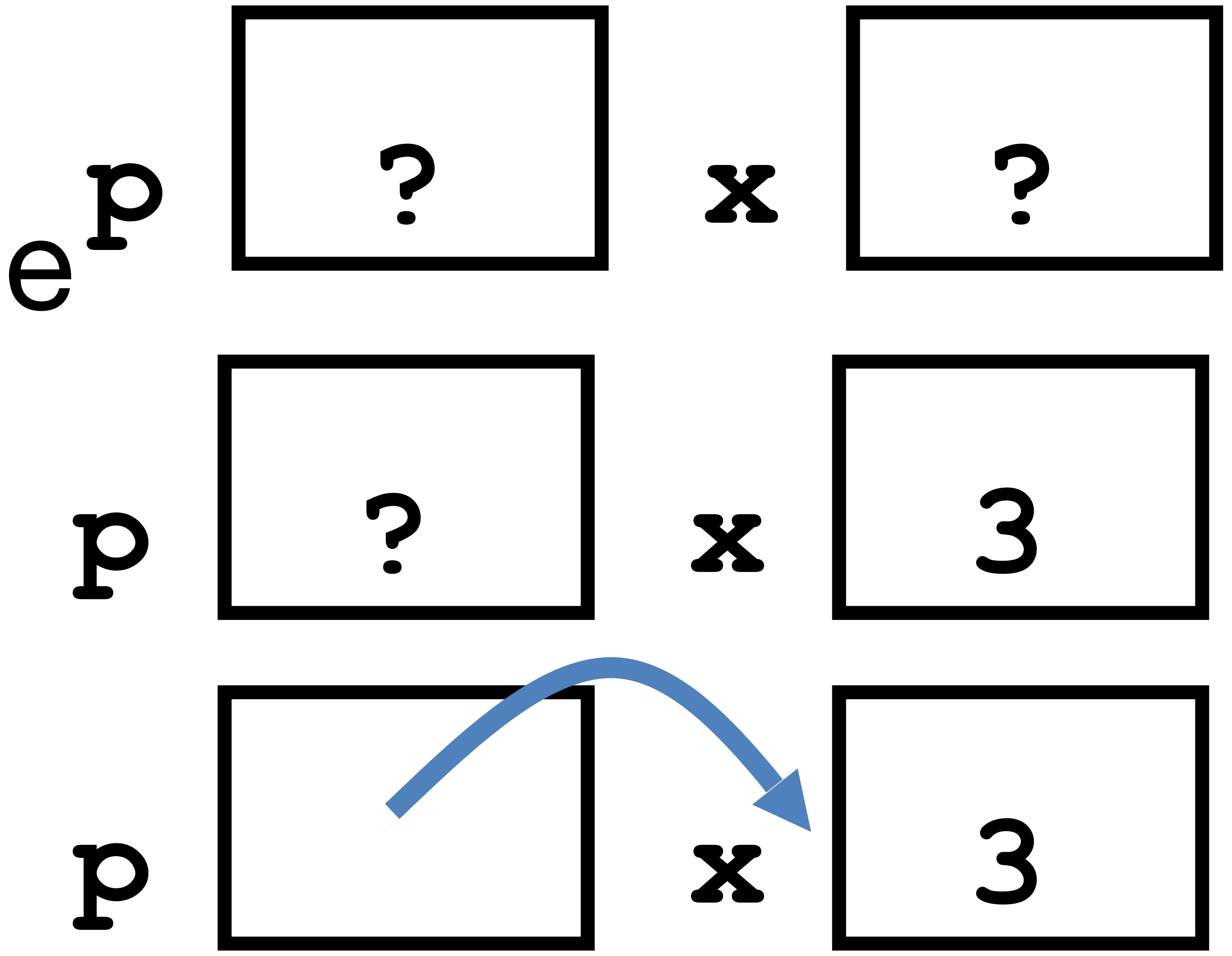
Creating and Using Pointers

- How to create a pointer:

& operator: get address of a variable

```
int *p, x;        x = 3;

                  p = &x;
```



Note the "*" gets used two different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

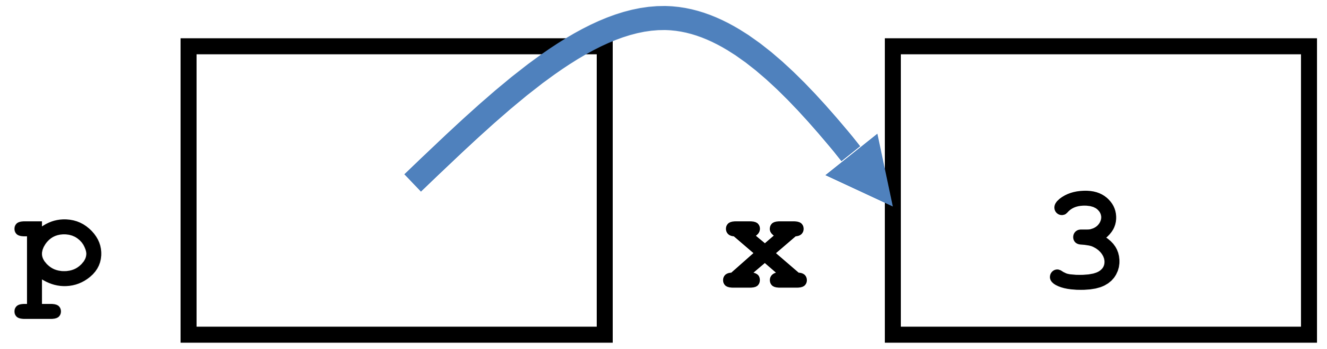
- How get a value pointed to?

"*" (dereference operator): get the value that the pointer points to

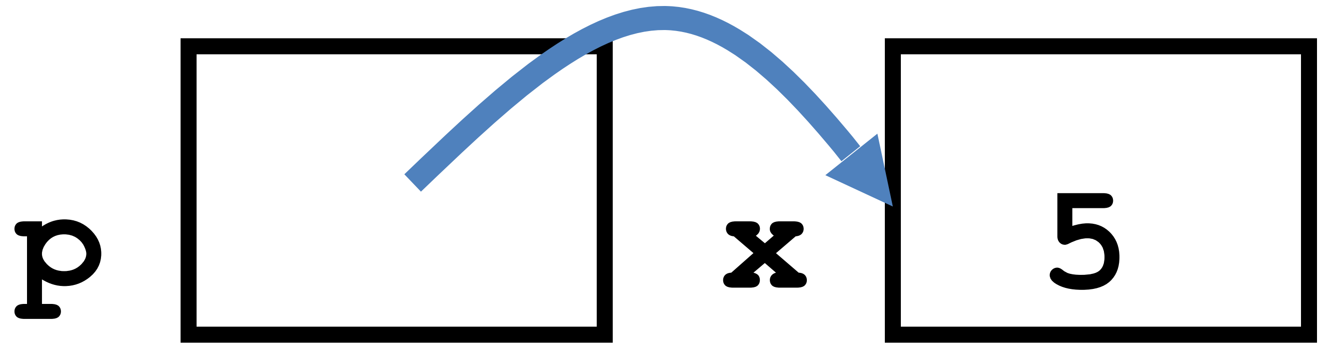
```
printf(“p points to %d\n”,*p);
```

Using Pointer for Writes

- How to change a variable pointed to?
    - Use the dereference operator `*` on left of assignment operator `=`



```
*p = 5;
```



Pointers and Parameter Passing

- Java and C pass basic parameters “by value”: Procedure/function/method gets a copy of the parameter, *so changing the copy cannot change the original*

```
void add_one (int x)
{
    x = x + 1;
}
int y = 3;
add_one(y);
```

y remains equal to 3

Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

```
void add_one (int *p)
{
    *p = *p + 1;
}
int y = 3;

add_one(&y);
```

y is now equal to 4

Types of Pointers

- Pointers are used to point to any kind of data (**`int`**, **`char`**, a **`struct`**, etc.)
- Normally a pointer only points to one type (**`int`**, **`char`**, a **`struct`**, etc.).
  - **`void *`** is a type that can point to anything (generic pointer)
  - Use **`void *`** sparingly to help avoid program bugs, and security issues, and other bad things!
- You can have pointers to pointers (and beyond)
  - **`int ****x;`** Declares x as a pointer to a pointer to a pointer to a pointer to an **`int`**!
- You can even have pointers to functions…
  - **`int (*fn) (void *, void *) = &foo`**
    - **`fn`** is a function that accepts two **`void *`** pointers and returns an **`int`** and is initially pointing to the function **`foo`**.
    - **`(*fn)(x, y)`** will then call the function

More C Pointer Dangers

- *Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!*
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```

Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Point;


Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too,
  copies contents of p2
  to p1 */
p1 = p2;
```

Casting and Casting Pointers

- You can cast (change the type) of basic C types which converts them

  -
    ```
    int x; float y; ...
    y = (float) x;
    ```
    - Will take the value in **x**, convert it to a floating point number, and assign the resulting floating point value to **y**.
- For pointers it only changes how they are interpreted

  -
    ```
    void foo(void *v) {
      ((Point *) v)->x = 42;
    }
    ```
    - Treats **v** as a pointer to a **Point** structure, sets the value of the **x** field in the structure pointed to by **v** to the value 42
    - If it turns out **v** is a pointer to some other type of data... Well, Undefined Behavior time!

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
    - Otherwise we'd need to copy a huge amount of data
    - You notice in Java that more complex objects are passed by reference.... Under the hood this is a pointer
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—coming up next time
    - Dangling references and memory leaks

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn’t produce efficient code
  - Computers 100,000x times faster today, compilers are also way way way way better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today’s compilers produce much better code, so may not need to use raw pointers in application code
  - Most other languages use “pass by reference” for objects, which is semantically similar but with checks for misuse
- Low-level system code still needs low-level access via pointers
  - And compilers basically convert "pass by reference" into pointer-based code