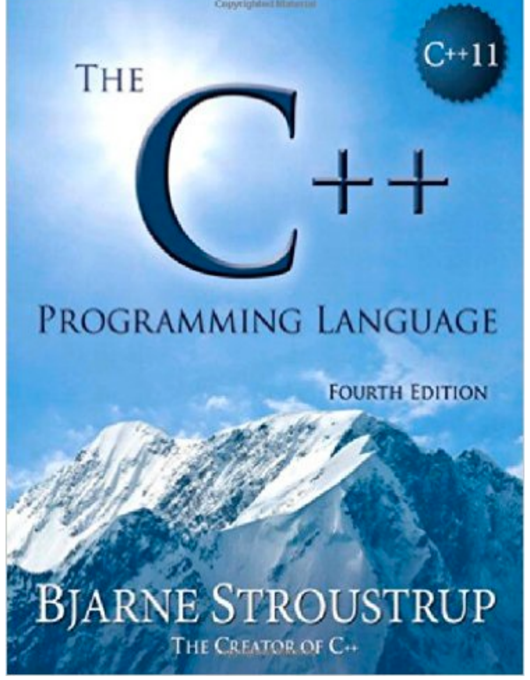
Principles of Programming Languages

CS 245 — Spring 2019
kmicinski.com/cs245

Kristopher Micinski

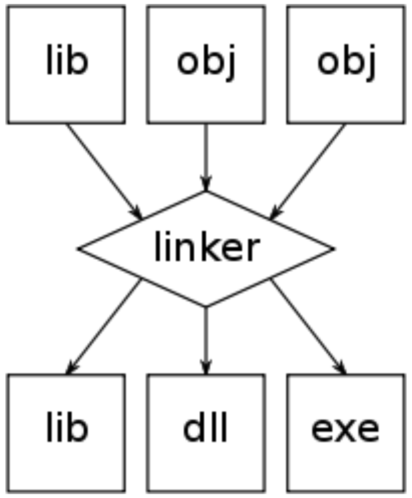
Jocelyn Dunkley   Myriam (Mimi) Benkoussa

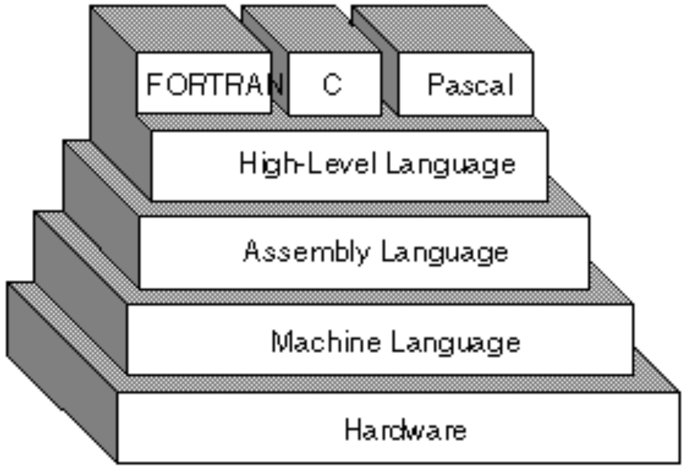


This class is about the **principles** of programming languages:

what kinds of languages are there?
how do they work *"under the hood"*?
how are they implemented?

We're also going to learn 4-8, *or more* (depending on how we count), different programming languages!





First, a bit of history…

Humans create tools to help them solve problems

Programming Languages allow us to *precisely* express solutions to certain classes of problems

How do we (partially) formally specify what we want?

How do we (partially) formally specify what we want?

How do we write a program that does what we want?

How do we (partially) formally specify what we want?

How do we write a program that does what we want?

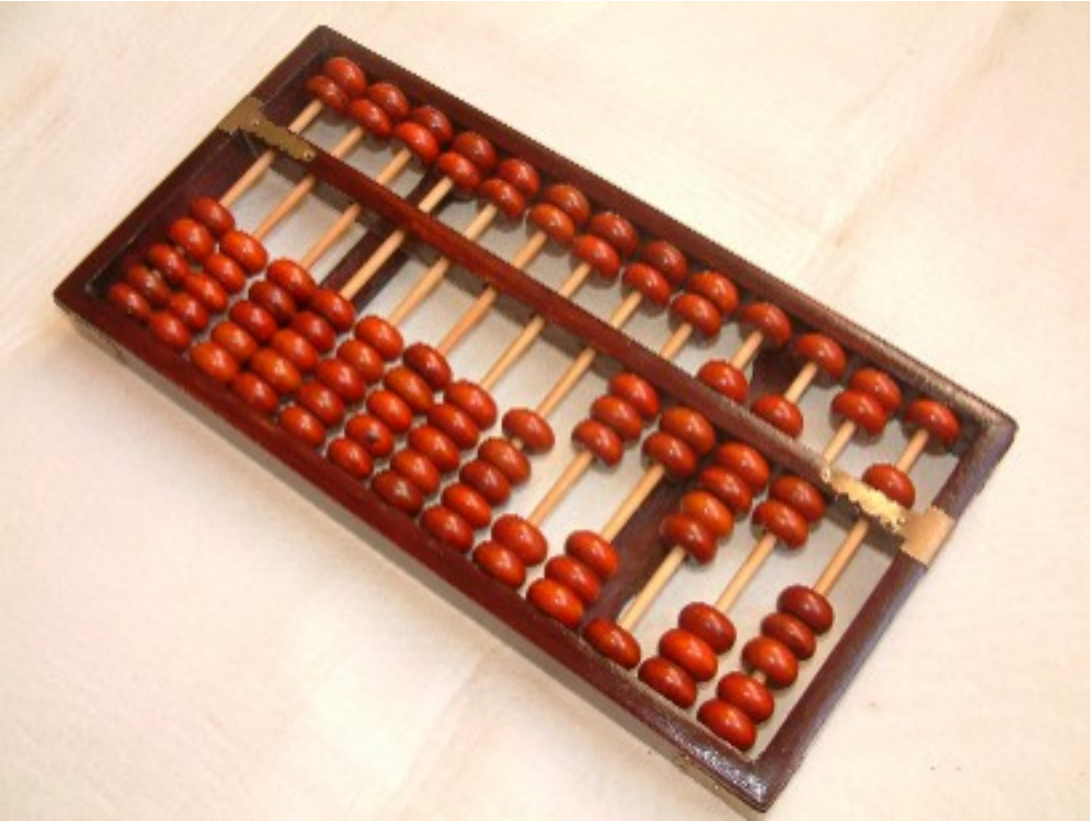
How do we run that program?

How do we (partially) formally specify what we want?

How do we write a program that does what we want?

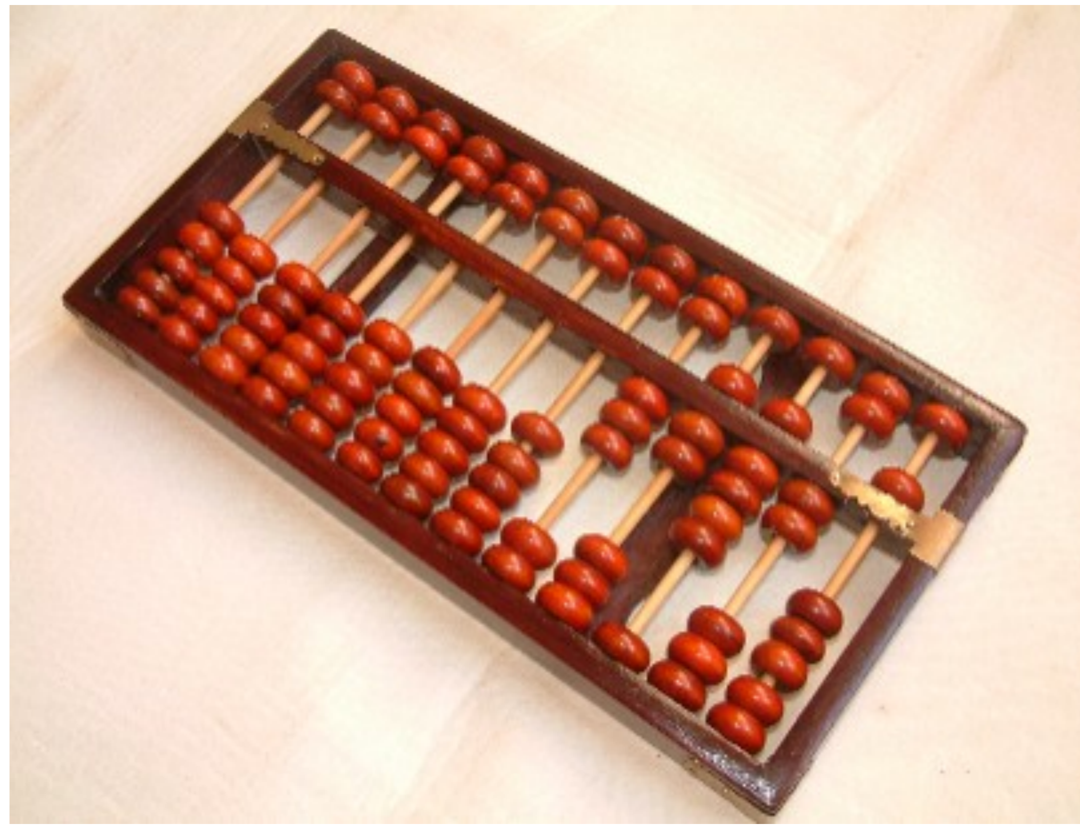
How do we run that program?

How can we be convinced that program is correct?



~500 BC

Allows us to solve arithmetic problems (if you know how to use it)



~500 BC

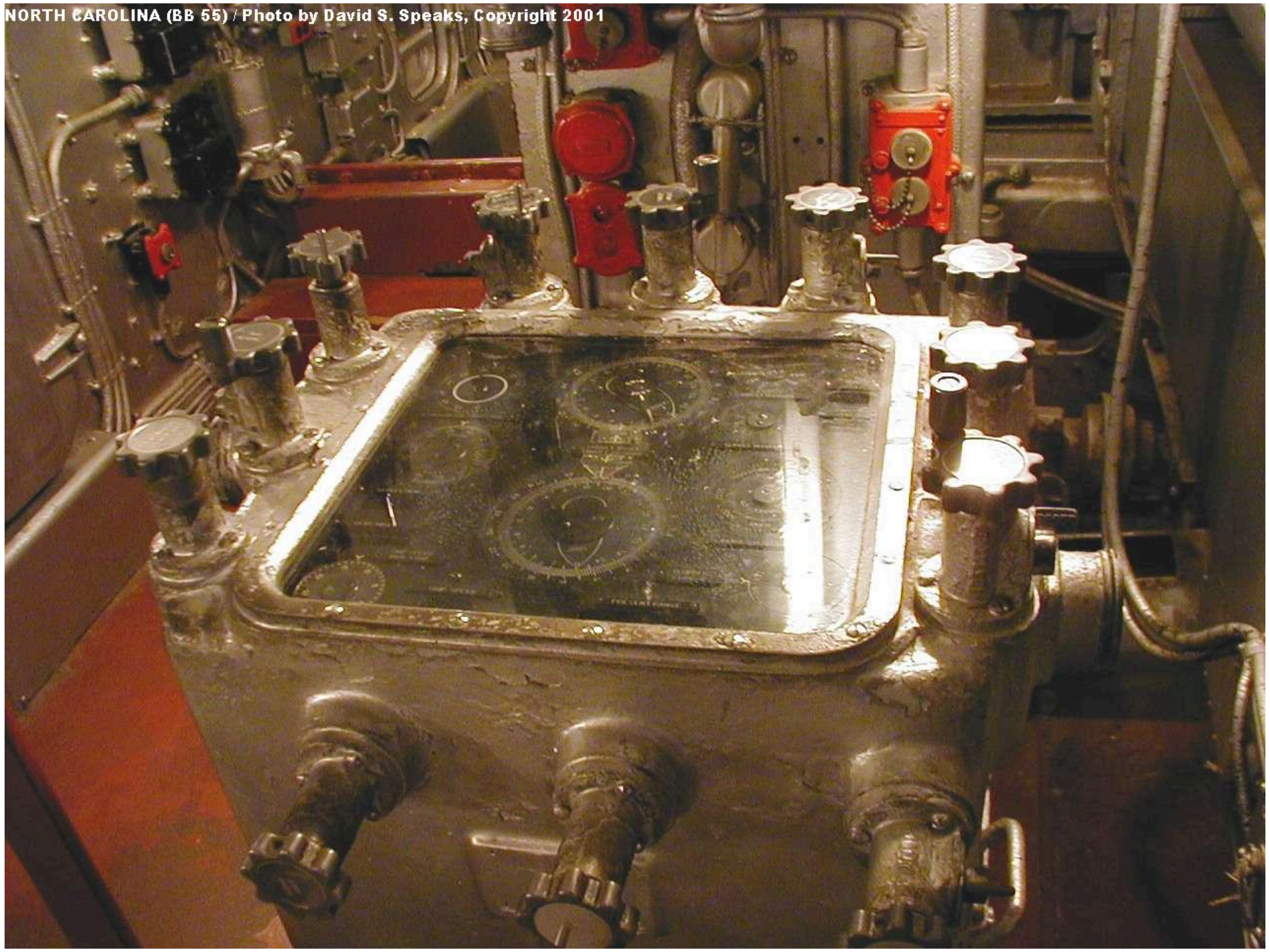
Not really a program, but machine that allows us to perform computation

Jacquard Loom (1804)

Reads punched cards to build, e.g., tapestries

You write a **program** to build the material

https://www.youtube.com/watch?v=OlJns3fPItE

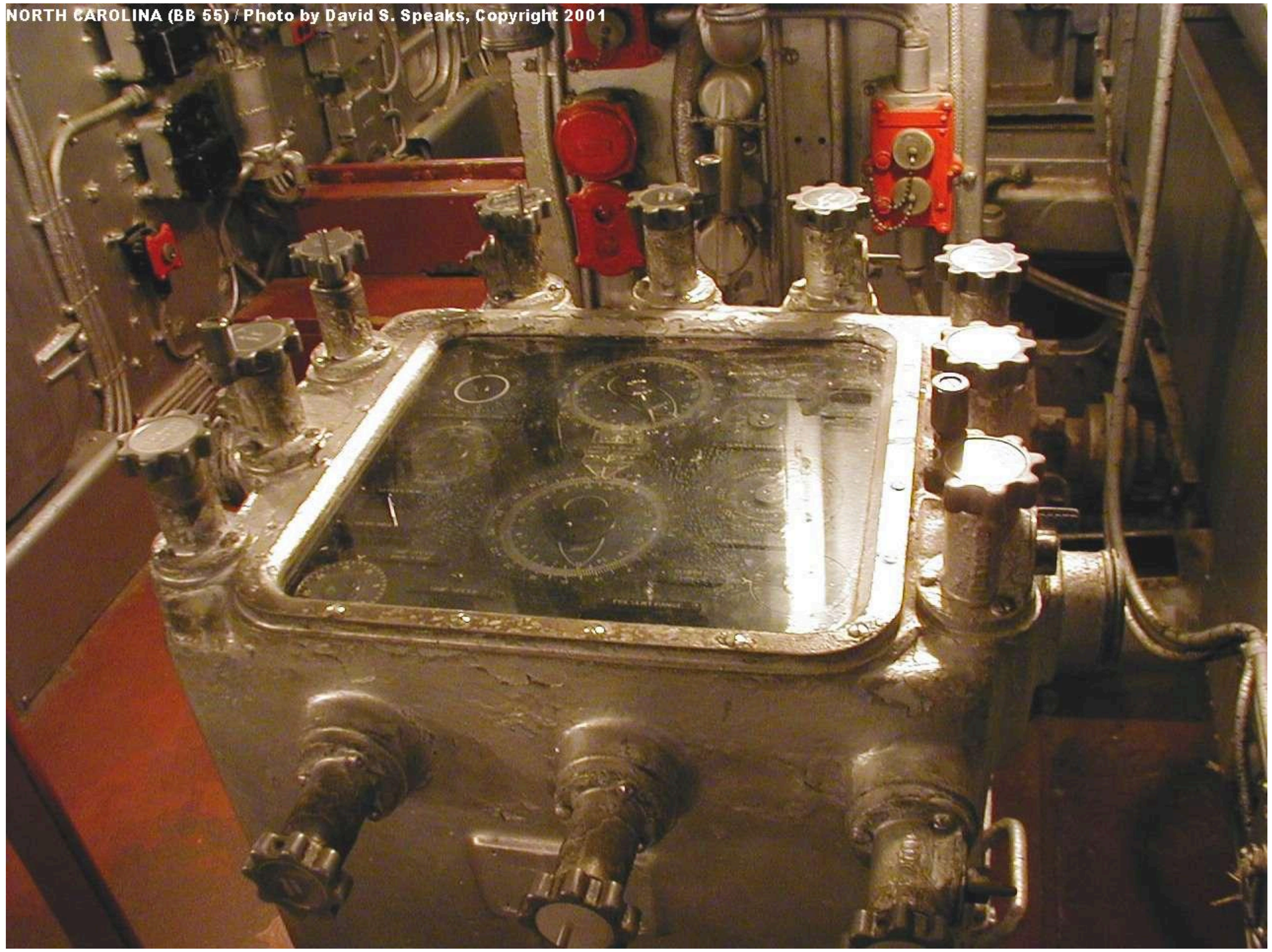


Analog targeting computer (USS North Carolina)

Helps aim guns given target distance / speed

Works using gears..

Not general purpose!!!



Analog targeting computer (USS North Carolina)

Helps aim guns given target distance / speed

Works using gears..

Ada Lovelace

Translated memoir describing general-purpose computer (1842)

Wrote notes of how to use this to compute Bernoulli numbers

Alonzo Church

Created lambda calculus (1936)

Alonzo Church

Created lambda calculus (1936)

Lambda calculus:
mathematically specified language

Notably: a **general purpose** language

But **ridiculously simple**

```
e ::=
    x
  | λx. e
  | e e
```

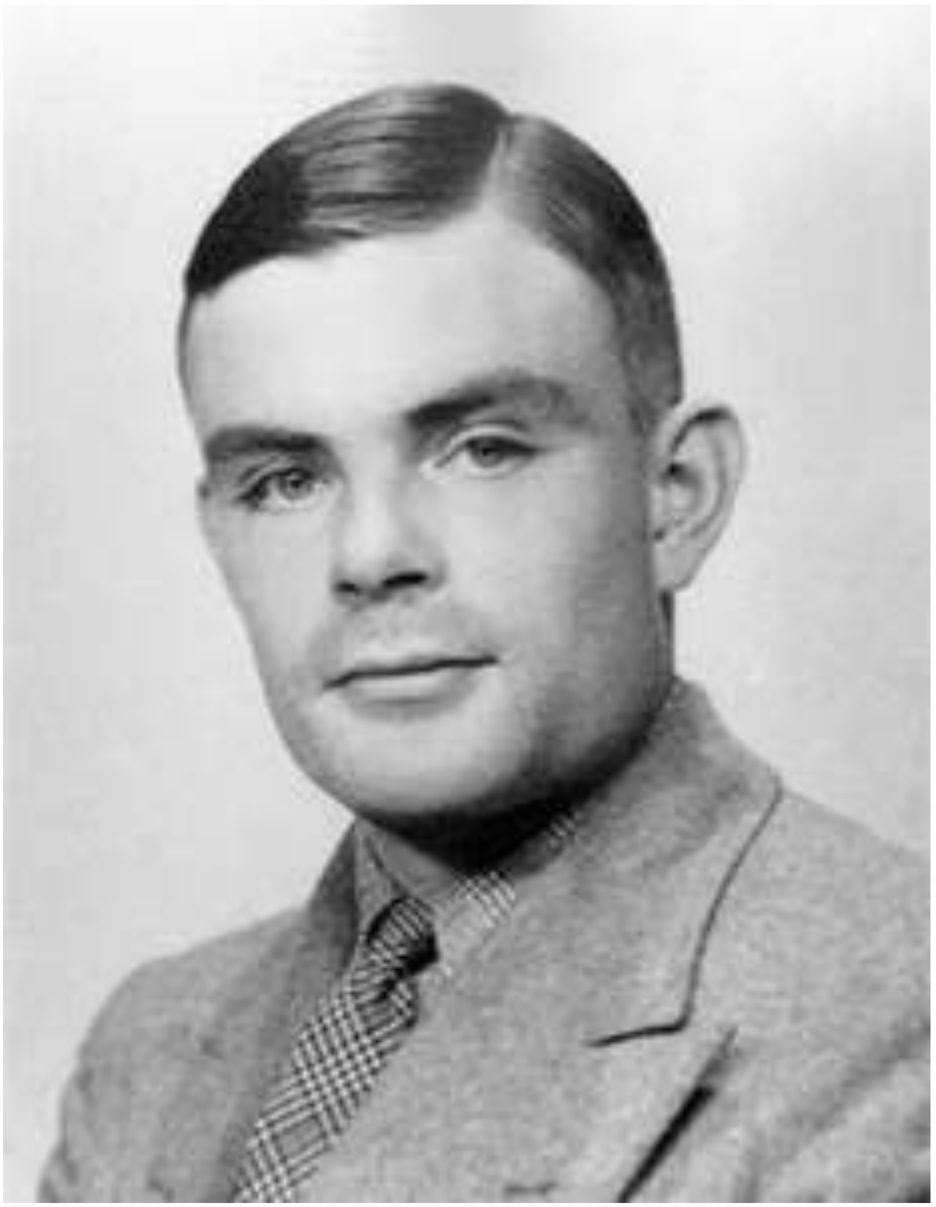
At this time, there were no “computer scientists”
Most people studying this were mathematicians, engineers, etc…

Also, nobody had actually **built** a general-purpose computer

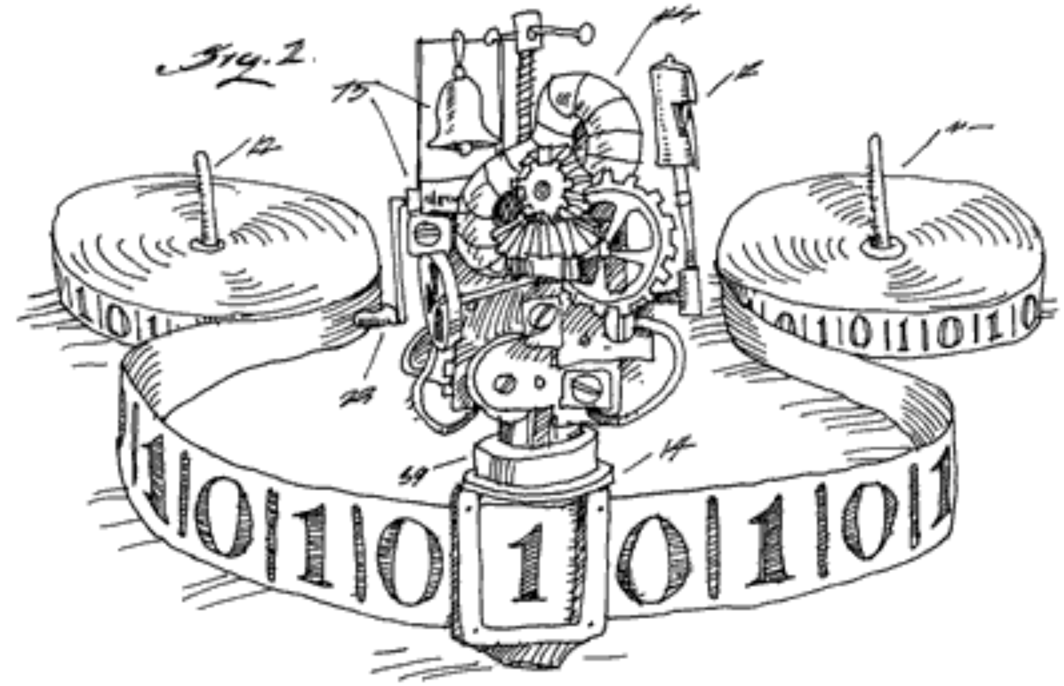
So we were free to think about what languages would look like without thinking about hardware

Alan Turing (Church's Student)

Invented Turing machines (1936)

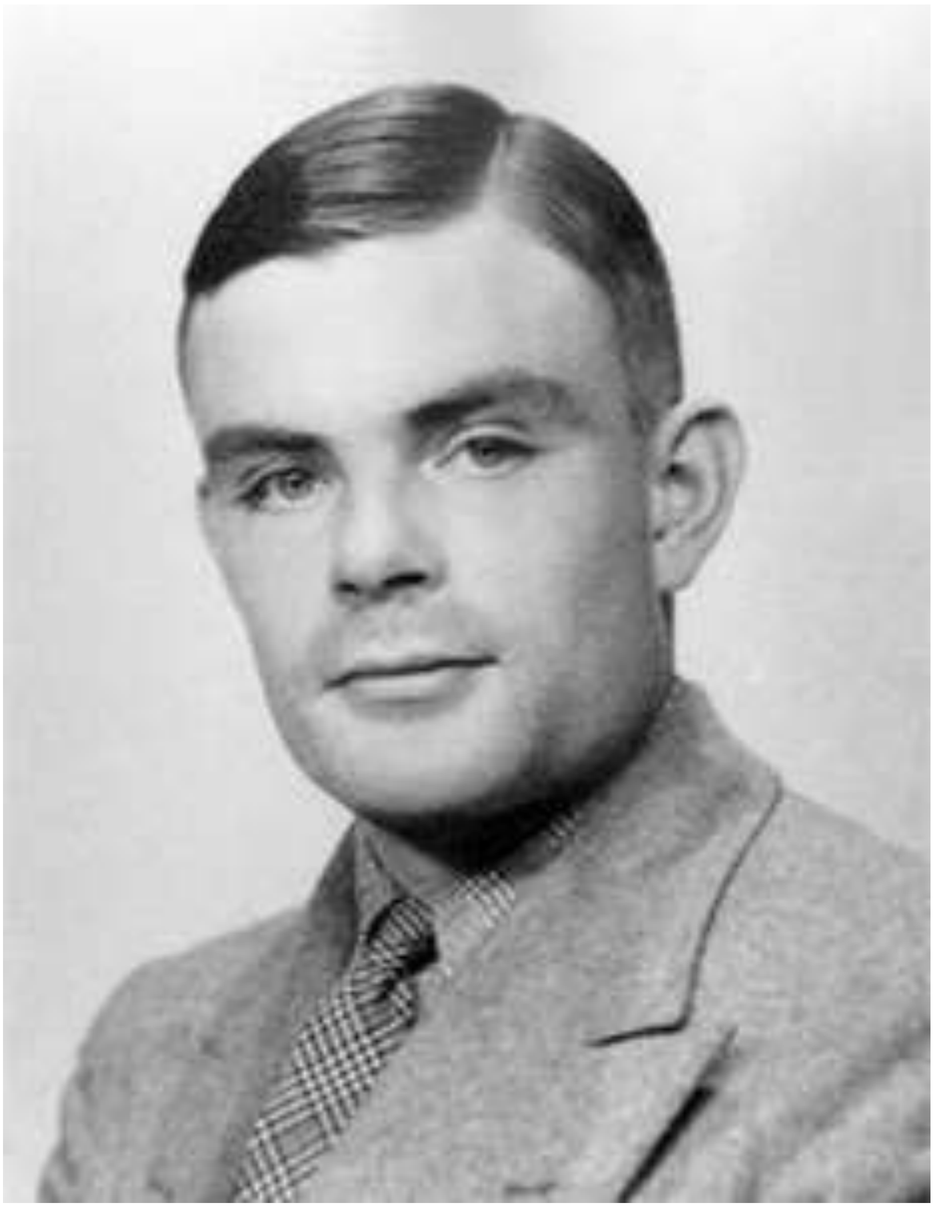


Alan Turing

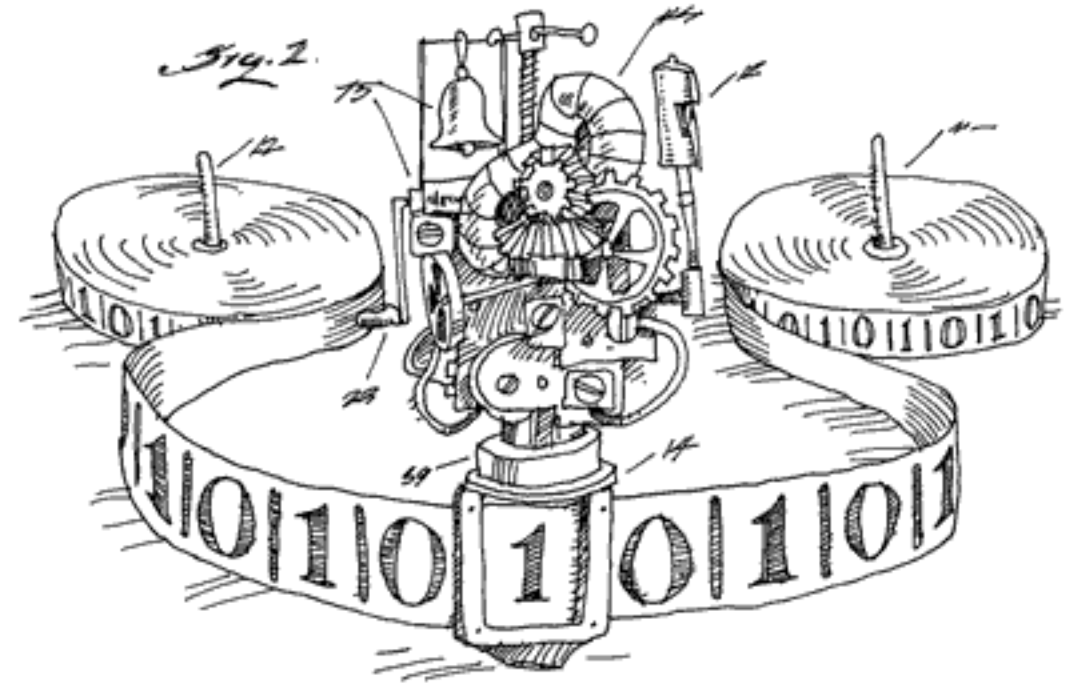


Model of computation that includes:

- Read / Write Tape (memory)
- Head (current position on tape)
- Current state
- Instructions



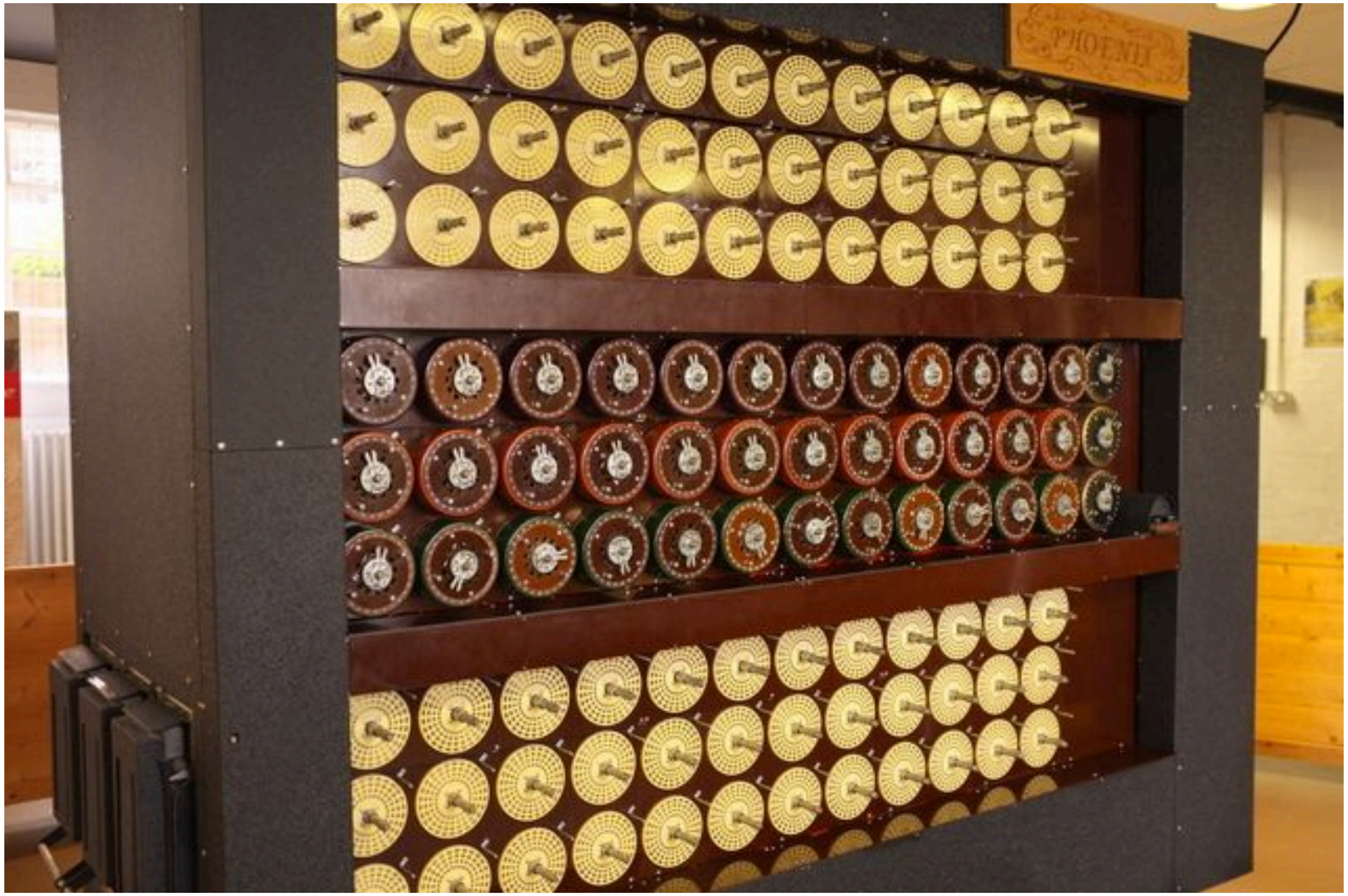
Alan Turing



Model of computation that includes:

- Read / Write Tape (memory)
- Head (current position on tape)
- Current state
- Instructions

Church-Turing Thesis:
Any **computable** function can be computed by *some* Turing machine



Turing’s Bombe

Cracks enigma by semi-brute-force exploiting a flaw in German code scheme

Alan Turing

Even after his work cracking enigma, Turing was prosecuted for his homosexuality

He committed suicide at the age of 41

Several general-purpose languages came about, mainly targeted at mechanical computers in the early 50s

These languages mostly resembled Turing machines and grew into the assembly languages we see today!

Corrado Böhm

Wrote first meta-circular compiler (1951)

In only 114 lines of code

John Backus

1954 — FORTRAN invented at IBM

First general purpose language w/ compiler that had widespread use

Also invented BNF

Grace Hopper

1955—Writes FLOW-MATIC (inspires COBOL)

John McCarthy

1958—Invents LISP (inspiration for Scheme/Racket)

Gets variable scoping **wrong** because he failed to read ***all*** of Church's 1936 paper…

“To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn’t understand the rest of his book, so I wasn’t tempted to try to implement his more general mechanism for defining functions.”

“I must confess that I regarded this difficulty as just a bug and expressed confidence that Steve Russell would soon fix it. He did fix it but by inventing the so-called FUNARG device that took the lexical environment along with the functional argument. Similar difficulties later showed up in Algol 60, and Russell’s turned out to be one of the more comprehensive solutions to the problem.”

—John McCarthy, History of Lisp, 1979

http://jmc.stanford.edu/articles/lisp/lisp.pdf

Remember this when we talk about closures :-)

Margret Hamilton

1960s: leads team that writes assembly code for Apollo rockets / lunar module / command module

Margret Hamilton

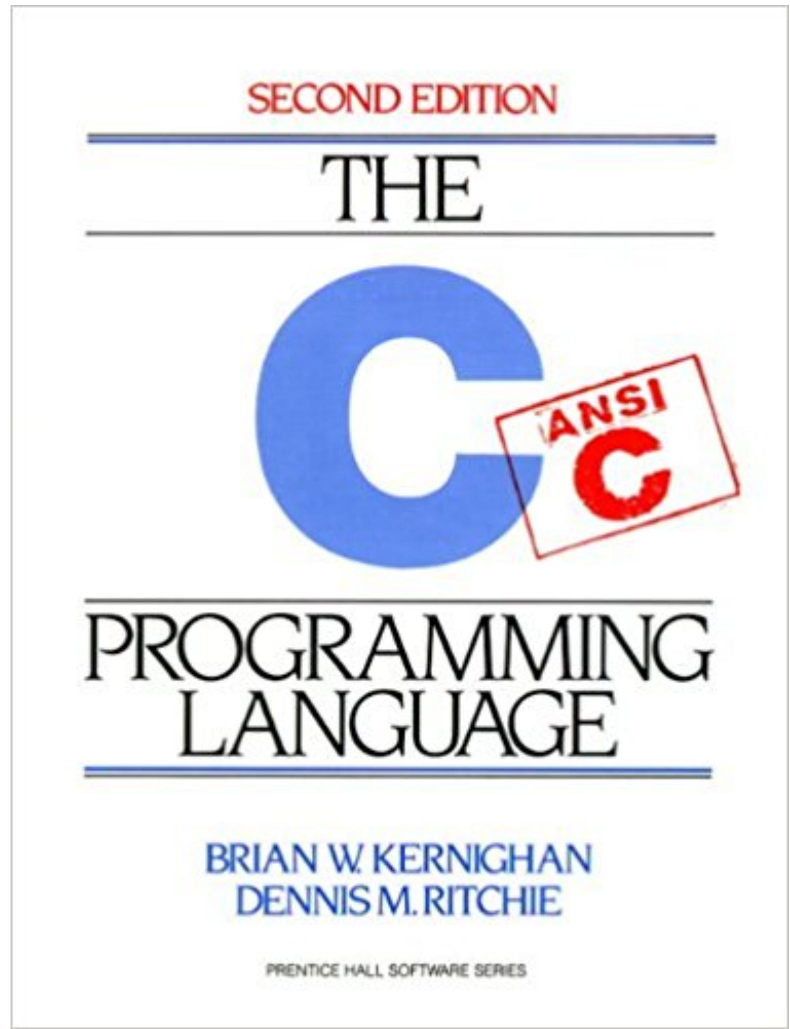
Think of how much testing this required!

Amazing what people can do even w/ weak languages!

Mid 60s: Ken Thompson and Dennis Ritchie get fed up hacking on the crummy code in MULTICS

Start writing UNIX for fun to get away from their bad code—First versions written in **assembly** in 1969

Writing in assembly is error-prone, so they created the C programming language—a derivative of BCPL (language around Bell labs at the time)



Early 70s: rewrite UNIX in C, create most famous operating system of all time

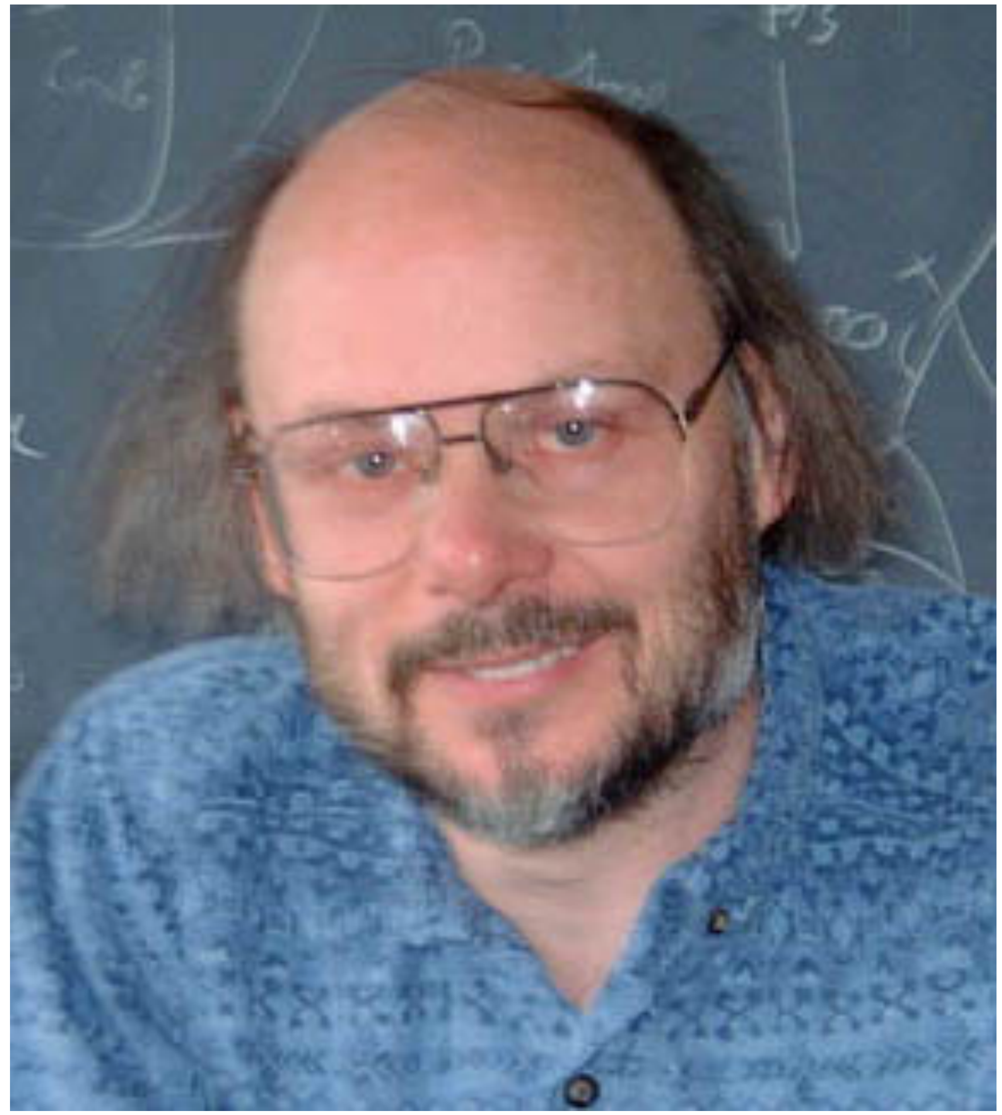
(My Mac's kernel is based on UNIX!)

Barbara Liskov

Early 70s: CLU—classes, abstract types, iterators

Liskov Substitution Principle: subtyping!

Bjarne Stroustrup



1979—Extended C to add classes, creates C++ (or C with classes)

And many others…!

Now, back to the lowest level…

Binary: The native language of the processor

- Modern processors are *very* fast
- (m/b)illions of *instructions* per sec

Processors execute a small number of *very basic* instructions

MOV r1, r2   ADD r1, r2,r3

IFZERO r1, +20

These instructions written in a binary encoding (**Why**?)

Binary: The native language of the processor

- Modern processors are *very* fast
- (m/b)illions of *instructions* per sec

Processors execute a small number of *very basic* instructions

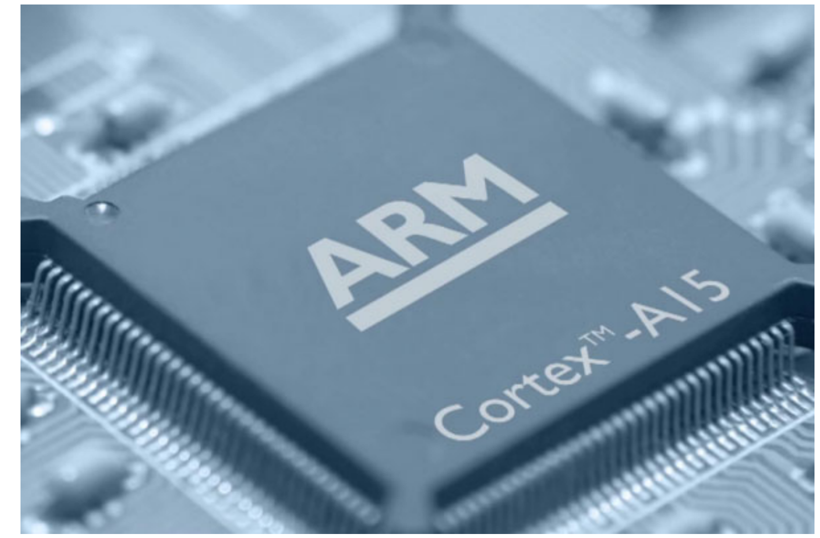
MOV r1, r2   ADD r1, r2,r3

IFZERO r1, +20

These instructions written in a binary encoding (Why?)

Compact representation   Quick to decode and execute

Thousands of different processors



Each speaks a different language

Called its *architecture*

Different versions of architecture add features, etc..

eclipse-workspace - sumnums/src/sumnums.cpp - Eclipse

factorial.cc | stringAnd_HOF_Examples.cc | main.cc | sumnums.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;

int sum(const unsigned int number) {
    int i = number;
    int accumulator = number;
    while (i > 0) {
        accumulator += i;
        i--;
    }
    return accumulator;
}

// This program accepts 1 argument
int main(int argc, char *argv[]) {
    int number;

    if (argc < 2) {
        cerr << "This program needs at least one argument.\n";
        exit(1);
    }

    try {
        number = stoi(argv[1]);
    } catch(const invalid_argument& ia) {
        cerr << "Invalid argument: " << ia.what() << '\n';
        exit(1);
    }

    if (number < 0) {
        cerr << "This program expects a non-negative argument.\n";
        exit(1);
    }

    cout << "I am going to sum the numbers from 0 to " << argv[0] << "\n";
    cout << "Sum: " << sum(number) << "\n";

    return 0;
}
```

Writable | Smart Insert | 14 :

So I need to turn this into something my i7 speaks…

To do that, I use a *compiler*

“Compile a file named sumnums.cpp, and output an executable file named sumnums”

```
clang++ sumnums.cpp -o sumnums
```

“Compile a file named sumnums.cpp, and output an executable file named sumnums”

clang++ sumnums.cpp -o sumnums

(*Ton* of options here, especially for large projects with complex configs / multifiles) eclipse-workspace - sumnums/src/sumnums.cpp - Eclipse

factorial.cc | stringAnd_HOF_Examples.cc | main.cc | sumnums.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;

int sum(const unsigned int number) {
    int i = number;
    int accumulator = number;
    while (i > 0) {
        accumulator += i;
        i--;
    }
    return accumulator;
}

// This program accepts 1 argument
int main(int argc, char *argv[]) {
    int number;

    if (argc < 2) {
        cerr << "This program needs at least one argument.\n";
        exit(1);
    }

    try {
        number = stoi(argv[1]);
    } catch(const invalid_argument& ia) {
        cerr << "Invalid argument: " << ia.what() << '\n';
        exit(1);
    }

    if (number < 0) {
        cerr << "This program expects a non-negative argument.\n";
        exit(1);
    }

    cout << "I am going to sum the numbers from 0 to " << argv[0] << "\n";
    cout << "Sum: " << sum(number) << "\n";

    return 0;
}
```

Compiler


```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00000000: cffa edfe 0700 0001 0300 0000 0100 0000  ................
00000010: 0400 0000 5002 0000 0020 0000 0000 0000  ....P.... ......
00000020: 1900 0000 d801 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 6812 0000 0000 0000 7002 0000 0000 0000  h.......p.......
00000050: 6812 0000 0000 0000 0700 0000 0700 0000  h...............
00000060: 0500 0000 0000 0000 5f5f 7465 7874 0000  ........__text..
00000070: 0000 0000 0000 0000 5f5f 5445 5854 0000  ........__TEXT..
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0b0e 0000 0000 0000 7002 0000 0400 0000  ........p.......
000000a0: d814 0000 3a00 0000 0004 0080 0000 0000  ....:...........
000000b0: 0000 0000 0000 0000 5f5f 6763 635f 6578  ........__gcc_ex
000000c0: 6365 7074 5f74 6162 5f5f 5445 5854 0000  cept_tab__TEXT..
000000d0: 0000 0000 0000 0000 0c0e 0000 0000 0000  ................
000000e0: 3c01 0000 0000 0000 7c10 0000 0200 0000  <.......|.......
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 0000 0000 5f5f 6373 7472 696e  ........__cstrin
00000110: 6700 0000 0000 0000 5f5f 5445 5854 0000  g.......__TEXT..
00000120: 0000 0000 0000 0000 480f 0000 0000 0000  ........H.......
00000130: 8b00 0000 0000 0000 b811 0000 0000 0000  ................
00000140: 0000 0000 0000 0000 0200 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 5f5f 636f 6d70 6163  ........__compac
00000160: 745f 756e 7769 6e64 5f5f 4c44 0000 0000  t_unwind__LD....
00000170: 0000 0000 0000 0000 d80f 0000 0000 0000  ................
00000180: 0001 0000 0000 0000 4812 0000 0300 0000  ........H.......
00000190: a816 0000 0e00 0000 0000 0002 0000 0000  ................
000001a0: 0000 0000 0000 0000 5f5f 6568 5f66 7261  ........__eh_fra
000001b0: 6d65 0000 0000 0000 5f5f 5445 5854 0000  me......__TEXT..
000001c0: 0000 0000 0000 0000 d810 0000 0000 0000  ................
000001d0: 9001 0000 0000 0000 4813 0000 0300 0000  ........H.......
000001e0: 1817 0000 0100 0000 0b00 0068 0000 0000  ...........h....
000001f0: 0000 0000 0000 0000 2400 0000 1000 0000  ........$.......
00000200: 000c 0a00 0000 0000 0200 0000 1800 0000  ................
00000210: 2017 0000 2300 0000 5019 0000 2005 0000   ...#...P... ...
00000220: 0b00 0000 5000 0000 0000 0000 0300 0000  ....P...........
00000230: 0300 0000 0900 0000 0c00 0000 1700 0000  ................
00000240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000250: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000260: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000270: 5548 89e5 897d fc8b 7dfc 897d f88b 7dfc  UH...}..}..}..}.
00000280: 897d f483 7df8 000f 8d17 0000 008b 45f8  .}..}.........E.
00000290: 0345 f489 45f4 8b45 f883 c0ff 8945 f8e9  .E..E..E.....E..
000002a0: dfff ffff 8b45 f45d c30f 1f80 0000 0000  .....E.]........
000002b0: 5548 89e5 4881 ecd0 0000 00c7 45bc 0000  UH..H.......E...
000002c0: 0000 897d b848 8975 b083 7db8 010f 8d24  ...}.H.u..}....$
000002d0: 0000 0048 8b3d 0000 0000 488d 35d7 0e00  ...H.=....H.5...
000002e0: 00e8 0000 0000 bf01 0000 0048 8985 78ff  ...........H..x.
000002f0: ffff e800 0000 0048 8b45 b048 8b00 488d  .......H.E.H..H.
00000300: 4d90 4889 4dc8 4889 45c0 488b 55c8 4889  M.H.M.H.E.H.U.H.
00000310: 55d8 4889 45d0 488b 45d8 4889 45e0 4889  U.H.E.H.E.H.E.H.
00000320: 45e8 4889 45f0 4889 45f8 48c7 4010 0000  E.H.E.H.E.H.@...
00000330: 0000 48c7 4008 0000 0000 48c7 0000 0000  ..H.@.....H.....
00000340: 0048 8b55 d048 89d7 4889 8d70 ffff ff48  .H.U.H..H..p...H
00000350: 8985 68ff ffff 4889 9560 ffff ffe8 0000  ..h...H..`......
00000360: 0000 488b bd68 ffff ff48 8bb5 60ff ffff  ..H..h...H..`...
00000370: 4889 c2e8 0000 0000 4531 c044 89c6 ba0a  H.......E1.D....
00000380: 0000 0048 8bbd 70ff ffff e800 0000 0089  ...H..p.........
00000390: 855c ffff ffe9 0000 0000 488d 7d90 e800  .\........H.}...
000003a0: 0000 008b 855c ffff ff89 45ac 837d ac00  .....\....E..}..
000003b0: 0f8d 4000 0000 488b 3d00 0000 0048 8d35  ..@...H.=....H.5
000003c0: 1f0e 0000 e800 0000 00bf 0100 0000 4889  ..............H.
000003d0: 8550 ffff ffe8 0000 0000 89d1 4889 4588  .P..........H.E.
000003e0: 894d 8448 8d7d 90e8 0000 0000 e900 0000  .M.H.}..........
000003f0: 00e9 8b00 0000 488b 3d00 0000 0048 8d35  ......H.=....H.5
00000400: 0e0e 0000 e800 0000 0048 8b75 b048 8b36  .........H.u.H.6
00000410: 4889 c7e8 0000 0000 488d 351c 0e00 0048  H.......H.5....H
00000420: 89c7 e800 0000 0048 8b3d 0000 0000 488d  .......H.=....H.
00000430: 3508 0e00 0048 8985 48ff ffff e800 0000  5....H..H.......
00000440: 008b 7dac 4889 8540 ffff ffe8 0000 0000  ..}.H..@........
00000450: 488b bd40 ffff ff89 c6e8 0000 0000 488d  H..@..........H.
00000460: 35d6 0d00 0048 89c7 e800 0000 0031 c948  5....H.......1.H
-UU=:----F1  sumnums.o     Top L1     (Hexl company) ------
```

So, the compiler turns C++ into a giant list of these instructions…

So, the compiler turns C++ into a giant list of these instructions…

These are written in *assembly*
(Human-readable binary)

Let’s see what assembly the compiler generates…

```
clang++ -S sumnums.cpp
```

(Note I really used:
clang++ -S sumnums -fno-asynchronous-unwind-tables
This is because otherwise extra debugging overhead is inserted.)

```
	.section	__TEXT,__text,regular,pure_instructions
	.macosx_version_min 10, 12
	.globl	__Z3sumj
	.p2align	4, 0x90
__Z3sumj:                               ## @_Z3sumj
## BB#0:
	pushq	%rbp
	movq	%rsp, %rbp
	movl	%edi, -4(%rbp)
	movl	-4(%rbp), %edi
	movl	%edi, -8(%rbp)
	movl	-4(%rbp), %edi
	movl	%edi, -12(%rbp)
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
	cmpl	$0, -8(%rbp)
	jle	LBB0_3
## BB#2:                                ##   in Loop: Header=BB0_1 Depth=1
	movl	-8(%rbp), %eax
	addl	-12(%rbp), %eax
	movl	%eax, -12(%rbp)
	movl	-8(%rbp), %eax
	addl	$-1, %eax
	movl	%eax, -8(%rbp)
	jmp	LBB0_1
LBB0_3:
	movl	-12(%rbp), %eax
	popq	%rbp
	retq

	.globl	_main
	.p2align	4, 0x90
_main:                                  ## @main
Lfunc_begin0:
	.cfi_startproc
	.cfi_personality 155, ___gxx_personality_v0
	.cfi_lsda 16, Lexception0
## BB#0:
	pushq	%rbp
Ltmp24:
	.cfi_def_cfa_offset 16
Ltmp25:
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
Ltmp26:
	.cfi_def_cfa_register %rbp
	subq	$240, %rsp
	movl	$0, -68(%rbp)
	movl	%edi, -72(%rbp)
	movq	%rsi, -80(%rbp)
	cmpl	$2, -72(%rbp)
	jge	LBB1_2
## BB#1:
```

Divided up by function

```
	.section	__TEXT,__text,regular,pure_instructions
	.macosx_version_min 10, 12
	.globl	__Z3sumj
	.p2align	4, 0x90
__Z3sumj:                               ## @_Z3sumj
## BB#0:
	pushq	%rbp
	movq	%rsp, %rbp
	movl	%edi, -4(%rbp)
	movl	-4(%rbp), %edi
	movl	%edi, -8(%rbp)
	movl	-4(%rbp), %edi
	movl	%edi, -12(%rbp)
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
	cmpl	$0, -8(%rbp)
	jle	LBB0_3
## BB#2:                                ##   in Loop: Header=BB0_1 Depth=1
	movl	-8(%rbp), %eax
	addl	-12(%rbp), %eax
	movl	%eax, -12(%rbp)
	movl	-8(%rbp), %eax
	addl	$-1, %eax
	movl	%eax, -8(%rbp)
	jmp	LBB0_1
LBB0_3:
	movl	-12(%rbp), %eax
	popq	%rbp
	retq

	.globl	_main
	.p2align	4, 0x90
_main:                                  ## @main
Lfunc_begin0:
	.cfi_startproc
	.cfi_personality 155, ___gxx_personality_v0
	.cfi_lsda 16, Lexception0
## BB#0:
	pushq	%rbp
Ltmp24:
	.cfi_def_cfa_offset 16
Ltmp25:
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
Ltmp26:
	.cfi_def_cfa_register %rbp
	subq	$240, %rsp
	movl	$0, -68(%rbp)
	movl	%edi, -72(%rbp)
	movq	%rsi, -80(%rbp)
	cmpl	$2, -72(%rbp)
	jge	LBB1_2
## BB#1:
```

Divided up by function

Implementation of sum

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .globl  __Z3sumj
        .p2align        4, 0x90
__Z3sumj:                               ## @_Z3sumj
## BB#0:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edi
        movl    %edi, -8(%rbp)
        movl    -4(%rbp), %edi
        movl    %edi, -12(%rbp)
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
        cmpl    $0, -8(%rbp)
        jle     LBB0_3
## BB#2:                                ##   in Loop: Header=BB0_1 Depth=1
        movl    -8(%rbp), %eax
        addl    -12(%rbp), %eax
        movl    %eax, -12(%rbp)
        movl    -8(%rbp), %eax
        addl    $-1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_3:
        movl    -12(%rbp), %eax
        popq    %rbp
        retq

        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
Lfunc_begin0:
        .cfi_startproc
        .cfi_personality 155, ___gxx_personality_v0
        .cfi_lsda 16, Lexception0
## BB#0:
        pushq   %rbp
Ltmp24:
        .cfi_def_cfa_offset 16
Ltmp25:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp26:
        .cfi_def_cfa_register %rbp
        subq    $240, %rsp
        movl    $0, -68(%rbp)
        movl    %edi, -72(%rbp)
        movq    %rsi, -80(%rbp)
        cmpl    $2, -72(%rbp)
        jge     LBB1_2
## BB#1:
```

Don't worry that this code is hard to understand for now

Divided up by function

Implementation of main

(It also confuses me..)

I can manually transform the assembly to the binary…

```
as sumnums.s
```

```
Kyles-MacBook-Pro-2:src micinski$
Kyles-MacBook-Pro-2:src micinski$ ./sumnums.o
-bash: ./sumnums.o: cannot execute binary file
Kyles-MacBook-Pro-2:src micinski$ 
```

Crud...

For example: code to print to the screen

Insight: my program needs a lot of other stuff to run...

This is kept in a *library*

(But keep in mind, that's also **just code**. Nothing particularly magical)

Your code

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .globl  __Z3sumj
        .p2align        4, 0x90
__Z3sumj:                               ## @_Z3sumj
## BB#0:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edi
        movl    %edi, -8(%rbp)
        movl    -4(%rbp), %edi
        movl    %edi, -12(%rbp)
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
        cmpl    $0, -8(%rbp)
        jle     LBB0_3
## BB#2:                                ##   in Loop: Header=BB0_1 Depth=1
        movl    -8(%rbp), %eax
        addl    -12(%rbp), %eax
        movl    %eax, -12(%rbp)
        movl    -8(%rbp), %eax
        addl    $-1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_3:
        movl    -12(%rbp), %eax
        popq    %rbp
        retq

        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
Lfunc_begin0:
        .cfi_startproc
        .cfi_personality 155, ___gxx_personality_v0
        .cfi_lsda 16, Lexception0
## BB#0:
        pushq   %rbp
Ltmp24:
        .cfi_def_cfa_offset 16
Ltmp25:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp26:
        .cfi_def_cfa_register %rbp
        subq    $240, %rsp
        movl    $0, -68(%rbp)
        movl    %edi, -72(%rbp)
        movq    %rsi, -80(%rbp)
        cmpl    $2, -72(%rbp)
        jge     LBB1_2
## BB#1:
```

+

lstdc++

lm

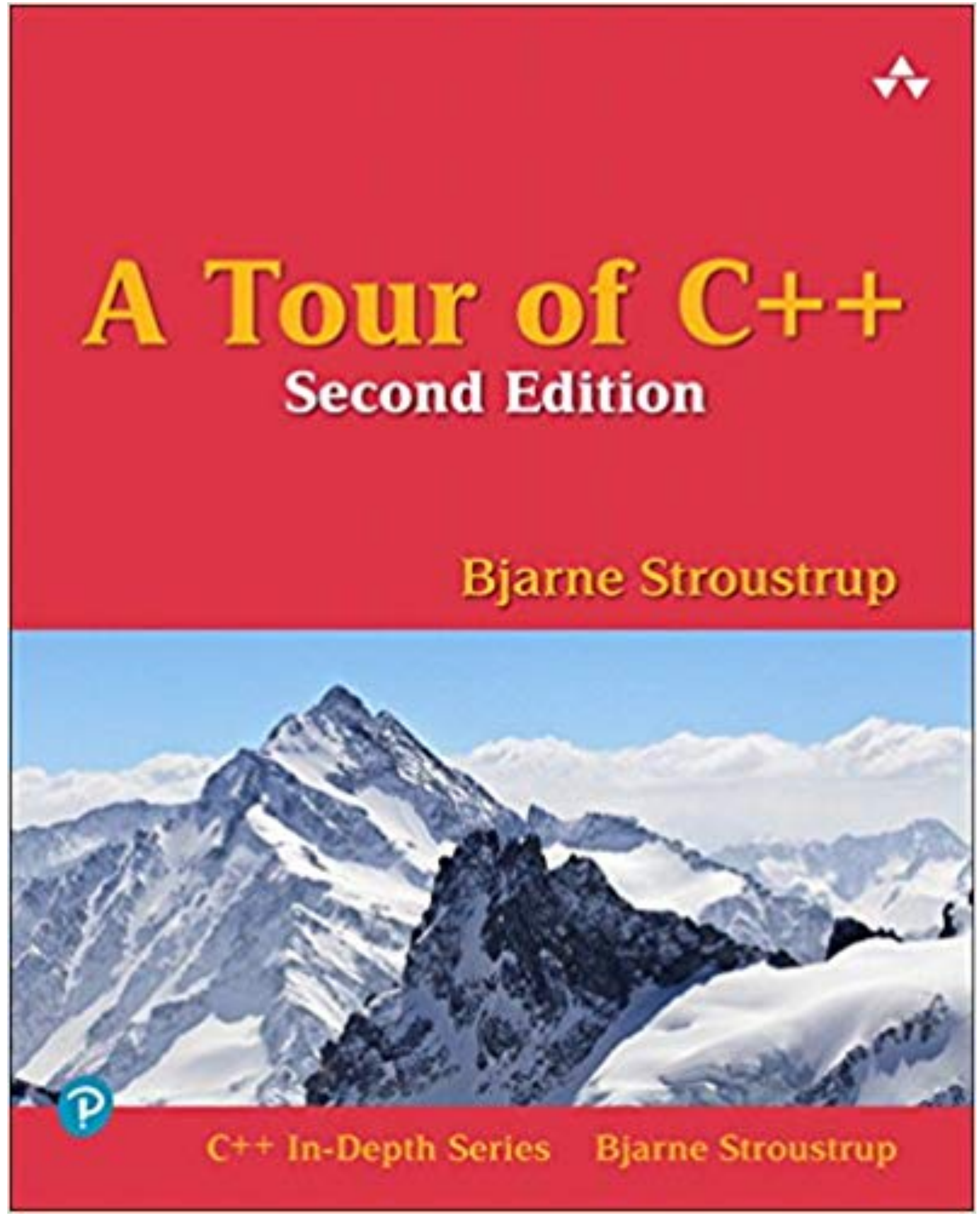
etc…

Executable file

=

Syllabus

http://kmicinski.com/cs245/syllabus/



A draft is freely available at: isocpp.org/tour

Grade breakdown

50% : ~8 coding projects

10% : weekly labs

35% : Two midterms (~6 weeks in and ~12 weeks in)

5% : Participation (graded in various ways)

Autograder

Academic honesty

All submissions are graded using Clang 5, Racket 7, Python 3.7 on an Ubuntu 18.04 LTR server.

If you have any trouble configuring this (or a compatible environment) on your home machine, I highly recommend you develop with:

lubuntu

+

VirtualBox

What programming paradigms have you heard of?

See if you know (or know of) any that your neighbors don't—or vice versa.

Programming languages: **paradigms**

- ***Imperative languages*** emphasize issuing commands that tell the machine what ***to do next*** at each step of evaluation.
- ***Structured languages*** emphasize structured control-flow (i.e., not `goto`) that can be properly nested, especially sequencing, conditionals, and looping constructs (while, for, do).
- ***Procedural programming*** is imperative programming with subroutines —emphasizes abstracting behaviors over data.
- ***Object-oriented programming*** emphasizes encapsulation of behaviors (methods) and data (fields) within classes, abstract modular schema for program values, that are instantiated as objects at run-time. Inheritance hierarchies used to promote code-reuse.
- ***Reactive programming*** emphasizes responding to events.

Programming languages: **paradigms**

- ***Dynamic languages*** emphasize permitting arbitrary manipulation of program values, control, and the environment at runtime. Primarily these use duck typing / structural typing. A related paradigm is that of ***reflective*** programming—dynamically modifying types at runtime.

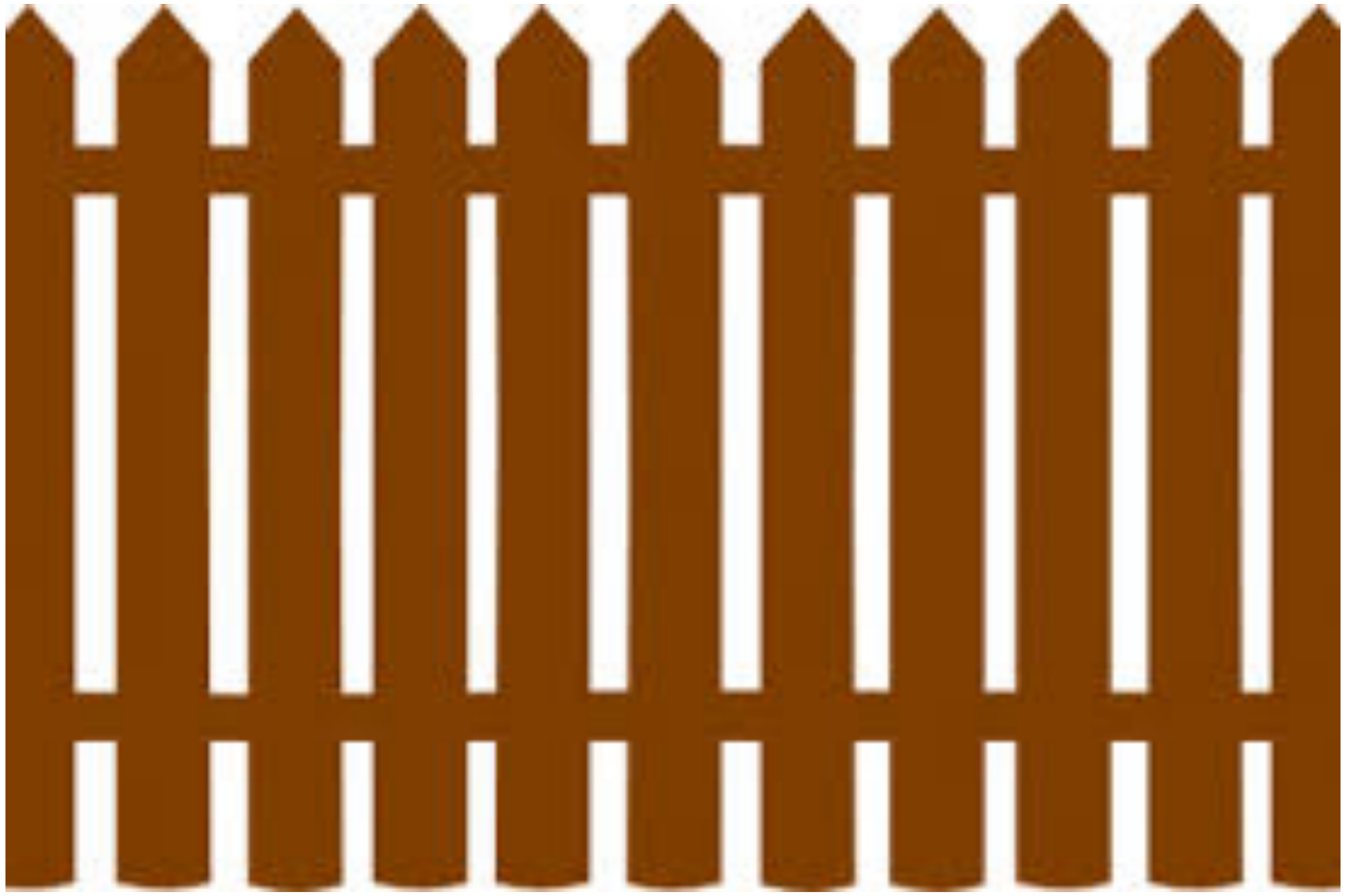
- ***Static languages*** emphasize bounding program behavior ahead-of-time. Primarily these use nominal typing and are type-checked.

- ***Array languages*** emphasize concisely manipulating arrays, matrices.

- ***Functional programming*** emphasizes immutability, like math. Programs are constructed from pipelines of composed functions that transform inputs to outputs without affecting their environment.

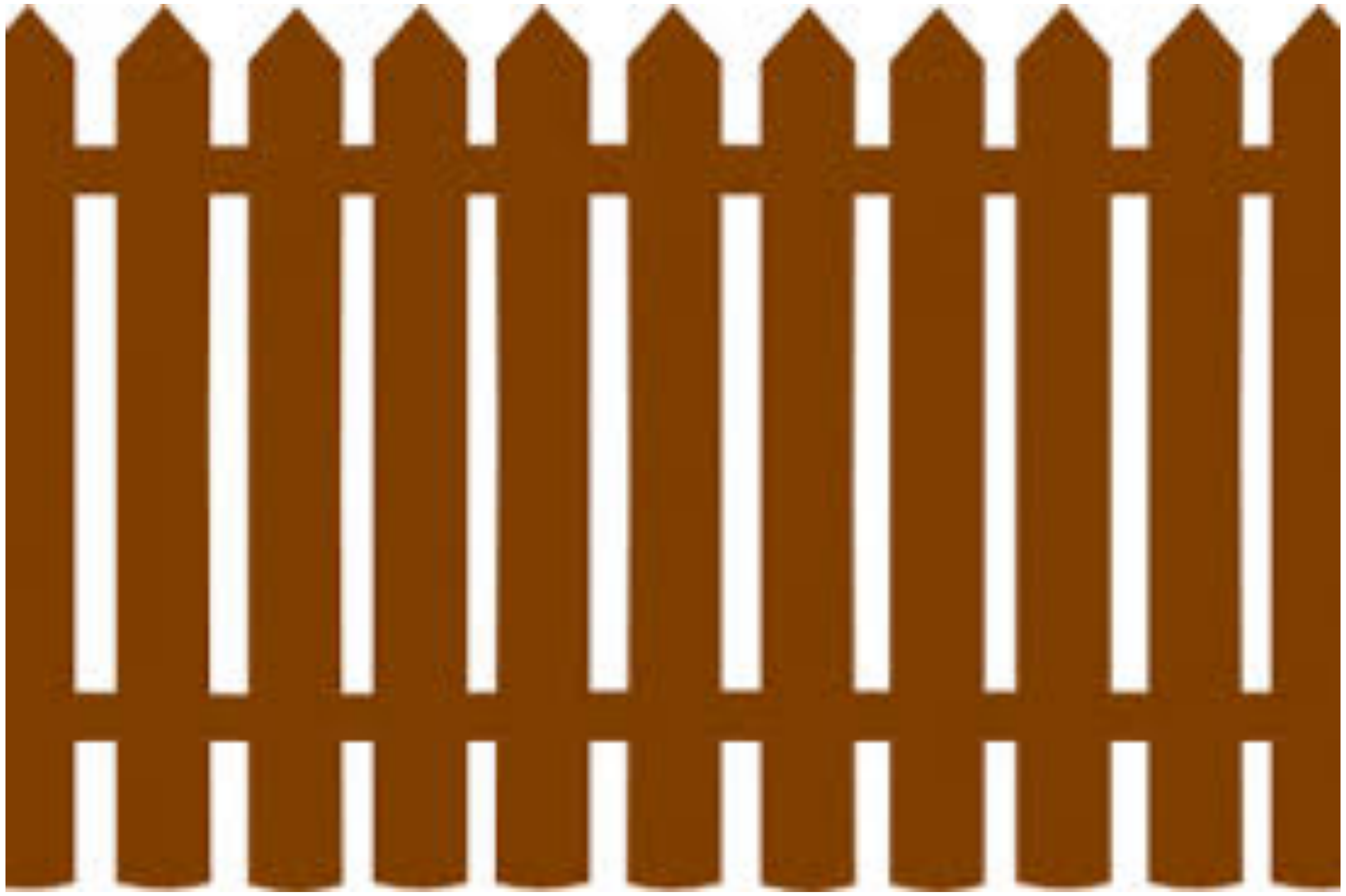
- ***Logic programming*** emphasizes declarations, propositions, logical constraints. The programmer states what must be true of a solution.

Programming languages: **imperative paradigm**



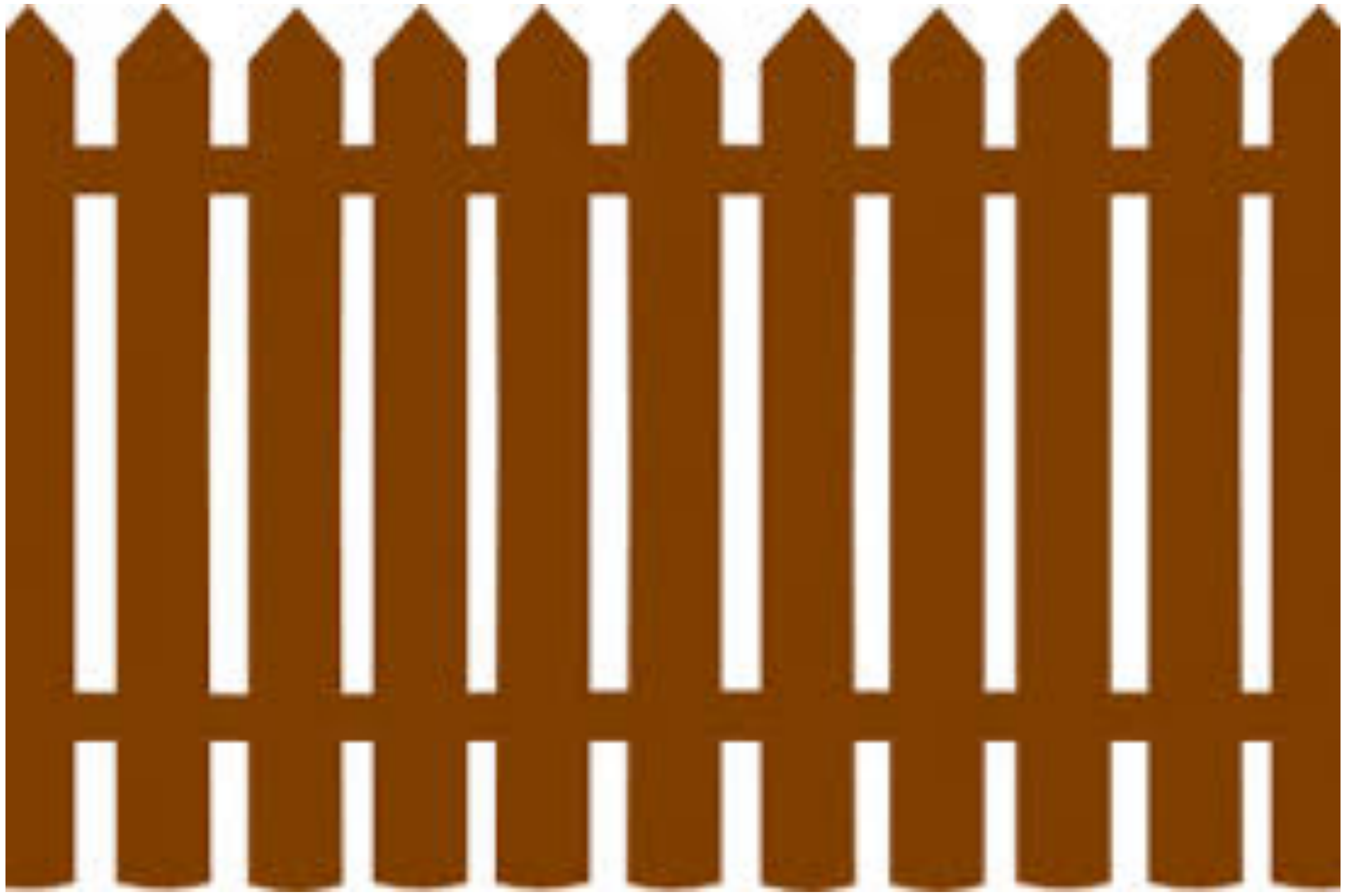
```
Place first board and rails
While fence incomplete:
    move half-a-foot to the left
    position a new board
    position a nail
    hammer nail into top rail
    ...
```

Programming languages: **functional paradigm**



```
function build_fence(len):
    if len == 1:
        return rails_and_first_picket()
    else:
        return add_one_picket(build_fence(len-1))
```

Programming languages: **logical paradigm**



```
def fence.
fence is 5 ft tall.
fence has two rails.
fence has 50 pickets,
    each picket is 4” wide
    every picket is 2” from at least one other.
```

C++ is a superset of C with
object-oriented features and generics/templates.

**Focusing on classic/vanilla C++
written from scratch...**

C/C++

C/C++ is an example of the *imperative*, *structured*, *procedural*, *static*, and *object-oriented* language paradigms.

Introduction to C++ ***syntax*** and ***semantics***

C/C++

The ***syntax*** of a language is the rules one must follow for a program to be parsed correctly. E.g., braces must match {…}, identifiers begin with a character in [_A-Za-z], semi-colons, etc.

The ***semantics*** of a language is the rules by which programs are run or evaluated to a result or behavior. E.g., operator precedence, order of operations, dynamic dispatch (which method is it), etc.

C++ syntax: **comments**

```
/* Multi-line or “C style” comments begin with a slash-star
 *************************************
          ...and end with star-slash */


// Single-line or “C++ style” comments start with two slashes
// and end with a newline


/* Multi-line comments cannot be nested

 …like this: /*    */  // <- this closes the whole comment

*/ // <- this dangles
```

C++ syntax: **identifiers, strings, numbers**

(The basics are very similar to Java, as Java was designed to have C-like syntax.)

IDs match [_a-zA-Z][_a-zA-Z0-9]*, and are not reserved keywords

```
x

_0123

A_0

a12
```

Numbers can take a number of forms in C/C++… e.g.

```
2.0, 2f

0xffff00ff

30500ULL
```

Characters are between single-quotes: e.g., ’a’, ‘\n’
Strings are between double-quotes: “Hello World\n”
Strings in C/C++ are just arrays of chars: e.g., char[16]

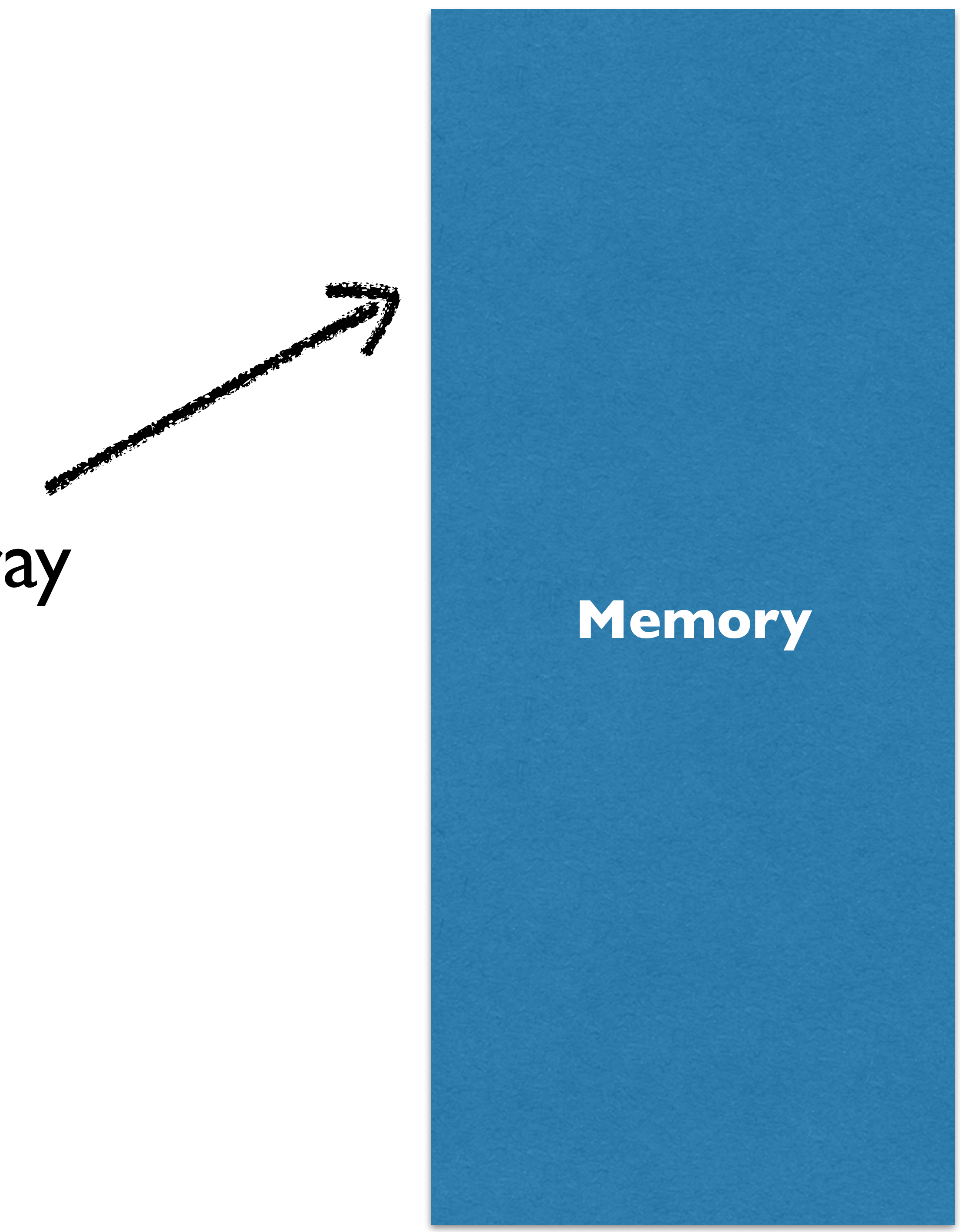
C++ syntax: **reserved keywords**

```
alignas       constexpr     inline            short
alignof       const_cast    int               signed
and           continue      long              sizeof
and_eq        decltype      mutable           static
asm           default       namespace         static_assert
auto(1)       delete        new               static_cast
bitand        do            noexcept          struct
bitor         double        not               switch
bool          dynamic_cast  not_eq            synchronized
break         else          nullptr           template
case          enum          operator          this
catch         explicit      or                thread_local
char          export        or_eq             throw
char8_t       extern        private           true
char16_t      false         protected         try           virtual
char32_t      float         public            typedef       void
class(1)      for           reflexpr          typeid        volatile
compl         friend        register          typename      wchar_t
concept       goto          reinterpret_cast  union         while
const         if            requires          unsigned      xor
consteval     import        return            using         xor_eq
```

C++ semantics: **memory model**

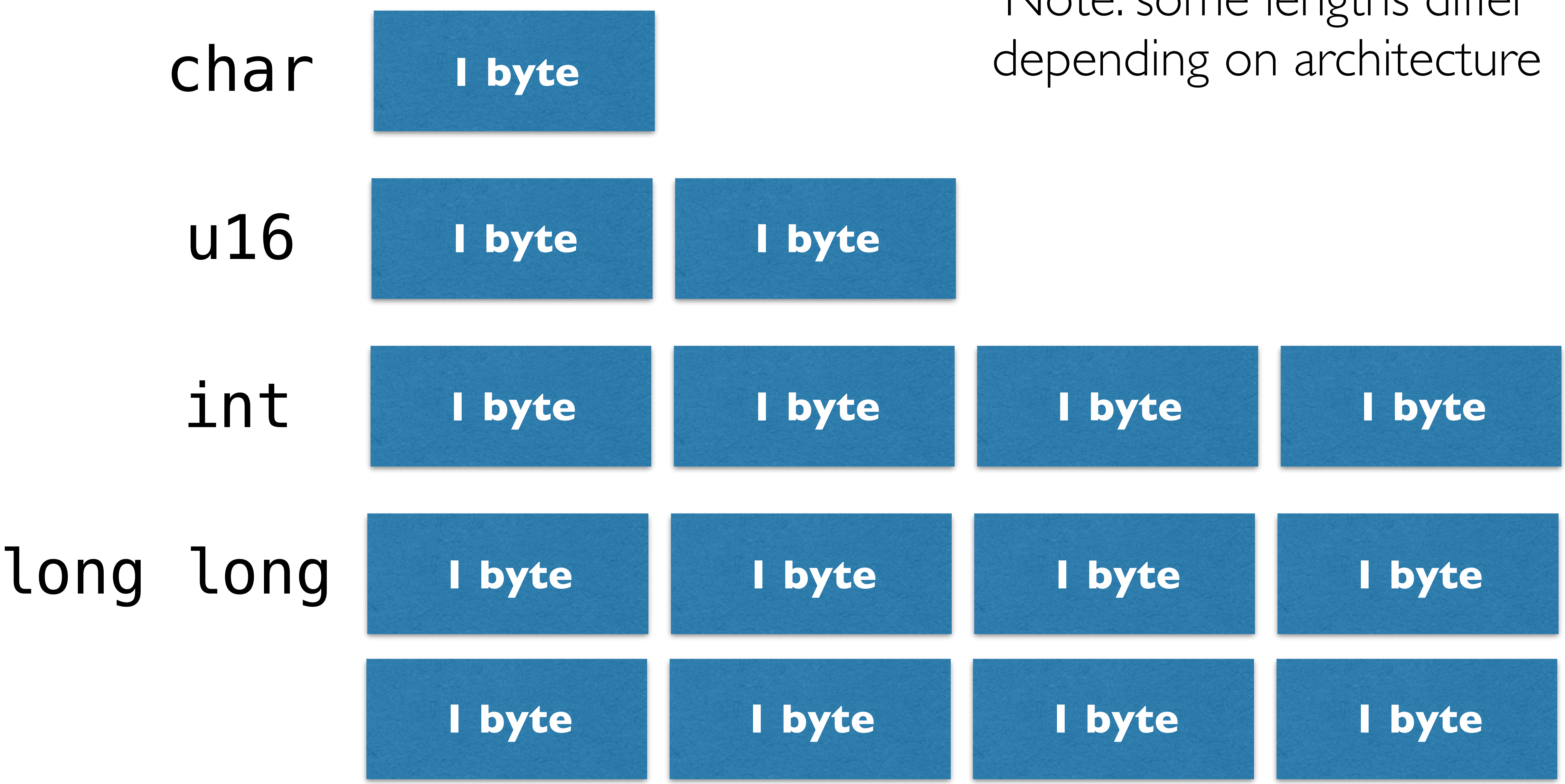
Each variable in C++ exists somewhere in memory

C++ thinks of this as a giant array of bytes



C++ semantics: **memory model**

Note: some lengths differ depending on architecture



(for a 32-bit architecture…)

C++ syntax: **includes and macros**

By convention, .cpp files are used for source, .h for libraries/declarations.

```
#include “path/to/file.h”
```

#include will textually replace this line with the entire contents of a file.

```
#include <library>
```

#define defines a macro: in this case,
textually replace occurrences of “MAX” with “255”.

```
#define MAX 255
```

C++ syntax: **anatomy of a function**

The smallest valid C program.

```
int main()
{
    return 0;
}
```

C++ syntax: **anatomy of a function**

The smallest valid C program.

main(...) is the entry-point of the program

Returns status code 0, success.

```
int main()
{
    return 0;
}
```

All statements end with a semi-colon, as in Java.

In C/C++ the preferred style is for curly braces to line up on the same row or column. As in Java though, whitespace only separates tokens and is not otherwise meaningful.

C++ syntax: **anatomy of a function**

“Hello World”

```
#include <iostream>

int main()
{
    std::cout << “Hello World”
              << std::endl;
    return 0;
}
```

Clang++: **compiling and running**

"Hello World"

```
$ clang++ -o hello hello.cpp
$ ls
hello    hello.cpp
$ ./hello
Hello World
$
```

-g compiles for debugging,
-std=c++14 compiles with c++14 features
-O2 compiles with optimization level 2

C++ syntax: **arrays, dereferencing a pointer**

An array (len=5) can be allocated on the stack using syntax T a[5];
or on the heap using syntax T* a = new T[5];

Using the prefix, unary operator * will explicitly dereference a pointer.
if `a` is of type `int*`, then `*a` is of type `int`.

```
int main()
{
    int* iarr = new int[5];
    *iarr = 99;
    // is the same as
    iarr[0] = 99;
    // ...
}
```

C++ syntax: **structs**

A custom type containing two publicly visible fields: x, and y.

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

int main()
{
   Point p;
   p.x = 5; // field access
   //...
}
```

C++ syntax: **new and delete**

keyword "new" allocates an object on the heap, "delete" frees it

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

int main()
{
   Point* p = new Point();
   p->x = 5; // Same as (*p).x = 5
   delete p;
   //...
}
```

C++ syntax: **pass by reference**

Using T& in place of T* means the pointer itself cannot be manipulated and dereference is implicit! These are called ***references.***

```
bool x_gt_y(const P& p)
{
    return p.x > p.y;
}
```

C++ semantics: **reading command-line arguments**

Give main arguments argc and argv as below.

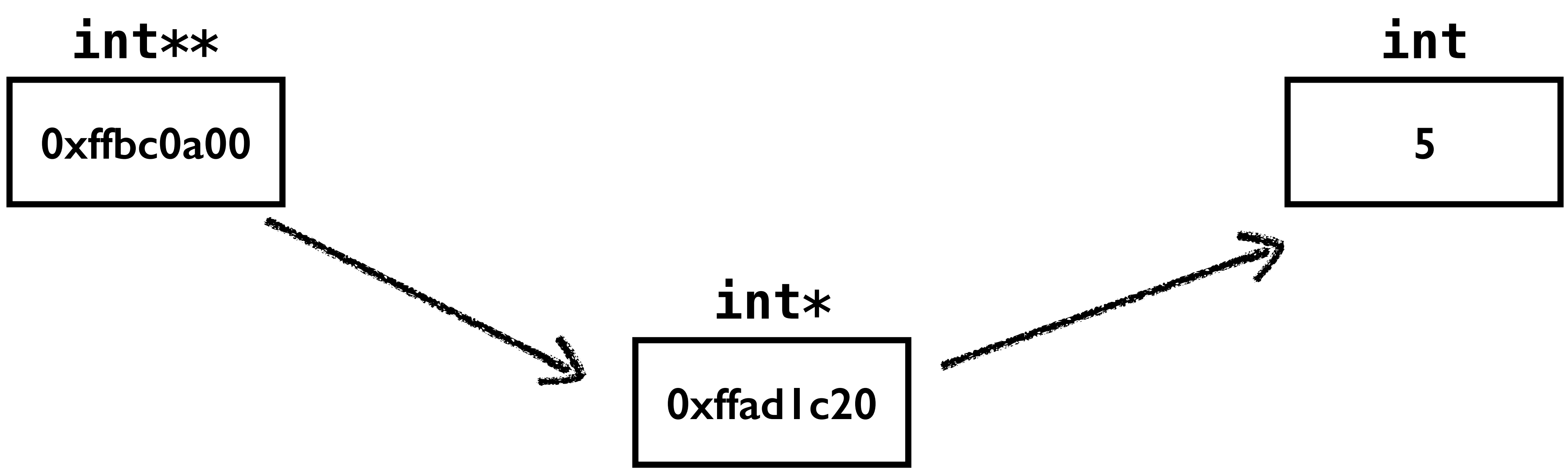
```
#include <iostream>

int main(int argc, const char** argv)
{
    if (argc <= 1) return 1; // failure
    std::cout << argv[1]
              << std::endl;
    return 0; // success
}
```

C++ semantics: **pointers**

A type T∗ means a pointer to something of type T.

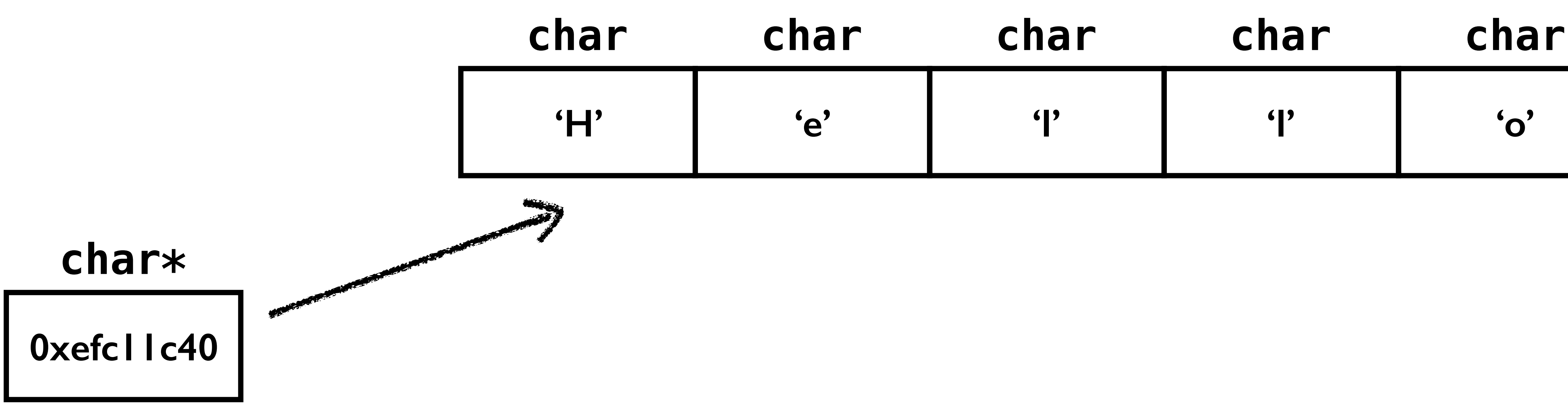
For example, an `int*` is a word of memory containing the location, in memory, of an integer. An `int**` is an address pointing to a location containing an address to an integer.



C++ semantics: **pointers**

Pointers in C do not have lengths. You can read as many words or bytes at the location as you wish. Thus pointers are all really arrays of length 1 or greater.

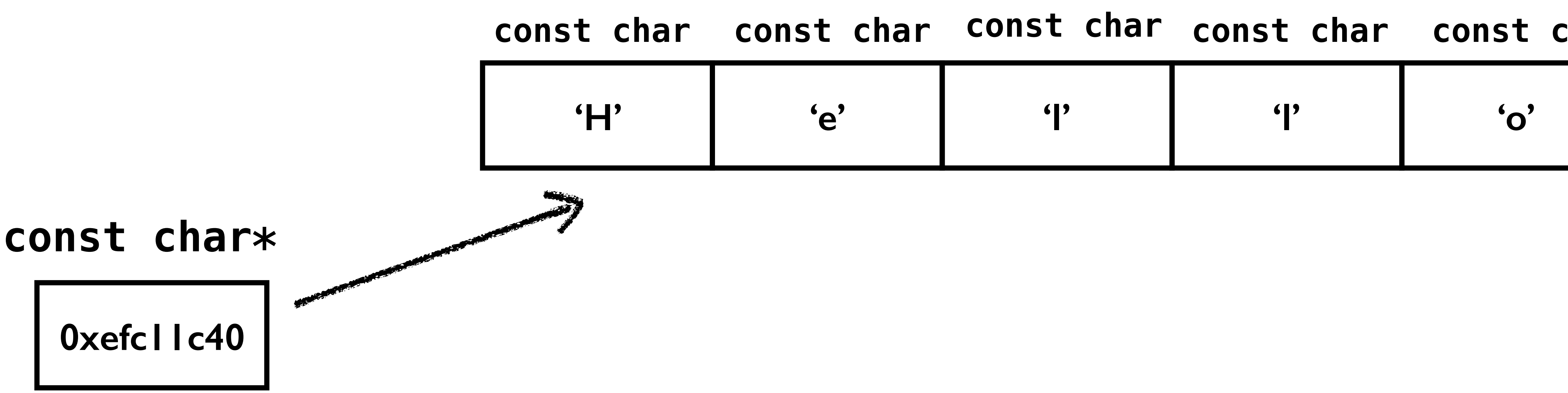
A string in C is just an array of chars, or a `char*`



C++ semantics: **const pointers**

A type may be preceded by keyword const, this tells the compiler to check that the value cannot be modified!

A const string in C is a `const char*`



If the pointer itself is also const, then it is a `const char* const`

Let's try out some examples