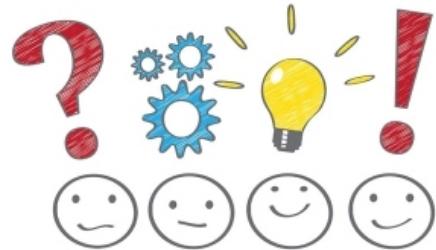


# Admin

*First lab and assign  
We're off and running!*

*Monday holiday  
Vote in piazza poll  
to schedule makeup*

*Check in*



## Today: From Assembly to C (and back again)

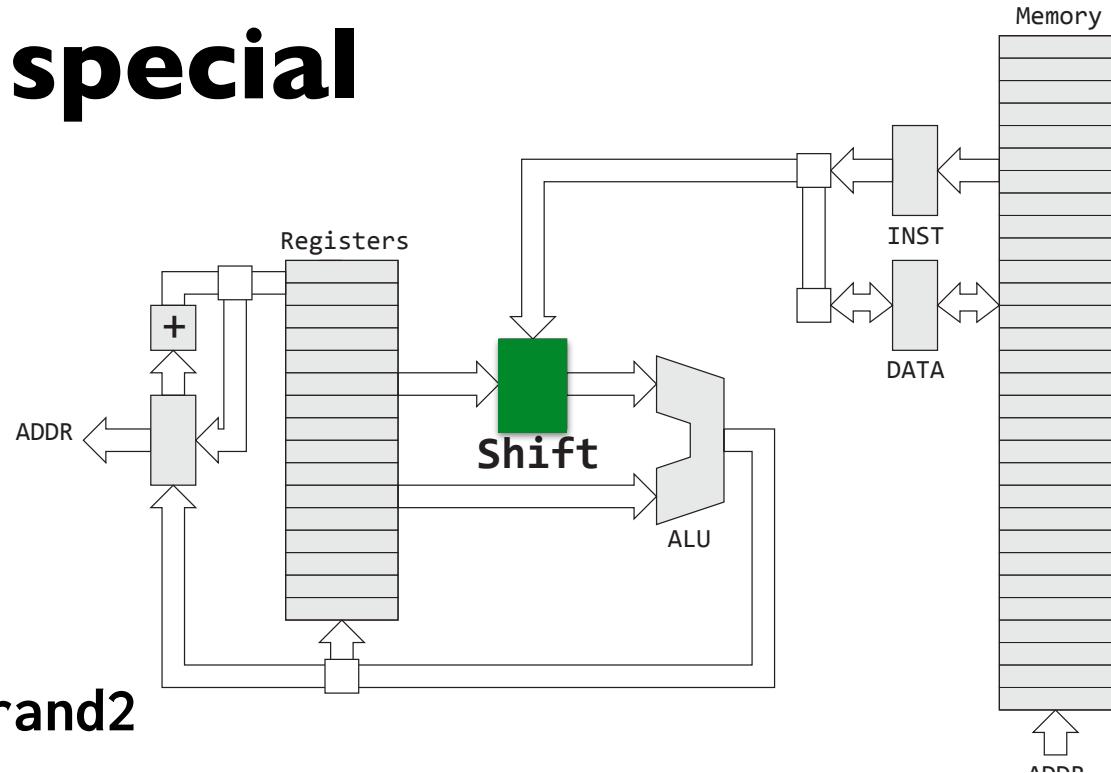
ARM condition codes, branch instructions

C language as “high-level” assembly

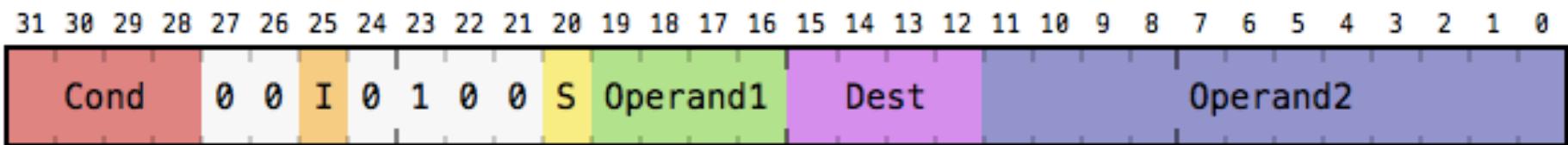
What does a compiler do?

Makefiles

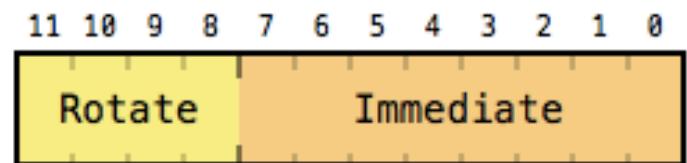
# Operand 2 is special



Dest = Operand1 op Operand2



**add r0, r1, #0x1f000**  
**sub r0, r1, #6**  
**rsb r0, r1, #6**  
**add r0, r1, r2, lsl #3**  
**mov r1, r2, ror #7**



lsl, lsr, asr, ror

# Control flow

Instructions stored in contiguous memory

Register pc (r15) tracks address in memory where instructions are being read

Default is "straight-line" code: next instruction to execute is at next higher memory address ( $pc = pc + 4$ )

Change pc to change which instruction is next, branch instruction has effect of  $pc = \text{target}$

**b target**

Above branch is unconditional (always taken), alternative is predicated on state of "condition codes"

# Condition Codes

Z result was 0

N result was < 0

C operation generated carry

V operation had arithmetic overflow

(More on carry and overflow in later lecture...)

## Which instructions set/clear codes?

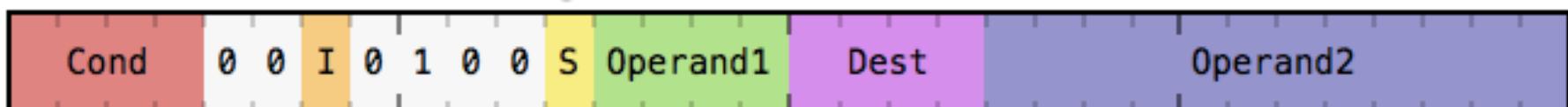
cmp (like sub, but discards result)

tst (like and, but discards result)

Any data processing instruction suffixed with s:

adds movs orrs lsrs ...

s bit  
(if on, instr will set condition codes)



<b>Code</b>	<b>Suffix</b>	<b>Description</b>	<b>Flags</b>
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

# Branch instructions

**b target**

**bne target**

**bmi target**

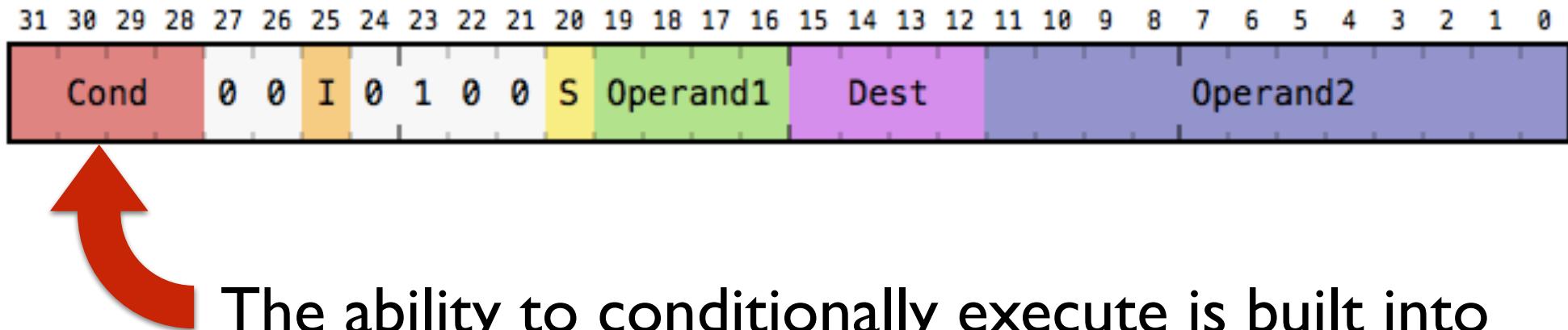
**bge target**

...

**branch** reads condition codes, as set by a previous instruction

If specific condition satisfied, branch is taken,  $pc = \text{target}$   
otherwise falls through,  $pc = pc + 4$

# Condition execution



```
addeq r0, r0, #3  
submi r1, r2, r3
```

*Q: Given our earlier foray into machine-encoded instructions, what do you suspect is the condition represented by 0xe?*

# Challenge for you all:

**Write an assembly program to count the "on" bits in a given numeric value**

```
mov r0, #val  
mov r1, #0  
  
// r0 holds input value  
// use r1 to store count of on bits in value
```

# VisUAL ARM Emulator

The screenshot shows the VisUAL ARM Emulator interface. At the top, there is a toolbar with buttons for New, Open, Save, Settings, Tools, Emulation Complete (with Line Issues 9 0), Execute, Reset, Step Backwards, and Step Forwards.

The main area displays assembly code:

```
1      mov    r0, #0x3a
2      mov    r1, #0
3
4
5      loop
6      tst    r0, #1
7      addne r1, r1, #1
8      lsrs   r0, r0, #1
9      bne    loop
10
11
12
13
14
15
16
17
18
19
20
```

A message "Reset to continue editing code" is displayed above the assembly code.

To the right, there is a register table showing the values for R0 through R12:

R0	0x0	Dec	Bin	Hex
R1	0x4	Dec	Bin	Hex
R2	0x0	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex

At the bottom, there are status indicators: Clock Cycles (1), Current Instruction (1 Total: 36), and CSPR Status Bits (NZCV) with values 0 1 1 0.

<https://salmanarif.bitbucket.io/visual/>

# **Why assembly?**

What you see is what you get

No surprises

Precise control, timing

Unfettered access to hardware

# **Why not assembly?**

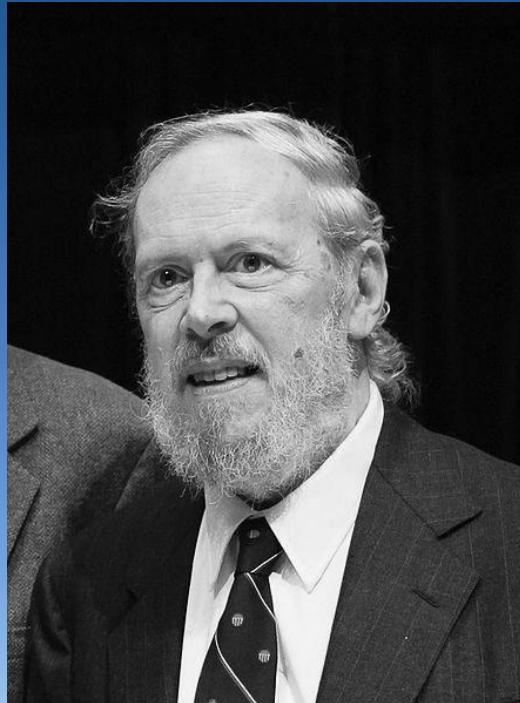
Tedious

Error-prone

Not readable

Not portable

*Real question is not whether to use assembly, but when...*



**Dennis Ritchie**

SECOND EDITION

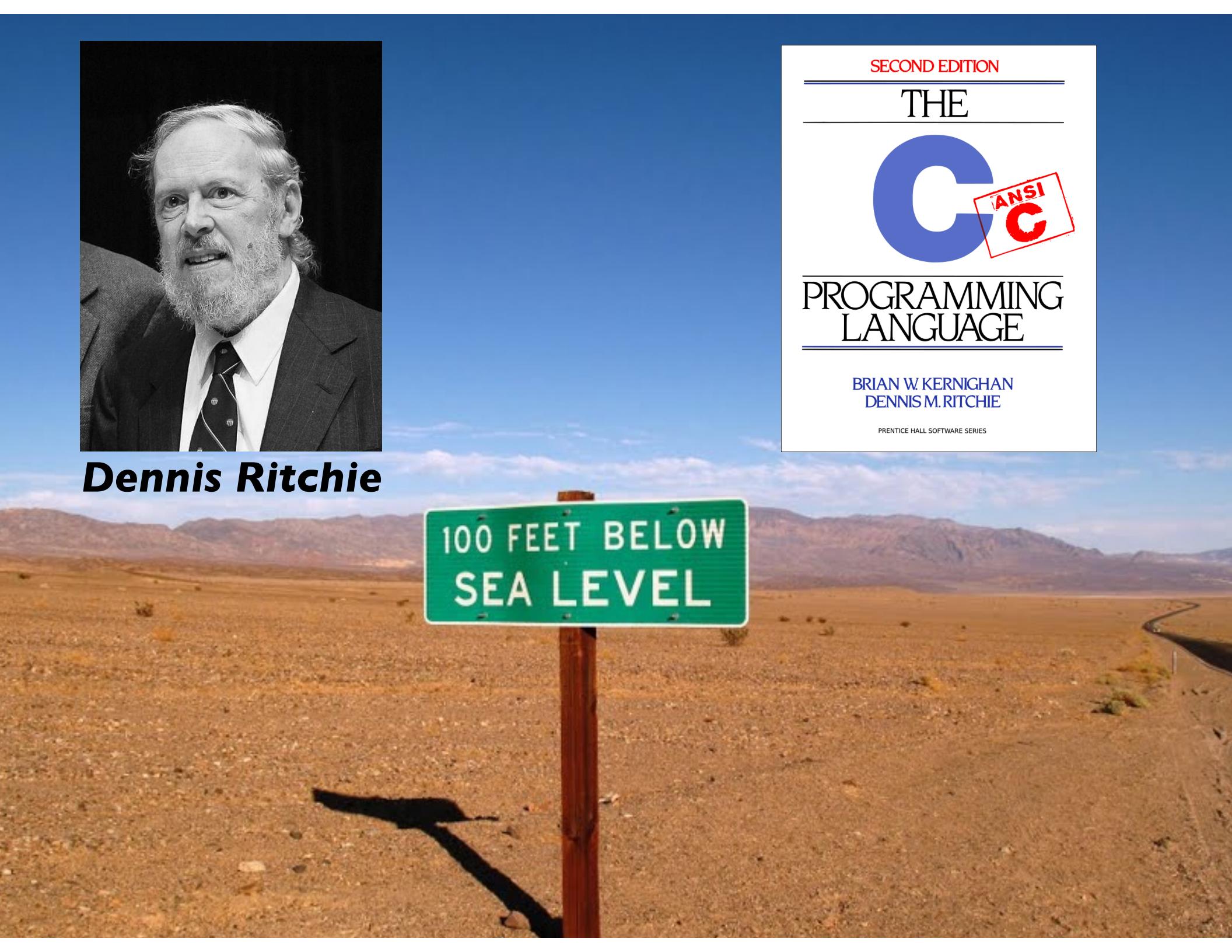
THE



**PROGRAMMING  
LANGUAGE**

BRIAN W. KERNIGHAN  
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

A green road sign on a wooden post in a desert landscape. The sign reads "100 FEET BELOW SEA LEVEL".

100 FEET BELOW  
SEA LEVEL

# C is the language of choice for systems programmers



*Ken Thompson built UNIX using C*

This is not coincidence!  
C features closely model the ISA:  
data types, arithmetic/logical  
operators, control flow, access to  
memory, ... all provided in form  
of portable abstractions

“BCPL, B, and C family of languages are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are “close to the machine” in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. ... At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.”

— Dennis Ritchie

# The C Programming Language

“C is quirky, flawed, and an enormous success”

— Dennis Ritchie

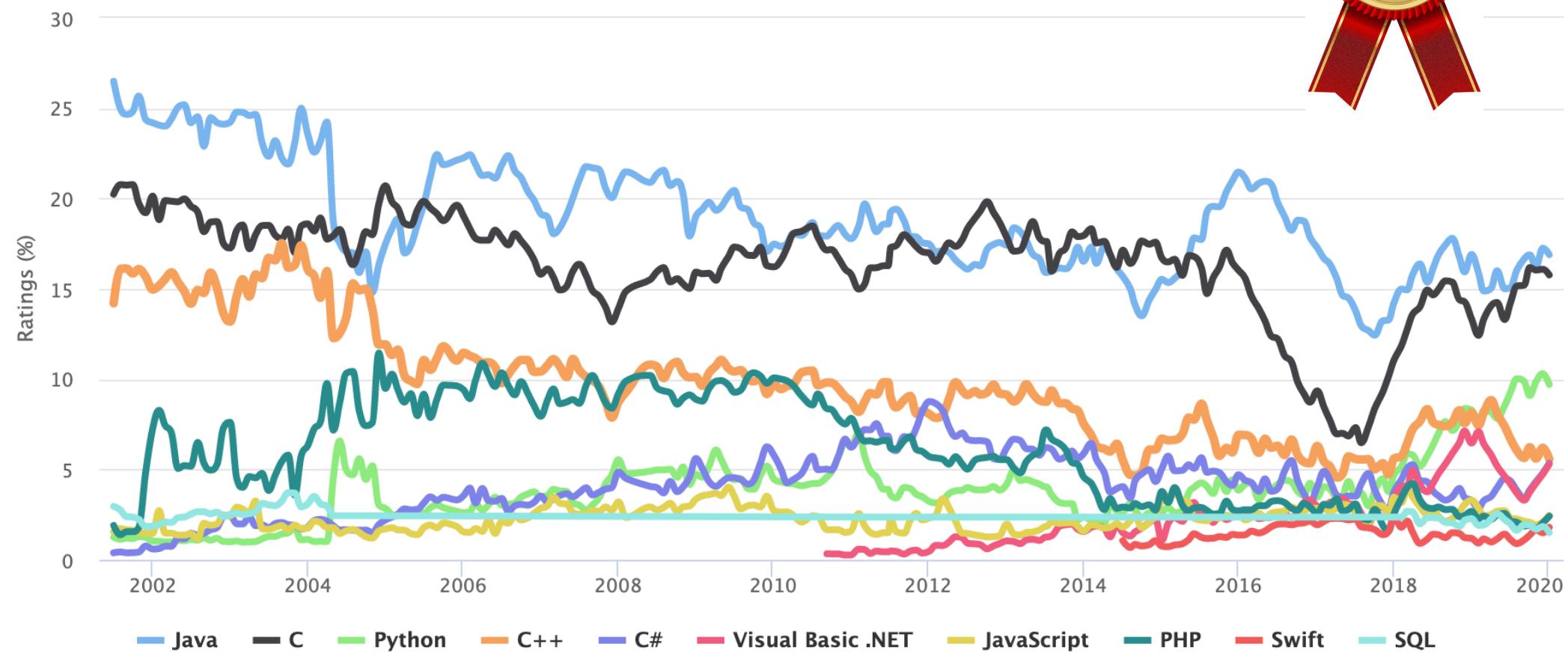
“C gives the programmer what the programmer wants; few restrictions, few complaints”

— Herbert Schildt

“C: A language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language”

— Unknown

January 2020 headline:  
Programming Language of the Year 2019...



Programming language popularity over time

# Compiler Explorer

is a neat interactive tool to see translation from C to assembly.  
Let's try it now!

The screenshot shows the Compiler Explorer interface. On the left, the 'C source #1' tab contains the following C code:

```
1 int global = 107;
2
3 void main(void)
4 {
5     global = global + 1;
6 }
```

The code editor has a red box around the dropdown menu where 'C' is selected. To the right, the 'ARM gcc 5.4.1 (none)' tab displays the generated assembly code:

```
1 main:
2     ldr    r2, .L2
3     ldr    r3, [r2]
4     add    r3, r3, #1
5     str    r3, [r2]
```

The assembly tab also has a red box around the 'ARM gcc 5.4.1 (none)' dropdown and another red box around the '-Og' optimization flag. Below the tabs, there are several output options: 11010, ./a.out, .LX0 (which is checked), lib.f., .text, and //.

<https://godbolt.org>

Configure settings to follow along:

C

ARM gcc 5.4.1(none)

-Og

# Why C?

## **Higher-level abstractions, structured programming**

Named variables, constants

Arithmetic/logical operators

Control flow

## **Portable**

Not tied to particular ISA or architecture

## **Low-level enough to get to machine when needed**

Bitwise operations

Direct access to memory

Embedded assembly, too!

# Know your tools!

## **Assembler** as

Transform assembly code (text)  
into object code (binary machine instructions)  
Mechanical rewrite, few surprises

## **Compiler** gcc

Transform C code (text)  
into object code  
(likely staged C → asm → object)

Complex translation, high artistry

# Make

One-step build process using `make`

`Makefile` is text file that describes build steps as "recipes"

Dependencies determine which steps needed to re-build

```
mine.bin: mine.elf  
        arm-none-eabi-objcopy mine.elf -O binary mine.bin  
  
mine.elf: mine.o  
        arm-none-eabi-gcc -nostdlib -e main mine.o -o mine.elf  
  
mine.o: mine.c  
        arm-none-eabi-gcc -ffreestanding -c mine.c -o mine.o
```

# Make pattern rules

```
NAME = mine
```

```
CFLAGS = -Og -g -Wall -std=c99 -ffreestanding  
LDFLAGS = -nostdlib -e main
```

```
all : $(NAME).bin
```

```
%.bin: %.elf  
        arm-none-eabi-objcopy $< -O binary $@
```

```
%.elf: %.o  
        arm-none-eabi-gcc $(LDFLAGS) $< -o $@
```

```
%.o: %.c  
        arm-none-eabi-gcc $(CFLAGS) -c $< -o $@
```

```
%.o: %.s  
        arm-none-eabi-as $< -o $@
```

# Bare-metal vs. Hosted

The default build process for C assumes a *hosted* environment.

What does a hosted system have that we don't?

- standard libraries
- standard start-up sequence
- OS services

To build bare-metal, our Makefile disables these defaults

We supply our own replacements where needed

# Build settings for bare-metal

Compile freestanding

**CFLAGS = -ffreestanding**

Link excludes standard library, start files

**LDFLAGS = -nostdlib**

Link with gcc if need division (b/c no ARM divide instruction)

**LDLIBS = -lgcc**

Write our own code for all libs and start files

This puts us in an exclusive club...

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```