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

Course grades

A

- **Complete all labs and basic assignments correctly**
- **Complete 3 extensions correctly**
- **Outstanding final project**

B

- **Complete all labs and basic assignments that work**
- **Good final project**

C or below

- **Partial credit on a significant amount of the work**

Goals for today

Understand relationship between C and asm

How to compile/build from C

Review C syntax, view translation to asm

Explore C pointers/memory

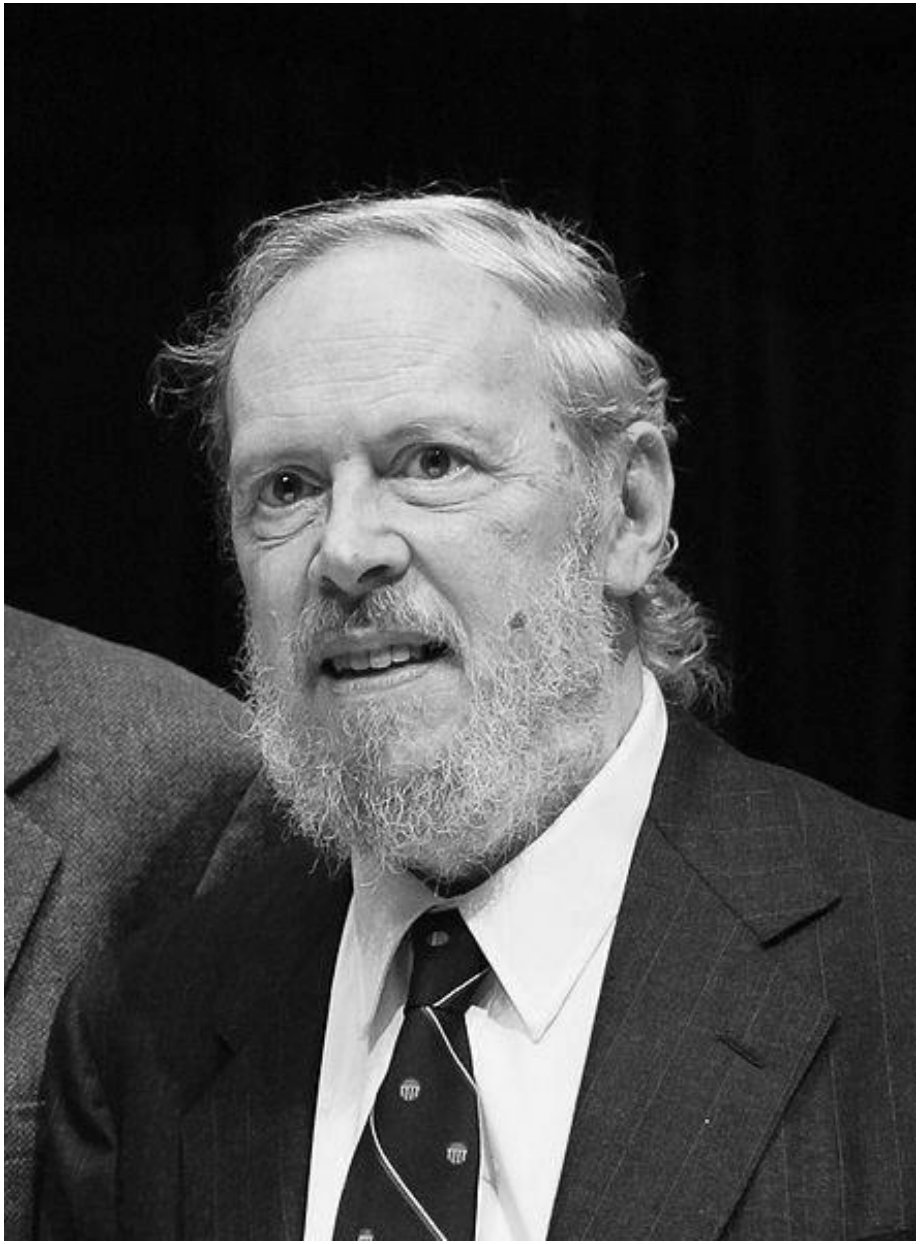
```
// What does this code do?

   mov    r3, #1
   mov    r2, #0
   b     .L2
.L1:
   add    r2, r2, r3
   add    r3, r3, #1
.L2:
   cmp    r3, #100
   ble    .L1
```

```
// What does this code do?

  ldr  r3, [pc, #12]
  mov  r2, #1
  str  r2, [r3, #8]
  mov  r2, #0x100000
  str  r2, [r3, #28]
.word  20200000
```

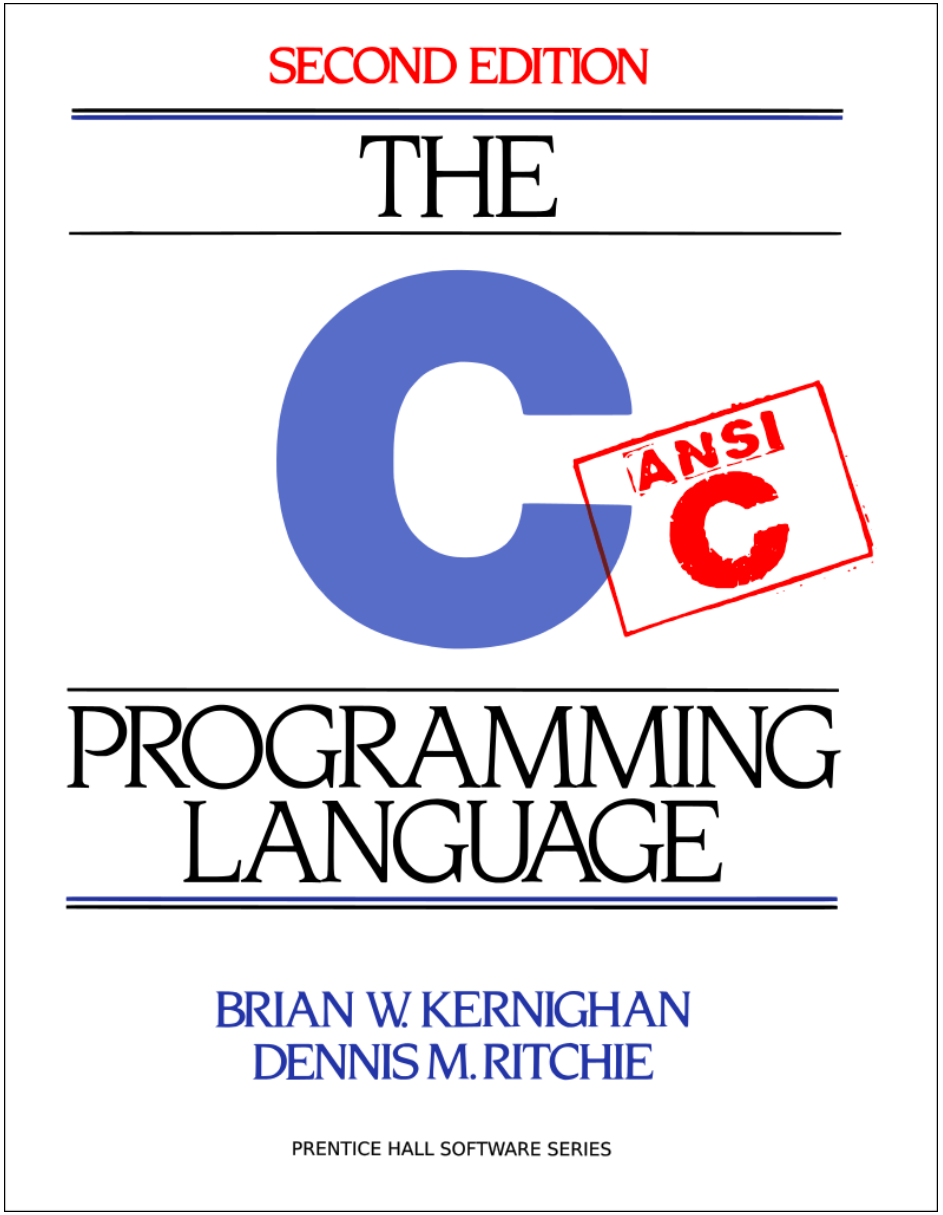
Assembly Language

the instructions you see are the instructions you get

```
#define MAX 100

void main()
{
   int sum = 0;

   for (int i = 1; i <= MAX; i++)
      sum += i;

   *(int *)0x20200008 = 1;
   *(int *)0x2020001C = 1 << 20;
```



Dennis Ritchie



“BCPL, B, and C all fit firmly in the traditional procedural family (of languages) typified by Fortran and Algol 60. They 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

Ken Thompson built UNIX using C

http://cm.bell-labs.com/cm/cs/who/dmr/picture.html

Know your tools: make

Makefile is a text file of “recipes”to manage build process

Recipe lists prerequisites and gives sequence of steps to build target from its ingredients

Syntax a bit goopy, whitespace matters

```
CC = arm-none-eabi-gcc
CFLAGS = -g -Wall -std=c99 -Og -ffreestanding

# Pattern rule to compile C source file
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@
```

see http://cs107e.github.io/guides/make/

C language features closely model ISA:
data types, arithmetic/logical operators,
control flow, access to memory, …

Online GCC explorer [https://gcc.godbolt.org]
is neat interactive tool to look at translations

Pointer types

Address **is a memory location, is represented as an unsigned int**

Pointer **is a variable that holds an address**

"*Pointee"* is the data stored at that address

At asm level, all addresses "look" same as each other/ints. In C, type system distinguishes pointers by type of pointee. The *type* of `int` is not same as `int*` nor is `int*` same type as `char*`.

Operations are restricted/different per-type, e.g.:

- **can't multiply `int*`'s, can't deference an `int`**
- **addition has different behavior for `int` vs. `int*` vs. `double*`**

Pointer basics

```
int m, n, *p, *q;

p = &n;
*p = n;       // same as prev line?

q = p;
*q = *p;      // same as prev line?

p = &m, q = &n;
*p = *q;
m = n;        // same as prev line?
```

Pointer and arrays

```
int n, arr[4], *p;

p = arr;
p = &arr[0]; // same as prev line?

*p = 3;
p[0] = 3;    // same as prev line?

n = *(arr + 1);
n = arr[1];  // same as prev line?
```

Pointers: the fault in our *s

Pointers are ubiquitous in C, and inherently dangerous. Watch out!

Q. For what reasons might a pointer be invalid?

Q. What is consequence of using an invalid pointer?

ARM addressing modes

```
  str r0, [r1]              // indirect

Preindex, non-updating
  str r0, [r1, #4]          // constant displacement
  str r0, [r1, r2]          // variable displacement
  str r0, [r1, r2, lsl #4]  // scaled index

Preindex, writeback (update before use)
  str r0, [r1, #4]!         // r1 pre-updated += 4
  str r0, [r1, r2]!
  str r0, [r1, r2, lsl #4]!

Postindex (update after use)
  str r0, [r1], #4          // r1 post-updated += 4
  ...
```

How do these modes map to C language features?

Pointee type matters!

```
char *mem = (char *)0;
int *gpio = (int *)0x20200000;

// which of these are same?

mem[0x2020008] = 1;
*(int *)(mem + 0x2020008) = 1;

*(gpio + 0x8) = 1;
gpio[2] = 1;
```

Hints (Previous Lecture)

Start with the simplest program.

Take baby steps, check that things work, then take another small step …

If something doesn't work, backup to a known working state. Identify state 0.

Start by typing it in by hand; do not learn by copy and pasting

Hints

Test your understanding with experiments. Hypothesize what will happen, try it, and see if that is what happened.

Rearrange the code so that it is easier to understand, but does the exact same thing. Compare the two versions to make sure they do the same thing.