The C Programming Language

```
// Turn on an LED

// configure GPIO 20 for OUTPUT
ldr r0, =0x20200008 // FSEL2
mov r1, #1
str r1, [r0]

// set  GPIO 20 to 3.3V
ldr r0, =0x2020001C // SET0
mov r1, #(1<<20)
str r1, [r0]

loop: b loop
```

Assembly Language

the instructions you see are the instructions you get

```
// code/on
main()
{
    int *r0;
    int r1;

    // configure GPIO 20 for output
    r0 = (int*)0x20200008; // ldr r0, =0x2020008
    r1 = 1;                // mov r1, #1
    *r0 = r1;              // str r1, [r0]

    // set GPIO 20 to 3.3V
    r0 = (int*)0x2020001C; // ldr r0, =0x202001C
    r1 = 1<<20;            // mov r1, #1
    *r0 = r1;              // str r1, [r0]

loop:
    goto loop;
}
```

```
Disassembly

8000 ldr     r3, [pc, #16] ; 8018
8004 mov     r2, #1
8008 str     r2, [r3, #8]
800C mov     r2, #0x100000
8010 str     r2, [r3, #28]
8014 b       8014
8018 20200000
```

Bare Metal

`-ffreestanding`

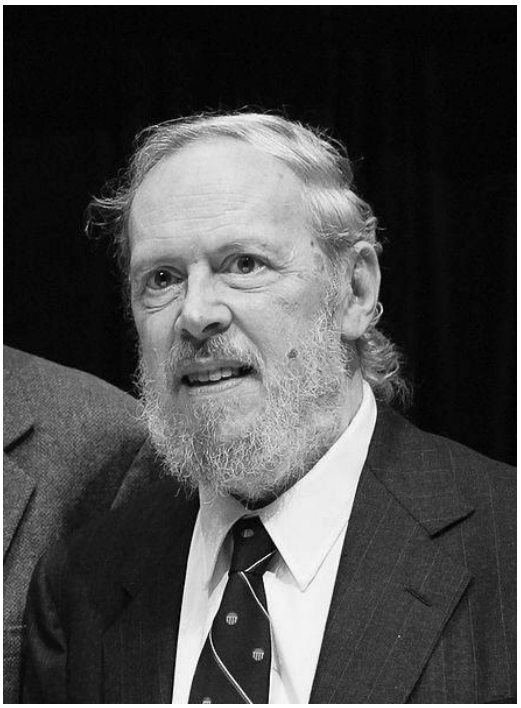
- **Program does not "stand on" (require) an operating system.**

`-nostdlib`

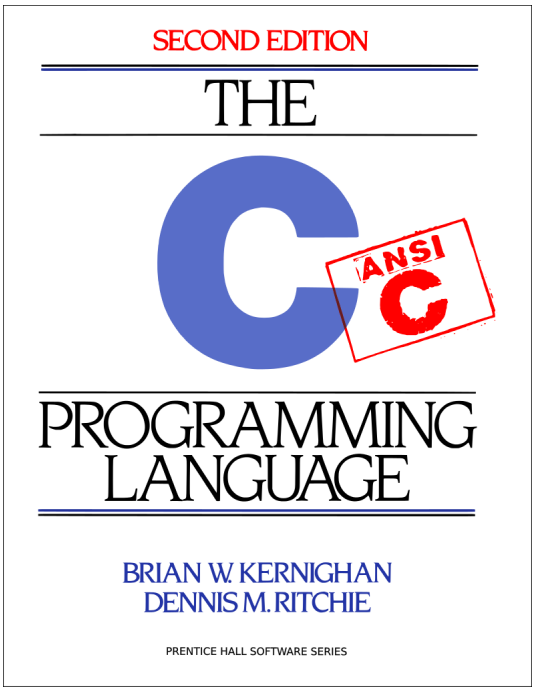
- **Program does not use standard libraries by default**

`-nostartfiles`

- **Don't run any start code when the program starts. The program will provide the start code.**



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

```
main()
{
    int *r0;

    r0 = (int*)0x20200000;
    *(r0+0x08/4) = 1;
    *(r0+0x1C/4) = 1 << 20;

    while( 1 ) ;
}
```

```
main()
{
    int *r0;

    r0 = (int*)0x20200000;
    *(r0+0x08/4) = 1;
    *(r0+0x1C/4) = 1 << 20;

    while( 1 ) ;
}
```

```
main()
{
    int *r0;

    r0 = (int*)0x20200000;
    r0[0x08/sizeof(int*)] = 1;
    r0[0x1C/sizeof(int*)] = 1 << 20;

    while( 1 ) ;
}
```

```
main()
{
    int *r0;

    r0 = (int*)0x20200000;
    r0[0x08/sizeof(*r0)] = 1;
    r0[0x1C/sizeof(*r0)] = 1 << 20;

    while( 1 ) ;
}
```

```
main()
{
    int a[2];
    a[0] = …
    a[1] = …

    int *p = a;
    p[0] = …
    p[1] = …
}
```

```
main()
{
    int *r0, r1;

    r0 = 0;
    for( r1 = 0; r1 < 0x400000; r1++ )
       r0[r1] = 0;
}

// What's this program do?
```

```
main()
{
    int *mem;
    int addr;

    mem = 0;
    for(addr=0; addr<0x400000; addr++)
       mem[addr] = 0;
}

// What's this program do?
```

```
int *p, n;

n = 1;
p = &n; // p is the address of n

*p = 2; // assign 2 to n
```

```
// Which of these statements generate errors?
// Hint: What operations make sense on addresses

   int *p, *q, n;

   n = 0x20200000;
   p = n;
   q = p + 1;
   q = 2 * p;
   n = q - p;
   if( p == q ) ;
   if( p != q ) ;
   if( p < q )  ;
   if( p == 0 ) ;
   if( p == 1 ) ;
   q = p & (~3);
   q = p | q;
   n = q;
   p = &(n+1);
   n = *&n;
   p = &*p
   p = &3;
```

Types and Pointers

`int` **is an integer**

`int*` **is the address of an integer**

Addresses are addresses, but what they point to may be a different type

The *type* of `int` **is not the same as** `int*`

The operations on `int` **are different than the operations on** `int*`

- **e.g. can't multiply 2** `int*`**'s**
- **e.g. + is different on** `int` **and** `int*`

Pointers are DANGEROUS

Hints (Previous Lecture)

Start with the simplest program. Try to make it simpler.

Take baby steps. Check each step, check it again, ..., and then take another step.

Start by typing it in by hand; do not learn by cutting 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.