Lecture 2: Bits, Bytes, Ints

CS 105

January 28, 2019

The C Language

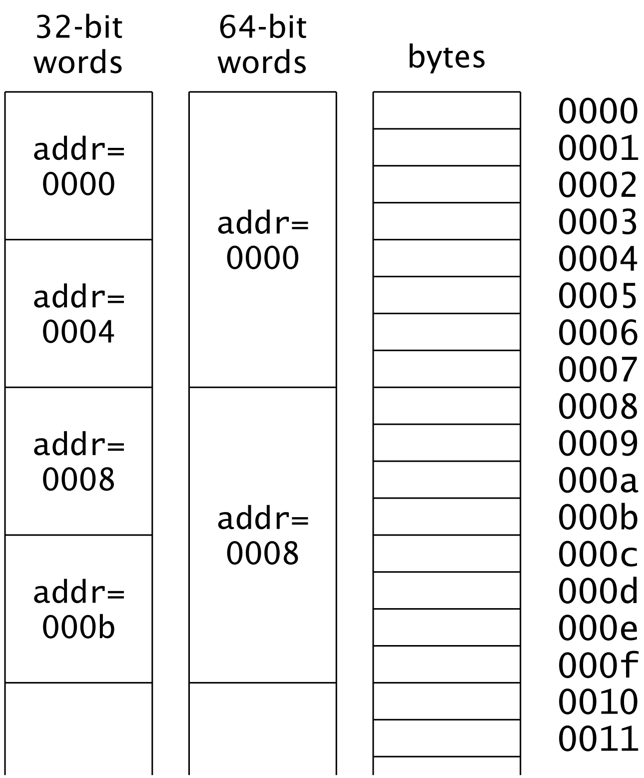
- Syntax like Java: declarations, **`if`**, **`while`**, **`return`**

- Data and execution model are "closer to the machine"
  - More power and flexibility
  - More ways to make mistakes
  - Sometimes confusing relationships
  - Pointers!!

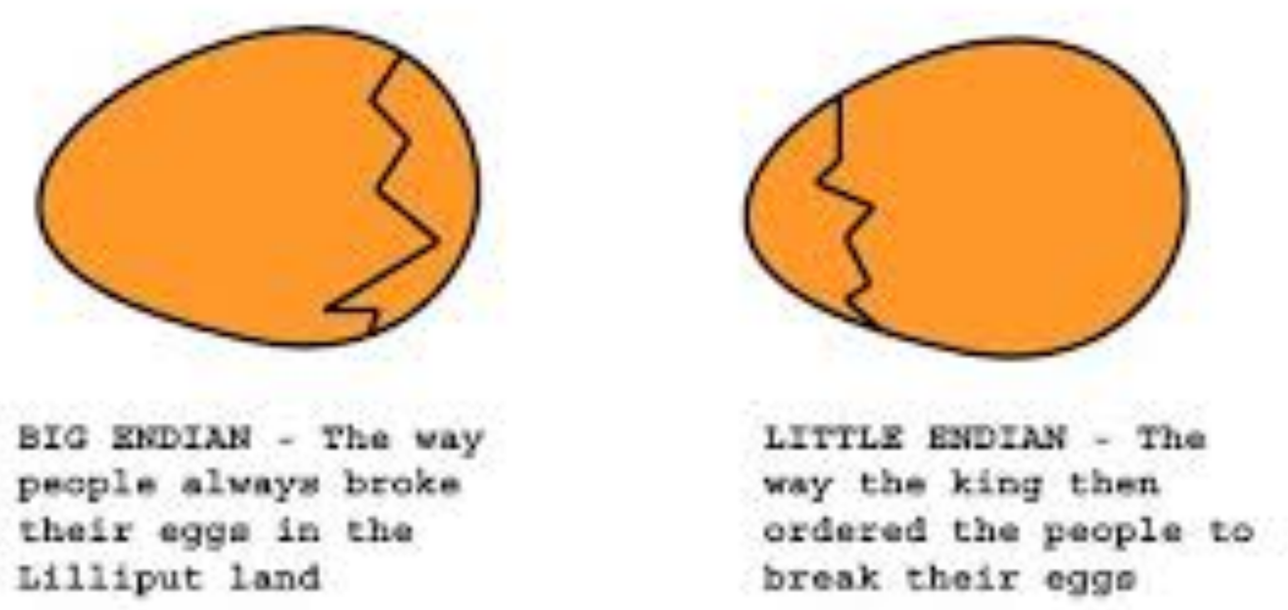
- A possible resource from CMU:
  - http://www.cs.cmu.edu/afs/cs/academic/class/15213-s16/www/recitations/c_boot_camp.pdf

Memory: A (very large) array of bytes

- An index into the array is an *address*, *location*, or *pointer*
  - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address
- Larger words (32- or 64-bit) are stored in contiguous bytes
  - The address of a word is the address of its first byte
  - Successive addresses differ by word size



Endianness



BIG ENDIAN - The way people always broke their eggs in the Lilliput land
LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic---encode “True” as 1 and “False” as 0

And

&	0	1
0	0	0
1	0	1

Or

|	0	1
0	0	1
1	1	1

Not

~	
0	1
1	0

Exclusive-Or (Xor)

^	0	1
0	0	1
1	1	0

General Boolean algebras

- Bitwise operations on words

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- How does this map to set operations?

Practice with Boolean algebras

- Assume: a = 01101001, b = 01010101

- What are the results of evaluating the following Boolean operations?

  - ~a
  - ~b
  - a & b
  - a | b
  - a ^ b

Bitwise vs Logical Operations in C

- Apply to any "integral" data type
  - `int, unsigned, long, short, char`
- Bitwise Operators &, |, ~, ^
  - View arguments as bit vectors
  - operations applied bit-wise in parallel
- Logical Operators `&&, ||, !`
  - View 0 as "False"
  - View anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

Bitwise vs Logical Operations in C

- Exercises (char data type, one byte)

  - `~0x41`
  - `~0x00`
  - `~~0x41`

  - `0x69 & 0x55`
  - `0x69 | 0x55`

  - `!0x41`
  - `!0x00`
  - `!!0x41`

  - `0x69 && 0x55`
  - `0x69 || 0x55`

Bit Shifting

- Left Shift: **`x << y`**
  - Shift bit-vector **`x`** left **`y`** positions
  - Throw away extra bits on left
  - Fill with 0's on right

Undefined Behavior if you shift amount < 0 or ≥ word size

- Right Shift: **`x >> y`**
  - Shift bit-vector **`x`** right **`y`** positions
  - Throw away extra bits on right
  - Logical shift: Fill with 0's on left
  - Arithmetic shift: Replicate most significant bit on left

Choice between logical and arithmetic depends on the type of data

Bit Shifting

- `0x41 << 4`
- `0x41 >> 4`

- `41 << 4`
- `41 >> 4`
- `-41 << 4`
- `-41 >> 4`

Representing Unsigned Integers

- Think of bits as the binary representation

$$\text{UnsignedValue}(x) = \sum_{j=0}^{w-1} x_j \cdot 2^j$$

- If you have w bits, what is the range?
- Can only represent non-negative numbers

Representing Signed Numbers

- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude
- Option 2: excess-K
  - Choose a positive K in the middle of the unsigned range
  - SignedValue(w) = UnsignedValue(w) – K
- Option 3: one’s complement
  - Flip every bit to get the negation

Representing Signed Integers

- Option 4: two's complement
  - Most commonly used
  - Like unsigned, except the high-order contribution is *negative*

$$\mathrm{SignedValue}(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{j=0}^{w-2} x_j \cdot 2^j$$

- Assume C short (2 bytes)
  - What is the hex/binary representation for 47?
  - What is the hex/binary representation for -47?

Example: Three-bit integers

unsigned		signed
111	7	
110	6	
101	5	
100	4	
011	3	011
010	2	010
001	1	001
000	0	000
	-1	111
	-2	110
	-3	101
	-4	100

- The high-order bit is the *sign bit.*
- The largest unsigned value is $11\ldots1$, UMax.
- The signed value for -1 is always $11\ldots1$.
- Signed values range between `TMin` and TMax.

This representation of signed values is called *two's complement.*

Two’s Complement Signed Integers

- “Signed” does not mean “negative”
- High order bit is the *sign bit*
  - To negate, complement all the bits and add 1
  - Remember the arithmetic right shift
  - Sign extension
- Arithmetic is the same as unsigned—same circuitry
- Error conditions and comparisons are different

Unsigned and Signed Integers

- Use w-bit words; w can be 8, 16, 32, or 64
- The bit sequence $b_{w-1} \ldots b_1\, b_0$ represents an integer

	unsigned	signed
value	$\sum_{i=0}^{w-1} b_i 2^i$	$-b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$
smallest	0	-2^{w-1}
largest	$2^w - 1$	$2^{w-1} - 1$

- Important!! ”signed” does not mean “negative”

Important Signed Numbers

	8	16	32	64
TMax	`0x7F`	`0x7FFF`	`0x7FFFFFFF`	`0x7FFFFFFFFFFFFFFF`
TMin	`0x80`	`0x8000`	`0x80000000`	`0x8000000000000000`
0	`0x00`	`0x0000`	`0x00000000`	`0x0000000000000000`
-1	`0xFF`	`0xFFFF`	`0xFFFFFFFF`	`0xFFFFFFFFFFFFFFFF`

Fun with Integers: Using Bitwise Operations

- x & 1 — "x is odd"
- (x + 7) & 0xFFFFFFF8 — "round up to a multiple of 8"

- p & ~0x3FF — "start of 1K block containing p" (ish)
- ((p >> 10) << 10) — same location (really)

- p & 0x3FF — "offset of p within the block"