COS 217: Introduction to Programming Systems

Crash Course in C (Part 2)

The Design of C Language Features and
Data Types and their Operations and Representations

PRINCETON UNIVERSITY

INTEGERS

Integer Data Types

Integer types of various sizes: {signed, unsigned} {char, short, int, long}

- char is 1 byte
  - Number of bits per byte is unspecified!
    (but in the 21st century, safe to assume it's 8)
- Sizes of other integer types not fully specified but constrained:
  - int was intended to be "natural word size" of hardware
  - $2 \leq$ sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long)

On ArmLab:

- Natural word size: 8 bytes ("64-bit machine")
- char: 1 byte
- short: 2 bytes
- int: 4 bytes (compatibility with widespread 32-bit code)
- long: 8 bytes

What decisions did the designers of Java make?

Integer Literals

- Decimal int: 123
- Octal int: 0173 = 123
- Hexadecimal int: 0x7B = 123
- Use "L" suffix to indicate long literal
- No suffix to indicate char-sized or short integer literals; instead, cast

Examples

- int: 123, 0173, 0x7B
- long: 123L, 0173L, 0x7BL
- short: (short)123, (short)0173, (short)0x7B

Unsigned Integer Data Types

unsigned types: unsigned char, unsigned short, unsigned int, and unsigned long

- Hold only non-negative integers

Default for short, int, long is signed

- char is system dependent (on armlab char is unsigned)
- Use "U" suffix to indicate unsigned literal

Examples

- unsigned int:
  - 123U, 0173U, 0x7BU
  - Oftentimes the U is omitted for small values: 123, 0173, 0x7B
    - (Technically there is an implicit cast from signed to unsigned, but in these cases it shouldn't make a difference.)
- unsigned long:
  - 123UL, 0173UL, 0x7BUL
- unsigned short:
  - (unsigned short)123, (unsigned short)0173, (unsigned short)0x7B

“Character” Data Type

The C char type

- char is designed to hold an ASCII character
  - Should be used when you’re dealing with characters: character-manipulation functions we’ve seen (such as toupper) take and return char
- char might be signed (-128..127) or unsigned (0..255)
  - But since $0 \leq$ ASCII ≤ 127 it doesn’t really matter when used as an actual character
  - If using chars for arbitrary one-byte data, good to specify as unsigned char

Character Literals

Single quote syntax: 'a'

Use backslash (the escape character) to express special characters

- Examples (with numeric equivalents in ASCII):

```
'a'     the a character (97, 01100001B, 61H)
'\141'  the a character, octal form
'\x61'  the a character, hexadecimal form
'b'     the b character (98, 01100010B, 62H)
'A'     the A character (65, 01000001B, 41H)
'B'     the B character (66, 01000010B, 42H)
'\0'    the null character (0, 00000000B, 0H)
'0'     the zero character (48, 00110000B, 30H)
'1'     the one character (49, 00110001B, 31H)
'\n'    the newline character (10, 00001010B, AH)
'\t'    the horizontal tab character (9, 00001001B, 9H)
'\\'    the backslash character (92, 01011100B, 5CH)
'\''    the single quote character (96, 01100000B, 60H)
```

Unicode

Back in 1970s, English was the only language in the world[citation needed]

so we all used this alphabet [citation needed] :

ASCII:

American Standard Code
for Information Interchange

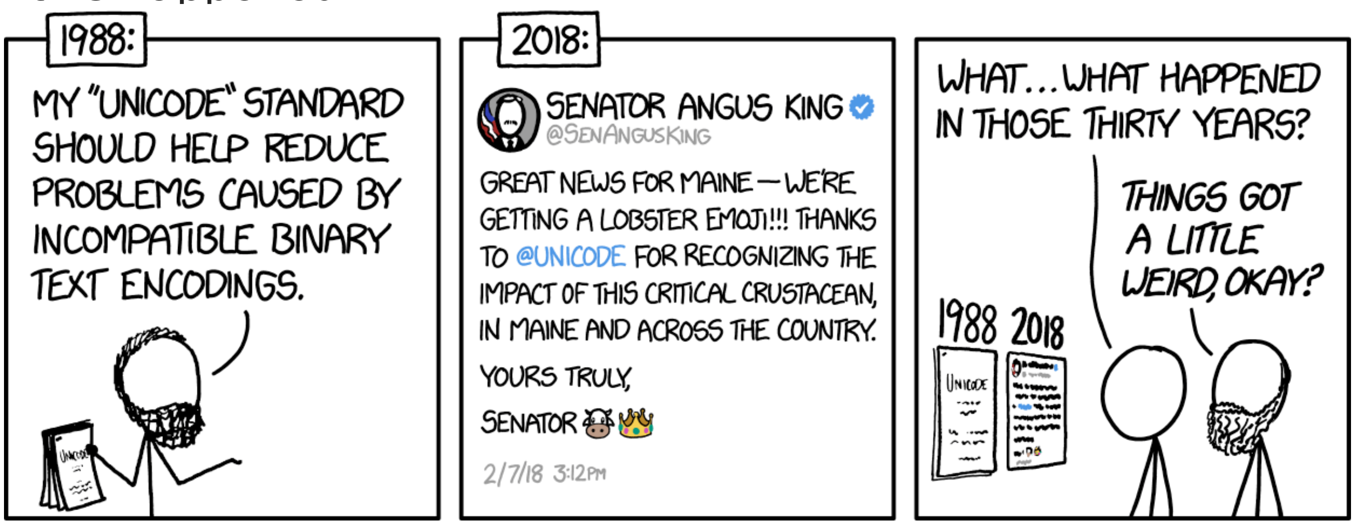
In the 21st century, it turns out there are other languages!

Modern Unicode

When C was designed, it only considered ASCII, which fits in 7 bits, so C's chars are 8 bits long.

When Java was designed, Unicode fit into 16 bits, so Java's chars are 16 bits long.

Then this happened:



https://xkcd.com/1953/

Integer Types in Java vs. C

	Java	C

Unsigned types	`char // 16 bits`	`unsigned char /* Note 2 */`
		`unsigned short`
		`unsigned (int)`
		`unsigned long`

Signed types	`byte // 8 bits`
	`short // 16 bits`
	`int // 32 bits`
	`long // 64 bits`
		`signed char /* Note 2 */`
		`(signed) short`
		`(signed) int`
		`(signed) long`

1. Not guaranteed by C, but on **`armlab`**, **`char`** = 8 bits, **`short`** = 16 bits, **`int`** = 32 bits, **`long`** = 64 bits
2. Not guaranteed by C, but on **`armlab`**, **`char`** is unsigned

To understand C, must consider the representation of these types!

Representing Unsigned Integers

Mathematics

- Non-negative integers' range is 0 to ∞

Computer programming

- Range limited by computer's word size
- Word size is n bits ⇒ range is 0 to 2n – 1
- Exceed range ⇒ overflow

Typical computers today

- n = 32 or 64, so range is 0 to $2^{32} - 1$ (~4B) or $2^{64} - 1$ (huge … ~1.8e19)

Pretend computer

- n = 4, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4

- All points generalize to word size = n (armlab: 64)

Representing Unsigned Integers

On 4-bit pretend computer

Unsigned Integer	Rep
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Adding Unsigned Integers

Addition

```
               1
  3         0011B
+ 10      + 1010B
  --        ----
  13        1101B
```

Start at right column
Proceed leftward
Carry 1 when necessary

```
             111
  7         0111B
+ 10      + 1010B
  --        ----
  1         0001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

Subtracting Unsigned Integers

Subtraction

```
               111
  10          1010B
-  7        - 0111B
  --          ----
   3          0011B
```

Start at right column
Proceed leftward
Borrow when necessary

```
              1
   3          0011B
- 10        - 1010B
  --          ----
   9          1001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

Shifting Unsigned Integers

Bitwise right shift (>> in C): fill on left with zeros

`10 >> 1 ⇒ 5`

1010_B 0101_B

`10 >> 2 ⇒ 2`

1010_B 0010_B

What is the effect arithmetically?

Bitwise left shift (<< in C): fill on right with zeros

`5 << 1 ⇒ 10`

0101_B 1010_B

`3 << 2 ⇒ 12`

0011_B 1100_B

`3 << 3 ⇒ 8`

0011_B 1000_B ← Results are mod 2^4

What is the effect arithmetically?

Other Bitwise Operations on Unsigned Integers

Bitwise NOT (~ in C)

- Flip each bit

```
~10 ⇒ 5
```

1010_B 0101_B

```
~5 ⇒ 10
```

0101_B 1010_B

Bitwise AND (& in C)

- AND (1=True, 0=False) corresponding bits

```
 10         1010B
& 7       & 0111B
 --         ----
  2         0010B
```

```
 10         1010B
& 2       & 0010B
 --         ----
  2         0010B
```

Useful for "masking" bits to 0

Other Bitwise Operations on Unsigned Ints

Bitwise OR: (| in C)

- Logical OR corresponding bits

```
   10        1010B
|   1      | 0001B
   --        ----
   11        1011B
```

Useful for “masking” bits to 1

Bitwise exclusive OR (^ in C)

- Logical exclusive OR corresponding bits

```
   10        1010B
^  10      ^ 1010B
   --        ----
    0        0000B
```

x ^ x sets
all bits to 0

A Bit Complicated

How do you set bit k (where the least significant bit is bit 0) of unsigned variable u to zero (leaving everything else in u unchanged)?

A. u &= (0 << k);

B. u |= (1 << k);

C. u |= ~(1 << k);

D. u &= ~(1 << k);

E. u = ~u ^ (1 << k);

D:

1 << k ➔ 0{n-1-k}10{k}

~(1 << n) ➔ 1{n-1-k}01{k}

u &= ~(1 << k); ➔ u_i{n-1-k}0u_i{k}

Aside: Using Bitwise Ops for Arith

Can use <<, >>, and & to do some arithmetic efficiently

$x * 2^y$ == x << y

- $3*4 = 3*2^2$ = 3<<2 ⇒ 12

Fast way to **multiply** by a power of 2

$x / 2^y$ == x >> y

- $13/4 = 13/2^2$ = 13>>2 ⇒ 3

Fast way to **divide** **unsigned** by power of 2

$x \% 2^y$ == x & (2^y-1)

- $13\%4 = 13\%2^2 = 13\&(2^2-1)$ = 13&3 ⇒ 1

Fast way to **mod** by a power of 2

```
 13        1101B
& 3      & 0011B
 --        ----
  1        0001B
```

Many compilers will do these transformations automatically!

Unfortunate reminder: negative numbers exist

Sign-Magnitude

Integer	Rep
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit indicates sign

- `0 ⇒ positive`
- `1 ⇒ negative`

Remaining bits indicate magnitude

$0101_B = 101_B = 5$

$1101_B = -101_B = -5$

Sign-Magnitude (cont.)

Integer	Rep
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Computing negative

neg(x) = flip high order bit of x

$\mathtt{neg}(0101_B) = 1101_B$

$\mathtt{neg}(1101_B) = 0101_B$

Pros and cons

+ easy to understand, easy to negate
+ symmetric
- two representations of zero
- need different algorithms to add signed and unsigned numbers

Ones' Complement

Integer	Rep
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight $-(2^{b-1}-1)$

$1010_B = (1*-7)+(0*4)+(1*2)+(0*1)$
$= -5$

$0010_B = (0*-7)+(0*4)+(1*2)+(0*1)$
$= 2$

Similar pros and cons to sign-magnitude

Two's Complement

Integer	Rep
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight $-(2^{b-1})$

$1010_B = (1*-8)+(0*4)+(1*2)+(0*1)$
$= -6$

$0010_B = (0*-8)+(0*4)+(1*2)+(0*1)$
$= 2$

Two's Complement (cont.)

Integer	Rep
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Computing negative

neg(x) = ~x + 1

neg(x) = onescomp(x) + 1

neg(0101_B) = 1010_B + 1 = 1011_B

neg(1011_B) = 0100_B + 1 = 0101_B

Pros and cons

- not symmetric
  ("extra" negative number)

+ one representation of zero

+ same algorithm adds
  signed and unsigned integers

Adding Signed Integers

pos + pos

```
              11
   3        0011B
 + 3      + 0011B
  --        ----
   6        0110B
```

pos + pos (overflow)

```
              111
   7        0111B
 + 1      + 0001B
  --        ----
  -8        1000B
```

pos + neg

```
            1111
   3        0011B
+ -1      + 1111B
  --        ----
   2        0010B
```

How would you detect overflow programmatically?

neg + neg

```
            11
  -3        1101B
+ -2      + 1110B
  --        ----
  -5        1011B
```

neg + neg (overflow)

```
            1 1
  -6        1010B
+ -5      + 1011B
  --        ----
   5        0101B
```

Subtracting Signed Integers

Perform subtraction with borrows

or

Compute two's comp and add

```
          11
  3       0011B
- 4     - 0100B
 --       ----
 -1       1111B
```

```
  3       0011B
+ -4    + 1100B
 --       ----
 -1       1111B
```

```
          11
 -5       1011B
--2     - 1110B
 --       ----
 -3       1101B
```

```
            1
 -5       1011B
+ 2     + 0010B
 --       ----
 -3       1101B
```

Negating Signed Ints: Math

Question: Why does two's comp arithmetic work?

Answer: $[-b] \bmod 2^4 = [\text{twoscomp}(b)] \bmod 2^4$

$$
\begin{aligned}
& [-b] \bmod 2^4 \\
&= [2^4 - b] \bmod 2^4 \\
&= [2^4 - 1 - b + 1] \bmod 2^4 \\
&= [(2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [\text{onescomp}(b) + 1] \bmod 2^4 \\
&= [\text{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

So: $[a - b] \bmod 2^4 = [a + \text{twoscomp}(b)] \bmod 2^4$

$$
\begin{aligned}
& [a - b] \bmod 2^4 \\
&= [a + 2^4 - b] \bmod 2^4 \\
&= [a + 2^4 - 1 - b + 1] \bmod 2^4 \\
&= [a + (2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [a + \text{onescomp}(b) + 1] \bmod 2^4 \\
&= [a + \text{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

Pithier Rationale: Math

Ring theory.

If $n > 0$, $\mathbb{Z}/(n)$ is a finite commutative ring, with properties:

$$\overline{a}_n + \overline{b}_n = \overline{(a+b)}_n ; \overline{a}_n - \overline{b}_n = \overline{(a-b)}_n ; \overline{a}_n \overline{b}_n = \overline{(ab)}_n$$

Shifting Signed Integers

Bitwise left shift (<< in C): fill on right with zeros

`3 << 1 ⇒ 6`

0011_B 0110_B

`-3 << 1 ⇒ -6`

1101_B 1010_B

`-3 << 2 ⇒ 4`

1101_B 0100_B

What is the effect arithmetically?

Results are mod 2^4

Bitwise right shift: fill on left with ???

Shifting Signed Integers (cont.)

Bitwise arithmetic right shift: fill on left with sign bit

```
6 >> 1 ⇒ 3
```

0110_B 0011_B

```
-6 >> 1 ⇒ -3
```

1010_B 1101_B

What is the effect arithmetically?

Bitwise logical right shift: fill on left with zeros

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => 5
```

1010_B 0101_B

What is the effect arithmetically???

In C, right shift (>>) could be logical or arithmetic

- Not specified by standard (happens to be arithmetic on armlab)
- Best to avoid shifting signed integers

Other Operations on Signed Ints

Bitwise NOT (~ in C)

- Same as with unsigned ints

Bitwise AND (& in C)

- Same as with unsigned ints

Bitwise OR: (| in C)

- Same as with unsigned ints

Bitwise exclusive OR (^ in C)

- Same as with unsigned ints

Best to avoid with signed integers

Special-Purpose Assignment

Issue: Should C provide tailored assignment operators?

Thought process

- The construct a = b + c is flexible
- The construct i = i + c is somewhat common
- The construct i = i + 1 is very common
- Special-purpose operators make code more expressive
  - Might reduce some errors
  - May complicate the language and compiler

Decisions

- Introduce += operator to do things like i += c
- Extend to -= *= /= ~= &= |= ^= <<= >>=
- Special-case increment and decrement: i++ i--
- Provide both pre- and post-inc/dec: x = ++i; y = i++;

Plusplus Playfulness

Q: What are i and j set to in the following code?

```
i = 5;
j = i++;
j += ++i;
```

A. 5, 7

B. 7, 5

C. 7, 11

D. 7, 12

E. 7, 13

D

	j	i
`i=5;`	?	5
`j = i++;`	5	6
`j += ++i;`	12	7

Incremental Iffiness

Q: What does the following code print?

```
int i = 1;
switch (i++) {
   case 1: printf("%d", ++i);
   case 2: printf("%d", i++);
}
```

A. 1
B. 2
C. 3
D. 22
E. 33

E!

i++ increments i to 2, but evaluates to i's old value: 1

So switch on 1, not 2!

++i evaluates to new value, so 3 is printed

FALL THROUGH GOTCHA!

i++ evaluates to old value, so 3 is printed again

Incremental Iteration

Q: What does the following code print?

```
int i = 1;
switch (i=i++) {
   case 1: printf("%d", ++i);
   case 2: printf("%d", i++);
}
```

A. 1
B. 2
C. 3
D. 22
E. 33

D

i++ increments i to 2, but evaluates to i's old value: 1

i = 1 overwrites our just-incremented 2 back to 1

... switch on 1, so now continue into case 1 with i as 1, so we end up printing 22.

sizeof Operator

Issue: How to determine the sizes of data?

Thought process

- The sizes of most primitive types are un- or under-specified
- Provide a way to find size of a given variable programmatically

Decisions

- Provide a sizeof operator
  - Applied at compile-time
  - Operand can be a data type
  - Operand can be an expression, from which the compiler infers a data type

Examples, on armlab using gcc217

- sizeof(int) evaluates to 4
- sizeof(i) – where i is a variable of type int – evaluates to 4

iClicker Question

Q: What is the value of the following sizeof expression on the armlab machines?

```
int i = 1;

sizeof(i + 2L)
```

A. 3

B. 4

C. 8

D. 12

E. error

C

Promote i to long, add 1L + 2L.

Result, 3L, is a long.

longs are 8 bytes on armlab.

LOGICAL TYPES

Logical Data Types

- No separate logical or Boolean data type
- Represent logical data using type char or int
  - Or any primitive type! :/
- Conventions:
  - Statements (if, while, etc.) use $0 \Rightarrow$ FALSE, $\neq 0 \Rightarrow$ TRUE
  - Relational operators (<, >, etc.) and logical operators (!, &&, ||) produce the result 0 or 1, specifically

Logical Data Type Shortcuts

Using integers to represent logical data permits shortcuts

```
…
int i;
…
if (i)  /* same as (i != 0) */
   statement1;
else
   statement2;
…
```

It also permits some really bad code…

```
i = (1 != 2) + (3 > 4);
```

Logical Data Type Dangers

The lack of a logical data type hampers compiler's ability to detect some errors

```
…
int i;
…
i = 0;
…
if (i = 5)
   statement1;
…
```

What happens in Java?

What happens in C?

Logical vs. Bitwise Ops

Logical AND (&&) vs. bitwise AND (&)

- 2 (TRUE) && 1 (TRUE) => 1 (TRUE)

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
   && 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      1  00000000 00000000 00000000 00000001
```

- 2 (TRUE) & 1 (TRUE) => 0 (FALSE)

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
    & 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      0  00000000 00000000 00000000 00000000
```

Implication:

- Use logical AND to control flow of logic
- Use bitwise AND only when doing bit-level manipulation
- Same for OR and NOT

Agenda

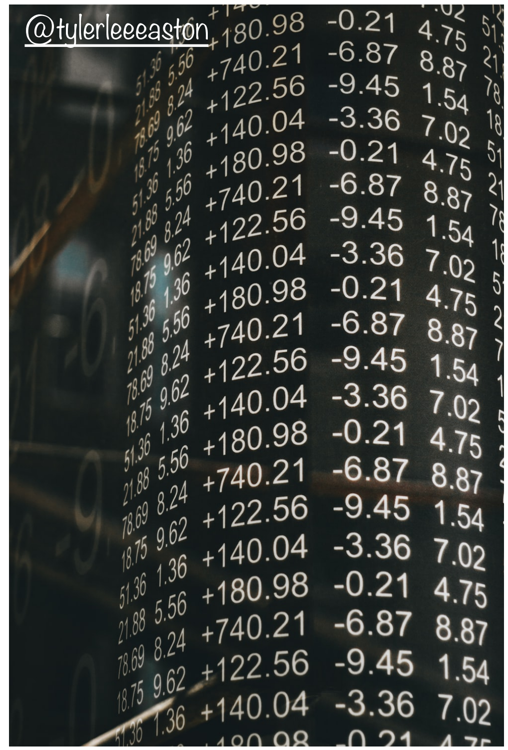
Thus far:

- Integer types in C
- Finite representation of unsigned integers
- Finite representation of signed integers
- Logical types (or lack thereof) in C

Up next:

- Finite representation of rational (floating-point) numbers

FLOATING POINT



Rational Numbers

Mathematics

- A rational number is one that can be expressed as the ratio of two integers
- Unbounded range and precision

Computer science

- Finite range and precision
- Approximate using floating point number

Floating Point Numbers

Like scientific notation: e.g., c is

$$2.99792458 \times 10^{8} \text{ m/s}$$

This has the form

$$(\text{multiplier}) \times (\text{base})^{(\text{power})}$$

In the computer,

- Multiplier is called mantissa
- Base is almost always 2
- Power is called exponent

Floating-Point Data Types

C specifies:

- Three floating-point data types:
  float, double, and long double
- Sizes unspecified, but constrained:
- sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

On ArmLab (and on pretty much any 21st-century computer using the IEEE standard)

- float: 4 bytes
- double: 8 bytes

On ArmLab (but varying across architectures)

- long double: 16 bytes

Floating-Point Literals

How to write a floating-point number?

- Either fixed-point or “scientific” notation
- Any literal that contains decimal point or "E" is floating-point
- The default floating-point type is double
- Append "F" to indicate float
- Append "L" to indicate long double

Examples

- double: 123.456, 1E-2, -1.23456E4
- float: 123.456F, 1E-2F, -1.23456E4F
- long double: 123.456L, 1E-2L, -1.23456E4L

IEEE Floating Point Representation

Common finite representation: IEEE floating point

- More precisely: ISO/IEEE 754 standard

Using 32 bits (type **`float`** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbbbb

Using 64 bits (type **`double`** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

When was floating-point invented?

mantissa (noun): decimal part of a logarithm, 1865, ←Answer: long before computers!
from Latin mantisa "a worthless addition, makeweight"

COMMON LOGARITHMS $\log_{10} x$

x	0	1	2	3	4	5	6	7	8	9	Δ_m +	1	2	3
50	·6990	6998	7007	7016	7024	7033	7042	7050	7059	7067	9	1	2	3
51	·7076	7084	7093	7101	7110	7118	7126	7135	7143	7152	8	1	2	2
52	·7160	7168	7177	7185	7193	7202	7210	7218	7226	7235	8	1	2	2
53	·7243	7251	7259	7267	7275	7284	7292	7300	7308	7316	8	1	2	2
54	·7324	7332	7340	7348	7356	7364	7372	7380	7388	7396	8	1	2	2
55	·7404	7412	7419	7427	7435	7443	7451	7459	7466	7474	8	1	2	2

Floating Point Example

11000001110110110000000000000000

32-bit representation

Sign (1 bit):

- 1 ⇒ negative

Exponent (8 bits):

- $10000011_B = 131$
- $131 - 127 = 4$

Mantissa (23 bits):

- $1.10110110000000000000000_B$
- $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5})+$
  $(1*2^{-6})+(1*2^{-7}) +(0*2^{-...})= 1.7109375$

Number:

- $-1.7109375 * 2^4 = -27.375$

Floating Point Consequences

“Machine epsilon”: smallest positive number you can add to 1.0 and get something other than 1.0

For float: $\varepsilon \approx 10–7$

- No such number as 1.000000001
- Rule of thumb: “almost 7 digits of precision”

For double: $\varepsilon \approx 2 \times 10–16$

- Rule of thumb: “not quite 16 digits of precision”

These are all relative numbers

Floating Point Consequences, cont

Just as decimal number system can represent only some rational numbers with finite digit count…

- Example: 1/3 cannot be represented

Decimal Approx	Rational Value
.3	3/10
.33	33/100
.333	333/1000
...	

Binary number system can represent only some rational numbers with finite digit count

- Example: 1/5 cannot be represented

Binary Approx	Rational Value
0.0	0/2
0.01	1/4
0.010	2/8
0.0011	3/16
0.00110	6/32
0.001101	13/64
0.0011010	26/128
0.00110011	51/256
...	

Beware of round-off error

- Error resulting from inexact representation
- Can accumulate
- Be careful when comparing two floating-point numbers for equality

Floating away …

What does the following code print?

```
double sum = 0.0;
double i;
for (i = 0.0; i != 10.0; i++)
   sum += 0.1;
if (sum == 1.0)
   printf("All good!\n");
else
   printf("Yikes!\n");
```

A. All good!

B. Yikes!

C. (Infinite loop)

D. (Compilation error)

B: Yikes!

… loop terminates, because we can represent 10.0 exactly by adding 1.0 at a time.

… but sum isn't 1.0 because we can't represent 1.0 exactly by adding 0.1 at a time.

Summary

Integer types in C

Finite representation of unsigned integers

Finite representation of signed integers

Logical types in C (or lack thereof)

Floating point types in C

Finite representation of rational (floating-point) numbers

Essential for proper understanding of

- C primitive data types
- Assembly language
- Machine language