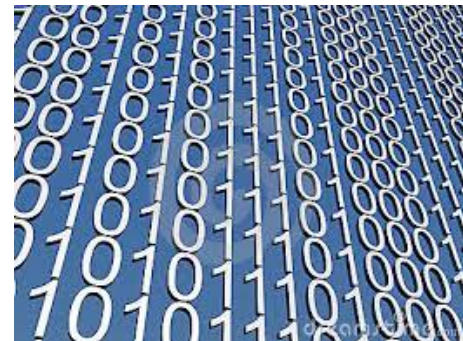




# Number Systems and Number Representation

**Q:** Why do computer programmers  
confuse Christmas and Halloween?

**A:** Because  $25 \text{ Dec} = 31 \text{ Oct}$





# Goals of this Lecture

Help you learn (or refresh your memory) about:

- The binary, hexadecimal, and octal number systems
- Finite representation of unsigned integers
- Finite representation of signed integers
- Finite representation of rational (floating-point) numbers

Why?

- A power programmer must know number systems and data representation to fully understand C's **primitive data types**

Primitive values and  
the operations on them

# Agenda



## Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational (floating-point) numbers

# The Decimal Number System



## Name

- “decem” (Latin)  $\Rightarrow$  ten

## Characteristics

- Ten symbols
  - 0 1 2 3 4 5 6 7 8 9
- Positional
  - $2945 \neq 2495$
  - $2945 = (2 \cdot 10^3) + (9 \cdot 10^2) + (4 \cdot 10^1) + (5 \cdot 10^0)$

(Most) people use the decimal number system

Why?

# The Binary Number System



## binary

*adjective*: being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal.  
From Late Latin *bīnārius* (“consisting of two”).

## Characteristics

- Two symbols
  - 0 1
- Positional
  - $1010_B \neq 1100_B$

Most (digital) computers use the binary number system

## Terminology

- **Bit**: a binary digit
- **Byte**: (typically) 8 bits
- **Nibble (or nybble)**: 4 bits

Why?

# Decimal-Binary Equivalence



<u>Decimal</u>	<u>Binary</u>
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

<u>Decimal</u>	<u>Binary</u>
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111
...	...



# Decimal-Binary Conversion

Binary to decimal: expand using positional notation

$$\begin{aligned} 100101_B &= (1 \cdot 2^5) + (0 \cdot 2^4) + (0 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= 32 + 0 + 0 + 4 + 0 + 1 \\ &= 37 \end{aligned}$$

Most-significant  
bit (msb)

Least-significant  
bit (lsb)

# ~~Integer~~ Decimal-Binary Conversion



~~Integer~~

Binary to ~~decimal~~: expand using positional notation

$$\begin{aligned} 100101_B &= (1 \cdot 2^5) + (0 \cdot 2^4) + (0 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= 32 + 0 + 0 + 4 + 0 + 1 \\ &= 37 \end{aligned}$$

These are integers

They exist as their pure selves  
no matter how we might choose  
to *represent* them with our  
fingers or toes





# Integer-Binary Conversion

## Integer to binary: do the reverse

- Determine largest power of  $2 \leq$  number; write template

$$37 = (? * 2^5) + (? * 2^4) + (? * 2^3) + (? * 2^2) + (? * 2^1) + (? * 2^0)$$

- Fill in template

$$37 = (1 * 2^5) + (0 * 2^4) + (0 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$$

-32  
5  
-4  
1  
-1  
0

100101<sub>B</sub>



# Integer-Binary Conversion

## Integer to binary shortcut

- Repeatedly divide by 2, consider remainder

37	/	2	=	18	R	1
18	/	2	=	9	R	0
9	/	2	=	4	R	1
4	/	2	=	2	R	0
2	/	2	=	1	R	0
1	/	2	=	0	R	1



Read from bottom  
to top:  $100101_B$

# The Hexadecimal Number System



## Name

- “hexa” (Greek)  $\Rightarrow$  six
- “decem” (Latin)  $\Rightarrow$  ten

## Characteristics

- Sixteen symbols
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Positional
  - $A13D_H \neq 3DA1_H$

Computer programmers often use hexadecimal

- In C: `0x` prefix (`0xA13D`, etc.)

Why?

# Decimal-Hexadecimal Equivalence



<u>Decimal</u>	<u>Hex</u>
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

<u>Decimal</u>	<u>Hex</u>
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F

<u>Decimal</u>	<u>Hex</u>
32	20
33	21
34	22
35	23
36	24
37	25
38	26
39	27
40	28
41	29
42	2A
43	2B
44	2C
45	2D
46	2E
47	2F
...	...

# Integer-Hexadecimal Conversion



Hexadecimal to integer: expand using positional notation

$$\begin{aligned} 25_{\text{H}} &= (2 \cdot 16^1) + (5 \cdot 16^0) \\ &= 32 + 5 \\ &= 37 \end{aligned}$$

Integer to hexadecimal: use the shortcut

$$\begin{aligned} 37 / 16 &= 2 \text{ R } 5 \\ 2 / 16 &= 0 \text{ R } 2 \end{aligned}$$



Read from bottom  
to top:  $25_{\text{H}}$



# Binary-Hexadecimal Conversion

Observation:  $16^1 = 2^4$

- Every 1 hexadecimal digit corresponds to 4 binary digits

## Binary to hexadecimal

1010000100111101 <sub>B</sub>
A 1 3 D <sub>H</sub>

Digit count in binary number  
not a multiple of 4  $\Rightarrow$   
pad with zeros on left

## Hexadecimal to binary

A 1 3 D <sub>H</sub>
1010000100111101 <sub>B</sub>

Discard leading zeros  
from binary number if  
appropriate

Is it clear why programmers  
often use hexadecimal?

# iClicker Question

Q: Convert binary 101010 into decimal and hex

- A. 21 decimal, 1A hex
- B. 42 decimal, 2A hex
- C. 48 decimal, 32 hex
- D. 55 decimal, 4G hex

# The Octal Number System



## Name

- “octo” (Latin)  $\Rightarrow$  eight

## Characteristics

- Eight symbols
  - 0 1 2 3 4 5 6 7
- Positional
  - $1743_8 \neq 7314_8$



Computer programmers often use octal (so does Mickey!)

- In C: 0 prefix (01743, etc.)





# Agenda



## Number Systems

**Finite representation of unsigned integers**

Finite representation of signed integers

Finite representation of rational (floating-point) numbers

# Integral Types in Java vs. C



	Java	C
Unsigned types	<code>char // 16 bits</code>	<code>unsigned char /* 8 bits */</code> <code>unsigned short</code> <code>unsigned (int)</code> <code>unsigned long</code>
Signed types	<code>byte // 8 bits</code> <code>short // 16 bits</code> <code>int // 32 bits</code> <code>long // 64 bits</code>	<code>signed char /* Note 2 */</code> <code>(signed) short</code> <code>(signed) int</code> <code>(signed) long</code>
Floating-point types	<code>float // 32 bits</code> <code>double // 64 bits</code>	<code>float</code> <code>double</code> <code>long double</code>

1. Not guaranteed by C, but on `courselab`, `char` = 8 bits, `short` = 16 bits, `int` = 32 bits, `long` = 64 bits, `float` = 32 bits, `double` = 64 bits
2. Not guaranteed by C, but on `courselab`, `char` is signed

To understand C, must consider representation of both unsigned and signed integers

# Representing Unsigned Integers



## Mathematics

- Range is 0 to  $\infty$

## Computer programming

- Range limited by computer's **word** size
- Word size is  $n$  bits  $\Rightarrow$  range is 0 to  $2^n - 1$
- Exceed range  $\Rightarrow$  **overflow**

## Typical computers today

- $n = 32$  or  $64$ , so range is 0 to  $2^{32} - 1$  or  $2^{64} - 1$  (huge!)

## Pretend computer

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

## Hereafter, assume word size = 4

- All points generalize to word size = 64, word size =  $n$

# Representing Unsigned Integers



On pretend computer

<u>Unsigned</u> <u>Integer</u>	<u>Rep</u>
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		0011 <sub>B</sub>
+	10	+	1010 <sub>B</sub>
--		--	----
	13		1101 <sub>B</sub>

			1
	7		0111 <sub>B</sub>
+	10	+	1010 <sub>B</sub>
--		--	----
	1		0001 <sub>B</sub>

Start at right column  
Proceed leftward  
Carry 1 when necessary

Beware of overflow

Results are mod  $2^4$

How would you  
detect overflow  
programmatically?



# Subtracting Unsigned Integers

## Subtraction

			<b>111</b>
10		1010 <sub>B</sub>	
- 7	-	0111 <sub>B</sub>	
--		----	
3		0011 <sub>B</sub>	

			<b>1</b>
3		0011 <sub>B</sub>	
- 10	-	1010 <sub>B</sub>	
--		----	
9		1001 <sub>B</sub>	

Start at right column  
Proceed leftward  
Borrow when necessary

Beware of overflow

Results are mod  $2^4$

How would you detect overflow programmatically?



# Shifting Unsigned Integers

Bitwise right shift ( $\gg$  in C): fill on left with zeros

$10 \gg 1 \Rightarrow 5$   
 $1010_B \quad 0101_B$

$10 \gg 2 \Rightarrow 2$   
 $1010_B \quad 0010_B$

What is the effect arithmetically?  
(No fair looking ahead)

Bitwise left shift ( $\ll$  in C): fill on right with zeros

$5 \ll 1 \Rightarrow 10$   
 $0101_B \quad 1010_B$

$3 \ll 2 \Rightarrow 12$   
 $0011_B \quad 1100_B$

What is the effect arithmetically?  
(No fair looking ahead)

Results are mod  $2^4$

# Other Operations on Unsigned Ints



## Bitwise NOT (~ in C)

- Flip each bit

$\sim 10 \Rightarrow 5$

$1010_{\text{B}}$     $0101_{\text{B}}$

## Bitwise AND (& in C)

- Logical AND corresponding bits

10	$1010_{\text{B}}$
& 7	$\& 0111_{\text{B}}$
--	----
2	$0010_{\text{B}}$

Useful for setting  
selected bits to 0



# Other Operations on Unsigned Ints



## Bitwise OR: (| in C)

- Logical OR corresponding bits

10	1010 <sub>B</sub>
1	0001 <sub>B</sub>
--	----
11	1011 <sub>B</sub>

Useful for setting  
selected bits to 1

## Bitwise exclusive OR (^ in C)

- Logical exclusive OR corresponding bits

10	1010 <sub>B</sub>
^ 10	^ 1010 <sub>B</sub>
--	----
0	0000 <sub>B</sub>

$x \wedge x$  sets  
all bits to 0

## iClicker Question

Q: How do you set bit “n” (counting lsb=0) of unsigned variable “u” to zero?

A.  $u \&= (0 \ll n);$

B.  $u |= (1 \ll n);$

C.  $u \&= \sim(1 \ll n);$

D.  $u |= \sim(1 \ll n);$

E.  $u = \sim u \wedge (1 \ll n);$

# Aside: Using Bitwise Ops for Arith



Can use  $\ll$ ,  $\gg$ , and  $\&$  to do some arithmetic efficiently

$$x * 2^y == x \ll y$$

$$\bullet 3 * 4 = 3 * 2^2 = 3 \ll 2 \Rightarrow 12$$

$$x / 2^y == x \gg y$$

$$\bullet 13 / 4 = 13 / 2^2 = 13 \gg 2 \Rightarrow 3$$

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

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

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

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

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

13	1101 <sub>B</sub>
& 3	& 0011 <sub>B</sub>
--	----
1	0001 <sub>B</sub>

Many compilers will  
do these transformations  
automatically!



# Aside: Example C Program

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    unsigned int n;
    unsigned int count = 0;
    printf("Enter an unsigned integer: ");
    if (scanf("%u", &n) != 1)
    {
        fprintf(stderr, "Error: Expect unsigned int.\n");
        exit(EXIT_FAILURE);
    }
    while (n > 0)
    {
        count += (n & 1);
        n = n >> 1;
    }
    printf("%u\n", count);
    return 0;
}
```

What does it write?

How could you express this more succinctly?

# Agenda



Number Systems

Finite representation of unsigned integers

**Finite representation of signed integers**

Finite representation of rational (floating-point) numbers



# Sign-Magnitude

<u>Integer</u>	<u>Rep</u>
-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  $\Rightarrow$  positive

1  $\Rightarrow$  negative

Remaining bits indicate magnitude

$$1101_B = -101_B = -5$$

$$0101_B = 101_B = 5$$



# Sign-Magnitude (cont.)

<u>Integer</u>	<u>Rep</u>
-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

$\text{neg}(x) = \text{flip high order bit of } x$

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

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

## Pros and cons

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



# Ones' Complement

<u>Integer</u>	<u>Rep</u>
-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 -7

$$\begin{aligned}1010_B &= (1 * -7) + (0 * 4) + (1 * 2) + (0 * 1) \\ &= -5\end{aligned}$$

$$\begin{aligned}0010_B &= (0 * -7) + (0 * 4) + (1 * 2) + (0 * 1) \\ &= 2\end{aligned}$$





# Ones' Complement (cont.)

<u>Integer</u>	<u>Rep</u>
-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

## Computing negative

$$\text{neg}(x) = \sim x$$

$$\text{neg}(0101_{\text{B}}) = 1010_{\text{B}}$$

$$\text{neg}(1010_{\text{B}}) = 0101_{\text{B}}$$

Similar pros and cons to  
sign-magnitude

# Two's Complement



<u>Integer</u>	<u>Rep</u>
-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 -8

$$\begin{aligned}1010_B &= (1 * -8) + (0 * 4) + (1 * 2) + (0 * 1) \\ &= -6\end{aligned}$$

$$\begin{aligned}0010_B &= (0 * -8) + (0 * 4) + (1 * 2) + (0 * 1) \\ &= 2\end{aligned}$$



# Two's Complement (cont.)

<u>Integer</u>	<u>Rep</u>
-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

$$\text{neg}(x) = \sim x + 1$$

$$\text{neg}(x) = \text{onescomp}(x) + 1$$

$$\text{neg}(0101_{\text{B}}) = 1010_{\text{B}} + 1 = 1011_{\text{B}}$$

$$\text{neg}(1011_{\text{B}}) = 0100_{\text{B}} + 1 = 0101_{\text{B}}$$

## Pros and cons

- not symmetric

+ one representation of zero

+ same algorithm adds unsigned numbers  
or signed numbers

# Two's Complement (cont.)



Almost all computers today use two's complement to represent signed integers

- Arithmetic is easy!

Is it after 1980?  
OK, then we're surely  
two's complement



Hereafter, assume two's complement



# Adding Signed Integers

pos + pos

		<b>11</b>
3		0011 <sub>B</sub>
+ 3	+	0011 <sub>B</sub>
--		----
6		0110 <sub>B</sub>

pos + pos (overflow)

		<b>111</b>
7		0111 <sub>B</sub>
+ 1	+	0001 <sub>B</sub>
--		----
-8		1000 <sub>B</sub>

pos + neg

		<b>1111</b>
3		0011 <sub>B</sub>
+ -1	+	1111 <sub>B</sub>
--		----
2		<b>10010<sub>B</sub></b>

How would you detect overflow programmatically?

neg + neg

		<b>11</b>
-3		1101 <sub>B</sub>
+ -2	+	1110 <sub>B</sub>
--		----
-5		<b>11011<sub>B</sub></b>

neg + neg (overflow)

		<b>1 1</b>
-6		1010 <sub>B</sub>
+ -5	+	1011 <sub>B</sub>
--		----
5		<b>10101<sub>B</sub></b>

# Subtracting Signed Integers



Perform subtraction  
with borrows

or

Compute two's comp  
and add

		1
		22
3	0011 <sub>B</sub>	
- 4	- 0100 <sub>B</sub>	
--	----	
-1	1111 <sub>B</sub>	



3	0011 <sub>B</sub>
+ -4	+ 1100 <sub>B</sub>
--	----
-1	1111 <sub>B</sub>

-5	1011 <sub>B</sub>
- 2	- 0010 <sub>B</sub>
--	----
-7	1001 <sub>B</sub>



	111
-5	1011
+ -2	+ 1110
--	----
-7	11001



# Negating Signed Ints: Math

**Question:** Why does two's comp arithmetic work?

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

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

See Bryant & O'Hallaron book for much more info

# Subtracting Signed Ints: Math



And so:

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

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

See Bryant & O' Hallaron book for much more info





# Shifting Signed Integers

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

$$\begin{array}{l} 3 \ll 1 \Rightarrow 6 \\ 0011_{\text{B}} \quad 0110_{\text{B}} \end{array}$$

$$\begin{array}{l} -3 \ll 1 \Rightarrow -6 \\ 1101_{\text{B}} \quad -1010_{\text{B}} \end{array}$$

What is the effect arithmetically?

Bitwise **arithmetic** right shift: fill on left **with sign bit**

$$\begin{array}{l} 6 \gg^* 1 \Rightarrow 3 \\ 0110_{\text{B}} \quad 0011_{\text{B}} \end{array}$$

$$\begin{array}{l} -6 \gg^* 1 \Rightarrow -3 \\ 1010_{\text{B}} \quad 1101_{\text{B}} \end{array}$$

What is the effect arithmetically?

Results are mod  $2^4$

\* The C language does not provide an arithmetic right shift operator, although the hardware (machine language) does.



# Shifting Signed Integers (cont.)

Bitwise **logical** right shift: fill on left **with zeros**

6 >> 1 => 3  
0110<sub>B</sub>      0011<sub>B</sub>

-6 >> 1 => 5  
1010<sub>B</sub>      0101<sub>B</sub>

What is the effect arithmetically???

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

- Not specified by C90 standard
- Compiler designer decides (typically it's logical right shift)

**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**

# Agenda



Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

**Finite representation of rational (floating-point) numbers**

# 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

# 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  $1.bbb$



# Floating Point Example

Sign (1 bit):

- 1  $\Rightarrow$  negative

11000001110110110000000000000000

32-bit representation

Exponent (8 bits):

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

Fraction (23 bits): also called "mantissa"

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

Number:

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



# When was floating-point invented?



Answer: long before computers!

**mantissa**

*noun*

decimal part of a logarithm, 1865, from Latin *mantisa* "a worthless addition, makeweight," perhaps a Gaulish word introduced into Latin via Etruscan (cf. Old Irish *meit*, Welsh *maint* "size").

COMMON LOGARITHMS  $\log_{10} x$

x	0	1	2	3	4	5	6	7	8	9	$\Delta_m$	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
56	.7482	7490	7497	7505	7513	7520	7528	7536	7543	7551	8	1	2	2
57	.7559	7566	7574	7582	7589	7597	7604	7612	7619	7627	8	1	2	2
58	.7634	7642	7649	7657	7664	7672	7679	7686	7694	7701	8	1	2	2
59	.7709	7716	7723	7731	7738	7745	7752	7760	7767	7774	7	1	1	2

# 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



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

- Example:  $1/3$  **cannot** be represented

<u>Decimal</u> <u>Approx</u>	<u>Rational</u> <u>Value</u>
.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

<u>Binary</u> <u>Approx</u>	<u>Rational</u> <u>Value</u>
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 **roundoff error**

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

# ▶ iClicker Question

Q: What does the following code print?

```
double sum = 0.0;
int i;
for (i = 0; i < 10; i++)
    sum += 0.1;
if (sum == 1.0)
    printf("All good!\n");
else
    printf("Yikes!\n");
```

- A. All good!
- B. Yikes!
- C. Code crashes
- D. Code enters an infinite loop

# Summary



The binary, hexadecimal, and octal number systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational (floating-point) numbers

Essential for proper understanding of

- C primitive data types
- Assembly language
- Machine language