Princeton University

Computer Science 217: Introduction to Programming Systems

Number Systems and Number Representation

Q: Why do computer programmers confuse Christmas and Halloween?

A: Because 25 Dec = 31 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) ⇒ ten

Characteristics

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

(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

Why?

Terminology

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

Decimal-Binary Equivalence

Decimal	Binary
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

Decimal	Binary
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

```
100101B = (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0)
        =    32  +   0   +   0   +   4   +   0   +   1
        =    37
```

Most-significant bit (msb)

Least-significant bit (lsb)

~~Decimal~~ Integer-Binary Conversion

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

```
100101B = (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0)
        =    32  +   0   +   0   +   4   +   0   +   1
        =    37
```

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 ≤ 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                        100101B
 -1
  0
```


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) ⇒ six
- "decem" (Latin) ⇒ 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

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	0	16	10	32	20
1	1	17	11	33	21
2	2	18	12	34	22
3	3	19	13	35	23
4	4	20	14	36	24
5	5	21	15	37	25
6	6	22	16	38	26
7	7	23	17	39	27
8	8	24	18	40	28
9	9	25	19	41	29
10	A	26	1A	42	2A
11	B	27	1B	43	2B
12	C	28	1C	44	2C
13	D	29	1D	45	2D
14	E	30	1E	46	2E
15	F	31	1F	47	2F
				...	...

Integer-Hexadecimal Conversion

Hexadecimal to integer: expand using positional notation

$$
\begin{aligned}
25_H &= (2*16^1) + (5*16^0) \\
&= 32 + 5 \\
&= 37
\end{aligned}
$$

Integer to hexadecimal: use the shortcut

```
37 / 16 = 2 R 5
 2 / 16 = 0 R 2
```

↑ Read from bottom to top: 25_H

Binary-Hexadecimal Conversion

Observation: $16^1 = 2^4$

- Every 1 hexadecimal digit corresponds to 4 binary digits

Binary to hexadecimal

$$
\begin{array}{l}
1010000100111101_B \\
\quad A \quad 1 \quad 3 \quad D_H
\end{array}
$$

Digit count in binary number not a multiple of 4 ⇒ pad with zeros on left

Hexadecimal to binary

$$
\begin{array}{l}
\quad A \quad 1 \quad 3 \quad D_H \\
1010000100111101_B
\end{array}
$$

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) ⇒ eight

Characteristics

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

Computer programmers often use octal (so does Mickey!)

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

Why?

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	`char // 16 bits`	`unsigned char /* 8 bits */`
		`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`

Floating-point types	`float // 32 bits`
	`double // 64 bits`
		`float`
		`double`
		`long double`

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 ∞

Computer programming
- Range limited by computer's **word** size
- Word size is n bits ⇒ range is 0 to $2^n – 1$
- Exceed range ⇒ **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

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

```
               1
   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
1010B   0101B

10 >> 2 ⇒ 2
1010B   0010B
```

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

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

```
5 << 1 ⇒ 10
0101B   1010B

3 << 2 ⇒ 12
0011B   1100B
```

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

```
~10 ⇒ 5
1010B  0101B
```

Bitwise AND (& in C)
- Logical AND corresponding bits

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

Useful for setting selected bits to 0

Other Operations on Unsigned Ints

Bitwise OR: (| in C)

- Logical OR corresponding bits

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

Useful for setting selected 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

iClicker Question

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

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

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

C. u &= ~(1 << n);

D. u |= ~(1 << n);

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

Aside: Using Bitwise Ops for Arith

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

x * 2^y == x << y

- 3*4 = 3*2² = 3<<2 ⇒ 12

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

x / 2^y == x >> y

- 13/4 = 13/2² = 13>>2 ⇒ 3

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

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

- 13%4 = 13%2² = 13&(2²-1) = 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!

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

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

  $1101_B = -101_B = -5$

  $0101_B = 101_B = 5$

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

$neg(0101_B) = 1101_B$

$neg(1101_B) = 0101_B$

Pros and cons

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

Ones' Complement

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 -7

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

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

Ones' Complement (cont.)

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

Computing negative

neg(x) = ~x

$neg(0101_B) = 1010_B$

$neg(1010_B) = 0101_B$

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 -8

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

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

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

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

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

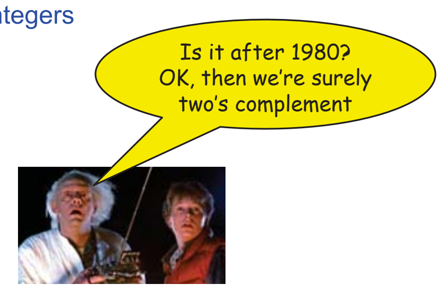
Pros and cons

- not symmetric
+ 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!



Hereafter, assume two's complement

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     10010B
```

How would you detect overflow programmatically?

neg + neg

```
          11
 -3      1101B
+ -2   + 1110B
 --      ----
 -5     11011B
```

neg + neg (overflow)

```
         1 1
 -6      1010B
+ -5   + 1011B
 --      ----
  5     10101B
```


Subtracting Signed Integers

Perform subtraction with borrows — or — Compute two's comp and add

```
          1
         22
  3     0011B
- 4   - 0100B
 --     ----
 -1     1111B
```

→

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

```
 -5     1011B
- 2   - 0010B
 --     ----
 -7     1001B
```

→

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


Negating Signed Ints: Math

Question: Why does two's comp arithmetic work?

Answer: `[-b] mod 2^4 = [twoscomp(b)] mod 2^4`

```
[-b] mod 2^4
= [2^4 - b] mod 2^4
= [2^4 - 1 - b + 1] mod 2^4
= [(2^4 - 1 - b) + 1] mod 2^4
= [onescomp(b) + 1] mod 2^4
= [twoscomp(b)] mod 2^4
```

See Bryant & O'Hallaron book for much more info

Subtracting Signed Ints: Math

And so:

`[a - b] mod 2^4 = [a + twoscomp(b)] mod 2^4`

```
[a - b] mod 2^4
= [a + 2^4 - b] mod 2^4
= [a + 2^4 - 1 - b + 1] mod 2^4
= [a + (2^4 - 1 - b) + 1] mod 2^4
= [a + onescomp(b) + 1] mod 2^4
= [a + twoscomp(b)] mod 2^4
```

See Bryant & O'Hallaron book for much more info

Shifting Signed Integers

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

```
3 << 1 ⇒ 6
0011B    0110B

-3 << 1 ⇒ -6
1101B    -1010B
```

What is the effect arithmetically?

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

```
6 >>* 1 ⇒ 3
0110B    0011B

-6 >>* 1 ⇒ -3
1010B    1101B
```

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
0110B    0011B

-6 >> 1 => 5
1010B    0101B
```

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

2.99792458×10^{8} 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.*bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb*

Floating Point Example

11000001110110110000000000000000

32-bit representation

Sign (1 bit):
- 1 ⇒ negative

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	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
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

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 **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