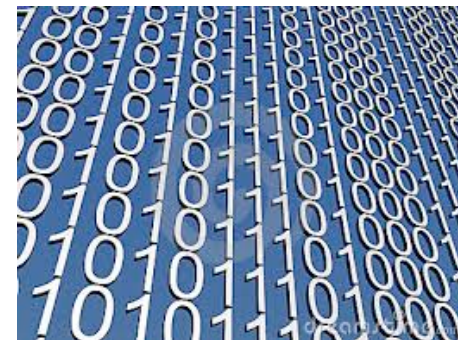




Number Systems and Number Representation





For Your Amusement

Question: Why do computers programmers confuse Christmas and Halloween?

Answer: Because 25 Dec = 31 Oct

-- <http://www.electronicweekly.com>



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 numbers (if time)

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 numbers (if time)



The Decimal Number System

Name

- “decem” (Latin) => ten

Characteristics

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

(Most) people use the decimal number system

Why?



The Binary Number System

Name

- “binarius” (Latin) => 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



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

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



Decimal-Binary Conversion

Decimal to binary: do the reverse

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

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

- Fill in template

$$\begin{array}{r} 37 = (1 * 2^5) + (0 * 2^4) + (0 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) \\ \hline -32 \\ 5 \\ \hline -4 \\ 1 \\ \hline -1 \\ 0 \end{array} \qquad 100101_B$$



Decimal-Binary Conversion

Decimal 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 the hexadecimal number system

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
...	...



Decimal-Hexadecimal Conversion

Hexadecimal to decimal: expand using positional notation

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

Decimal 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_{H}



Binary-Hexadecimal Conversion

Observation: $16^1 = 2^4$

- Every 1 hexadecimal digit corresponds to 4 binary digits

Binary to hexadecimal

1010000100111101 _B
A 1 3 D _H

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

Hexadecimal to binary

A 1 3 D _H
1010000100111101 _B

Discard leading zeros
from binary number if
appropriate

Is it clear why programmers
often use hexadecimal?



The Octal Number System

Name

- “octo” (Latin) => eight

Characteristics

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

Computer programmers often use the octal number system

Why?



Decimal-Octal Equivalence

<u>Decimal</u>	<u>Octal</u>
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17

<u>Decimal</u>	<u>Octal</u>
16	20
17	21
18	22
19	23
20	24
21	25
22	26
23	27
24	30
25	31
26	32
27	33
28	34
29	35
30	36
31	37

<u>Decimal</u>	<u>Octal</u>
32	40
33	41
34	42
35	43
36	44
37	45
38	46
39	47
40	50
41	51
42	52
43	53
44	54
45	55
46	56
47	57
...	...



Decimal-Octal Conversion

Octal to decimal: expand using positional notation

$$\begin{aligned} 37_{\text{o}} &= (3 \cdot 8^1) + (7 \cdot 8^0) \\ &= 24 + 7 \\ &= 31 \end{aligned}$$

Decimal to octal: use the shortcut

$$\begin{aligned} 31 / 8 &= 3 \text{ R } 7 \\ 3 / 8 &= 0 \text{ R } 3 \end{aligned}$$



Read from bottom
to top: 37_{o}



Binary-Octal Conversion

Observation: $8^1 = 2^3$

- Every 1 octal digit corresponds to 3 binary digits

Binary to octal

001010000100111101 _B
1 2 0 4 7 5 _O

Digit count in binary number
not a multiple of 3 =>
pad with zeros on left

Octal to binary

1 2 0 4 7 5 _O
001010000100111101 _B

Discard leading zeros
from binary number if
appropriate

Is it clear why programmers
sometimes use octal?



Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)



Unsigned Data Types: Java vs. C

Java has type

- `int`
 - Can represent signed integers

C has type:

- `signed int`
 - Can represent signed integers
- `int`
 - Same as `signed int`
- `unsigned int`
 - Can represent only unsigned integers

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 \Rightarrow range is 0 to $2^n - 1$
- Exceed range \Rightarrow **overflow**

Nobel computers with gcc217

- $n = 32$, so range is 0 to $2^{32} - 1$ (4,294,967,295)

Pretend computer

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

Hereafter, assume word size = 4

- All points generalize to word size = 32, word size = n



Representing Unsigned Integers

On pretend computer

<u>Unsigned 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 _B
+ 10	+	1010 _B
--		----
13		1101 _B

		11
7		0111 _B
+ 10	+	1010 _B
--		----
1		1 0001 _B

Start at right column
Proceed leftward
Carry 1 when necessary

Beware of overflow

Results are mod 2⁴

How would you detect overflow programmatically?



Subtracting Unsigned Integers

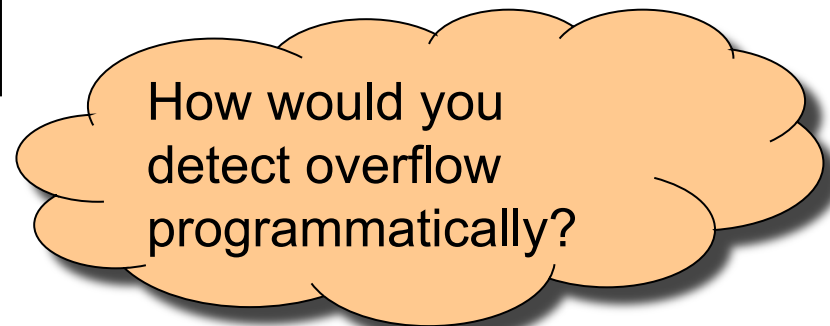
Subtraction

		12	
		0202	
10		1010 _B	
- 7	-	0111 _B	
--		----	
3		0011 _B	

		2	
		0011 _B	
3		0011 _B	
- 10	-	1010 _B	
--		----	
9		1001 _B	

Start at right column
Proceed leftward
Borrow 2 when necessary

Beware of overflow



Results are mod 2^4



Shifting Unsigned Integers

Bitwise right shift (\gg 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 (\ll 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	1010 _B
& 7	& 0111 _B
--	----
2	0010 _B

Useful for setting
selected bits to 0

Other Operations on Unsigned Ints



Bitwise OR: (| in C)

- Logical OR corresponding bits

10	1010 _B
1	0001 _B
--	----
11	1011 _B

Useful for setting
selected bits to 1

Bitwise exclusive OR (^ in C)

- Logical exclusive OR corresponding bits

10	1010 _B
^ 10	^ 1010 _B
--	----
0	0000 _B

$x \wedge x$ sets
all bits to 0



Aside: Using Bitwise Ops for Arith

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

$$\mathbf{x} * 2^y == \mathbf{x} \ll y$$

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

$$\mathbf{x} / 2^y == \mathbf{x} \gg y$$

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

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

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

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

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

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

13	1101 _B
& 3	& 0011 _B
--	----
1	0001 _B



Aside: Example C Program

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

What does it write?

How could this be expressed more succinctly?



Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)



Signed 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 => positive

1 => negative

Remaining bits indicate magnitude

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

$$0101_B = 101_B = 5$$



Signed 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 reps of zero



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

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

$$\text{neg}(0101_B) = 1010_B$$

$$\text{neg}(1010_B) = 0101_B$$

Computing negative (alternative)

$$\text{neg}(x) = 1111_B - x$$

$$\begin{aligned} \text{neg}(0101_B) &= 1111_B - 0101_B \\ &= 1010_B \end{aligned}$$

$$\begin{aligned} \text{neg}(1010_B) &= 1111_B - 1010_B \\ &= 0101_B \end{aligned}$$

Pros and cons

+ symmetric

- two reps of zero



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 rep of zero



Two's Complement (cont.)

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

Why?

- Arithmetic is easy
 - Will become clear soon

Hereafter, assume two's complement representation of signed integers



Adding Signed Integers

pos + pos

		11
3		0011 _B
+ 3	+	0011 _B
--		----
6		0110 _B

pos + pos (overflow)

		111
7		0111 _B
+ 1	+	0001 _B
--		----
-8		1000 _B

pos + neg

		1111
3		0011 _B
+ -1	+	1111 _B
--		----
2		10010 _B

How would you detect overflow programmatically?

neg + neg

		11
-3		1101 _B
+ -2	+	1110 _B
--		----
-5		11011 _B

neg + neg (overflow)

		1 1
-6		1010 _B
+ -5	+	1011 _B
--		----
5		10101 _B

Subtracting Signed Integers



Perform subtraction
with borrows

or

Compute two's comp
and add

		1
		22
3		0011 _B
- 4	-	0100 _B
--		----
-1		1111 _B



3		0011 _B
+ -4	+	1100 _B
--		----
-1		1111 _B

-5		1011 _B
- 2	-	0010 _B
--		----
-7		1001 _B



		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:

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

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

See Bryant & O'Hallaron book for much more info



Shifting Signed Integers

Bitwise left shift (\ll 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



Shifting Signed Integers (cont.)

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 C90 standard
- Compiler designer decides

Best to avoid shifting signed integers



Other Operations on Signed Ints

Bitwise NOT (\sim 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 (\wedge 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 numbers (if time)



Rational Numbers

Mathematics

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

Compute science

- Finite range and precision
- Approximate using **floating point** number
 - Binary point “floats” across bits

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 $1.ddddddddddddddddddddddddd$

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 $1.dd$



Floating Point Example

Sign (1 bit):

- 1 => negative

11000001110110110000000000000000

32-bit representation

Exponent (8 bits):

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

Fraction (23 bits):

- $1.10110110000000000000000000000000_B$
- 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:

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



Floating Point Warning

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

- Example: $1/3$

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

- Example: $1/5$

Beware of **roundoff error**

- Error resulting from inexact representation
- Can accumulate

<u>Decimal</u> <u>Approx</u>	<u>Rational</u> <u>Value</u>
.3	$3/10$
.33	$33/100$
.333	$333/1000$
...	

<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	$51/256$
...	



Summary

The binary, hexadecimal, and octal number systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers

Essential for proper understanding of

- C primitive data types
- Assembly language
- Machine language