



北京航空航天大学
BEIHANG UNIVERSITY



程序设计基础

Fundamentals of Programming

北京航空航天大学 程序设计课程组

软件学院 谭火彬

2022年



北京航空航天大学
BEIHANG UNIVERSITY



第三讲 数据处理

Data Processing

- ◆ 二进制与整数编码
- ◆ 位运算
- ◆ 浮点数与数据精度



C语言中的4种基本类型

数据类型		长度（二进制位数） (32/64位操作系统)	长度（二进制位数） (16位操作系统)	格式控制符
int	整型	32	16	%d
char	字符型	8	8	%c
float	单精度浮点型	32	32	%f
double	双精度浮点型	64	64	%lf (输出仍采用%f)

- ◆ C语言所有的数据类型都是建立在这四种基本类型之上的
- ◆ 这四种类型如何在计算机中存储？
 - ✓ 存储的位置：地址
 - ✓ 存储的实现方式：二进制和编码

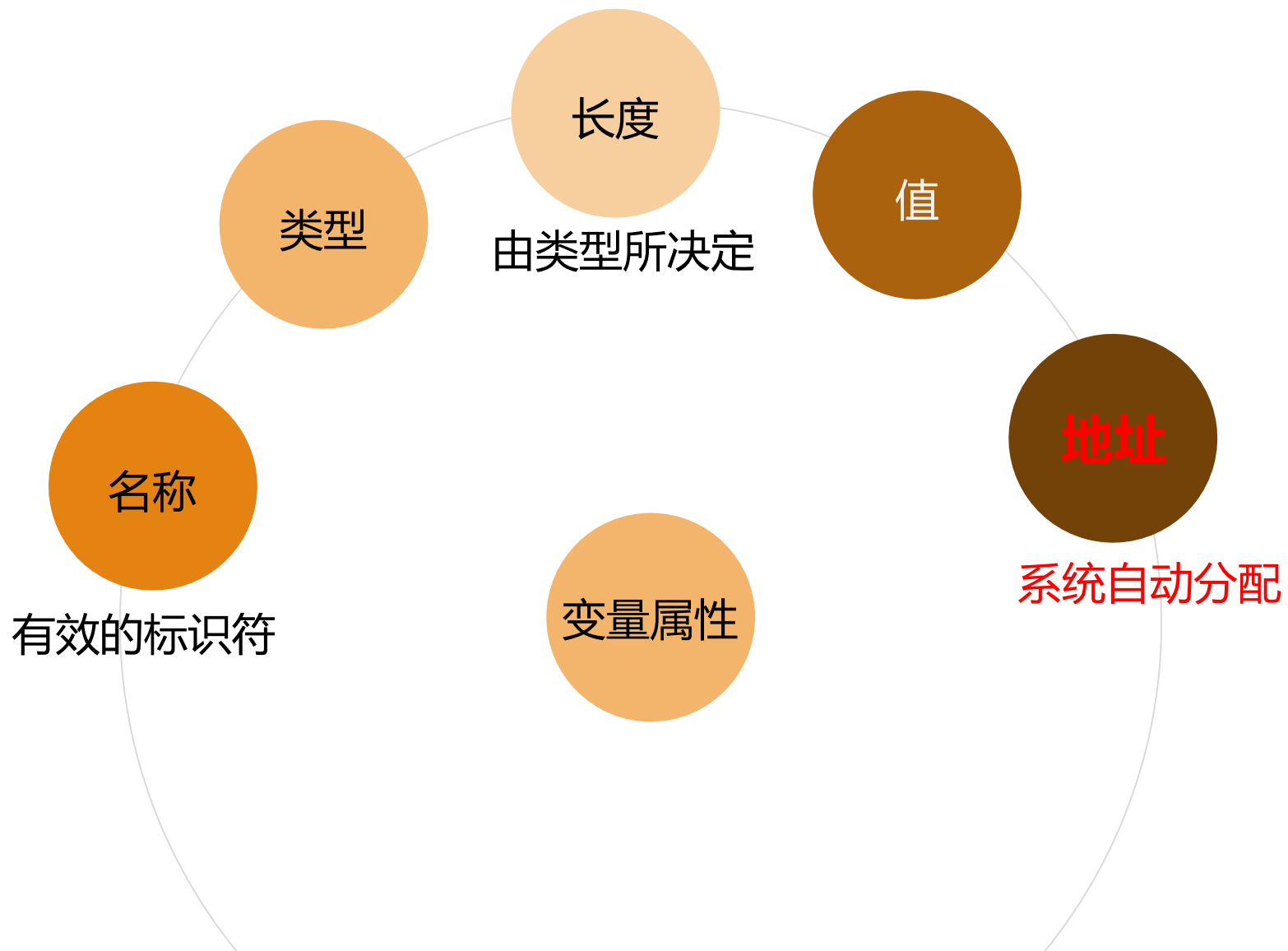


提纲：数据在计算机中的存储和计算

- ◆ 3.1 变量的地址
- ◆ 3.2 从二进制到整数编码
- ◆ 3.3 更多的进制：八进制和十六进制
- ◆ 3.4 基于二进制的位运算
- ◆ 3.5 浮点数的编码与精度问题
- ◆ 3.6 一维数组的存储和应用

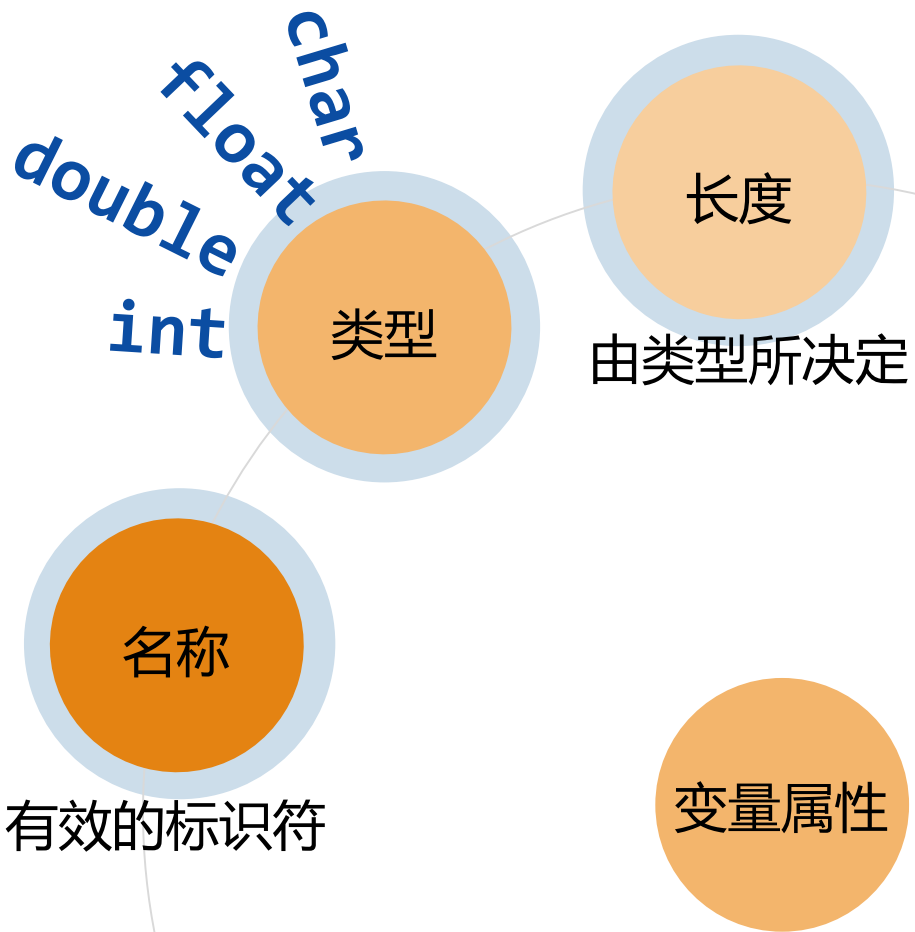


回顾：简单变量的定义





3.1 简单变量在内存中的存储：地址和存储方式



char a;

a (1002)



65 'A'



变量与内存的关系

```
char c;  
short s = 0;  
int a = 55, b = a, sum;  
double d;  
printf("%X\n", &c);  
printf("%X\n", &s);  
printf("%X\n", &a);  
printf("%X\n", &b);  
printf("%X\n", &d);  
//注意，不是printf("%X\n", d);
```

注意有无&的区别

内存 (Memory)

	60FEF8	60FEF9	60FEFA	60FEFB	60FEFC	60FEFD	60FEFE	60FEFF
		d							
60FF00	60FF01	...02	...03	60FF04	...05	...06	...07	60FF08	...09
				b				a	
...0A	...0B	60FF0C	...0D	...0E	60FF0F	...10		
		S			c				

60FF0F
60FF0C
60FF08
60FF04
60FEF8



基本数据类型及其内存存储空间 (32/64位系统)

类型	字节	位	有效数字	取值范围
char	■	8		-128 ~ 127
int	■■■■	32		-2147483648 ~ +2147483647
float	■■■■	32	6~7	$-3.4 \times 10^{-38} \sim 3.4 \times 10^{+38}$
double	■■■■■■■■	64	15~16	$-1.7 \times 10^{-308} \sim 1.7 \times 10^{+308}$
unsigned int	■■■■	32		0 ~ 4294967295
short int	■■	16		-32768 ~ 32767
long int	■■■■	32		-2147483648 ~ +2147483647
long long int	■■■■■■■■	64		$-2^{63} \sim +2^{63}-1$
long double	■■■■■■■■ ■■■■■■■■	128/ 96	18~19	$-1.2 \times 10^{-4932} \sim 1.2 \times 10^{+4932}$



两段有点“奇怪”的代码

```
#include <stdio.h>
int main()
{
    int a, b;
    char sum = 0;
    scanf("%d%d", &a, &b);
    sum = a + b;
    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int a = 625, b = 3;
    printf("%d, %d\n", (a==625), (b==3));
    float x = 0.625, y = 0.3;
    printf("%d, %d", (x==0.625), (y==0.3));
    return 0;
}
```

```
100 100
100 + 100 = -56
```

怪象1: 100+100 不等于 200?

```
1, 1
1, 0
```

说明: 这是dev c下编译运行的结果,
其他环境下可能是 1, 1

怪象2: 0.3 等于 0.3 不成立?



数据在计算机中如何存储和计算？

◆如何将日常的十进制数，存储在计算机**定长字节**中？

◆两种编码实现方式

✓**基于二进制的补码**：整数类型编码

➤int、char类型，及其相应的扩展类型

✓**IEEE754标准**：浮点数类型编码

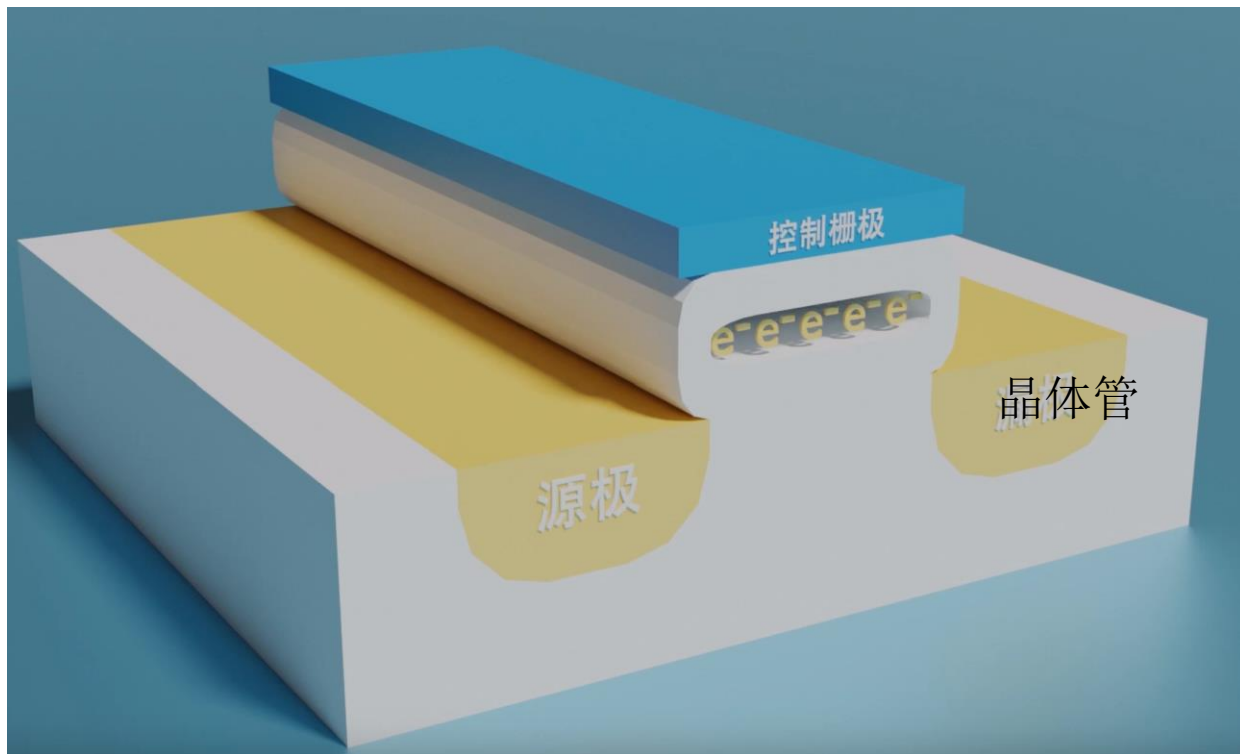
➤float、double类型，及其相应的扩展类型



3.2 从二进制到整数编码

0, 1, 10, 11, 100, 101...

二进制：满2进1



7, 8, 9, 10, 11, 12, 13...

十进制：满10进1



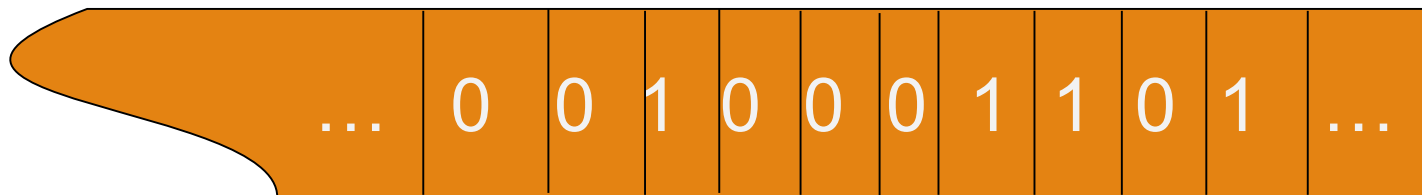


二进制

◆二进制

✓数据都是通过“0”和“1”来表示，逢二进一

◆位(bit)：二进制中的位，是计算机能处理的最小单位

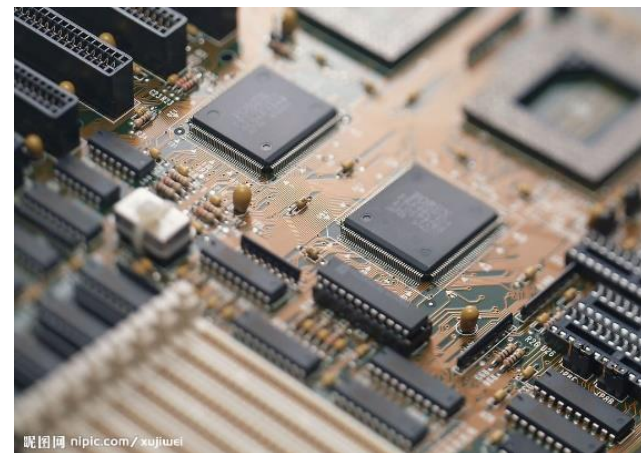


◆字节(Byte)：计算机处理的基本单位

✓计算机的内存是按字节进行分配的

✓一个字节由八位二进制数组成

✓C/C++语言中数据类型都是以字节为基本单元



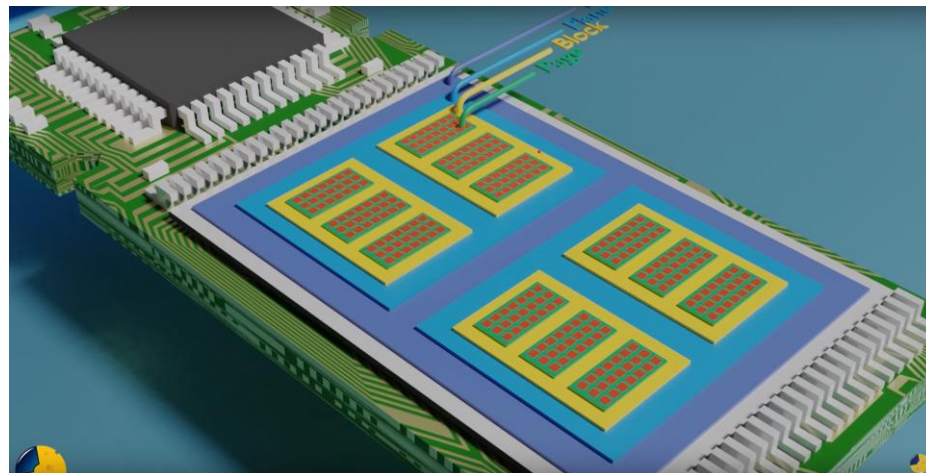


二进制的存储

地址 **9619**

内存

1000	0	0	1	0	0	1	1	1	← 1个字节
1001	1	0	0	0	0	1	1	1	
1002	0	0	0	0	0	0	1	1	
1003	1	0	0	0	0	1	1	1	← 4个字节
⋮	0	0	0	0	1	1	1	0	
	1	0	1	0	0	0	1	1	
	1	0	1	0	1	0	0	1	
	1	0	0	0	0	1	1	1	
	0	0	0	0	1	1	1	0	
	1	0	1	0	0	0	1	1	
	1	0	1	0	1	0	0	1	



字符型：1个字节

整型：4个字节

单精度浮点型：4个字节

双精度浮点型：8个字节



二进制：从十进制到二进制

◆例：如果已知十进制数 $(19)_{10}$ ，如何用二进制表示？

"十进制"整数转"二进制"数

2	$\overline{19}_{18}$	余数 1	低位
2	$\overline{9}_8$	1	
2	$\overline{4}_4$	0	
2	$\overline{2}_2$	0	
2	$\overline{1}_0$	1	高位

$(19)_{10} = (10011)_2$

除以2取余，逆序排列

记不住顺序？

"十进制"整数转"十进制"数

10	$\overline{123}_{120}$	余数 3	低位
10	$\overline{12}_{10}$	2	
10	$\overline{1}_0$	1	
	$\overline{0}$		高位

除以10取余，逆序排列



二进制：从二进制到十进制

◆例：已知二进制数 $(00010011)_2$ ，如何用十进制表示？

进制	十进制	二进制
实例	19	00010011

"二进制"数转"十进制"整数

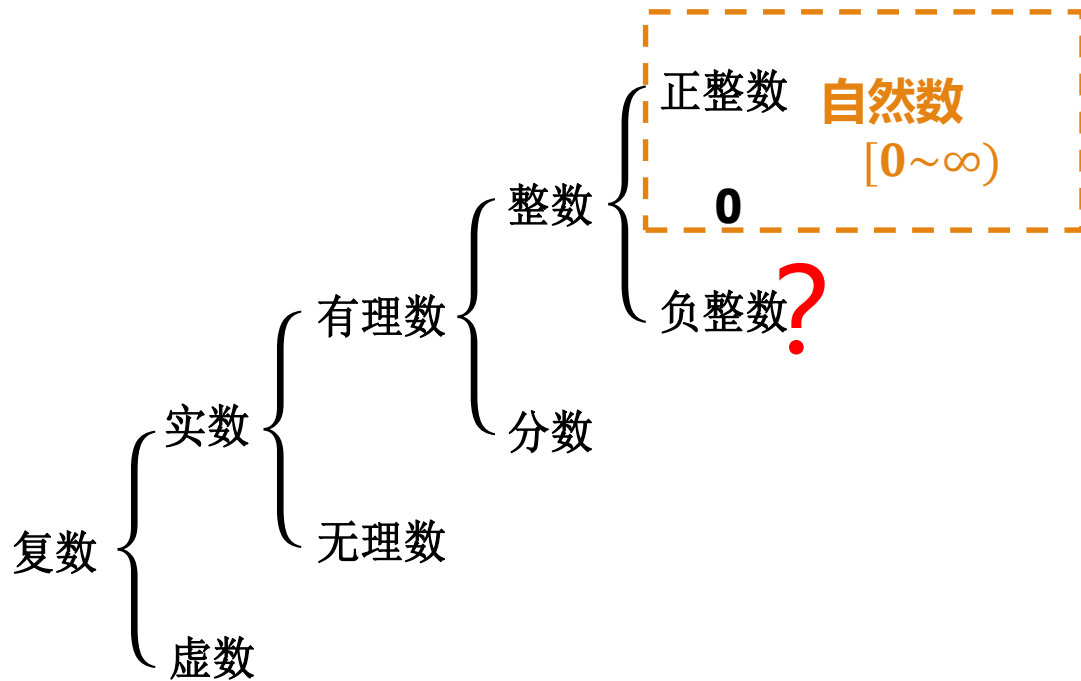
(1 0 0 1 1)₂

第4位

$$\begin{aligned} &= 1*2^4 + 1*2^1 + 1*2^0 \\ &= 16 + 2 + 1 \\ &= 19 \end{aligned}$$

$$B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i$$

$$123 = 1*10^2 + 2*10^1 + 3*10^0$$





二进制编码（含负数）：原码（以8位二进制为例！）

$7_{(10)}$ 转换成8位二进制数是 $(00000111)_2$ 那么-7呢？

+ 7	0	0	0	0	0	1	1	1
- 7	1	0	0	0	0	1	1	1

在原码中0有两种表示方式 +0 和 -0，第一位是符号位，在计算的时候根据符号位，选择对值区域加减，对于计算机很难，需要设计包含了计算数值和识别符号位两种电路，但是这样的硬件设计成本太高。

原码

- ◆ 最高位作为符号位（以0代表正，1代表负）
- ◆ 其余各位代表数值本身的绝对值
- ◆ 表示范围：
 $-127 \sim 127 \iff (-2^{8-1} + 1 \sim 2^{8-1} - 1)$

➤ 0 的表示不唯一

+0	0	0	0	0	0	0	0
-0	1	0	0	0	0	0	0

➤ 加减运算需要识别符号位，不适合计算机的运算



二进制编码：反码

◆反码：

- ✓正数的反码与原码相同
- ✓若为负数，则对其绝对值的原码取反

+7 原码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

-7 反码：对7的原码取反

+7 反码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

1	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

+

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = -0

反码：+0

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

原码：-0

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

反码：-0

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

- ◆同样，0 的表示不唯一，不适合计算机的运算
- ◆表示范围：-127~127 (+0, -0占用两种表示)



二进制编码：补码

◆补码

- ✓正数：原码、反码、补码相同
- ✓负数：对其绝对值的**原码取反**，再加 **1**（若有进位，则进位被丢弃）（反码+1）

+7 原码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 反码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 补码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

-7 补码: 7的原码 → 取反 → +1

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---



二进制编码：补码

◆补码

- ✓正数：原码、反码、补码相同
- ✓负数：对其绝对值的原码取反，再加 1（若有进位，则进位被丢弃）（反码+1）

+7 原码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 反码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 补码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

-7 补码: 7的原码 → 取反 → +1

1	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---



二进制编码：补码

◆补码

- ✓正数：原码、反码、补码相同
- ✓负数：对其绝对值的原码取反，再加 1（若有进位，则进位被丢弃）（反码+1）

+7 原码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 反码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

+7 补码

0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---

-7 补码: 7的原码 → 取反 → +1

1	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---



再说补码：补数

为了表示负数，我们在有限的计数系统中引入一个概念“补数”（即补码），先看时钟：

顺时针转9格和逆时针转3格是等价的。所以-3和9是关于12的补数。

以4位二进制数为例，共可以表示16个数

正数补数即为本身，

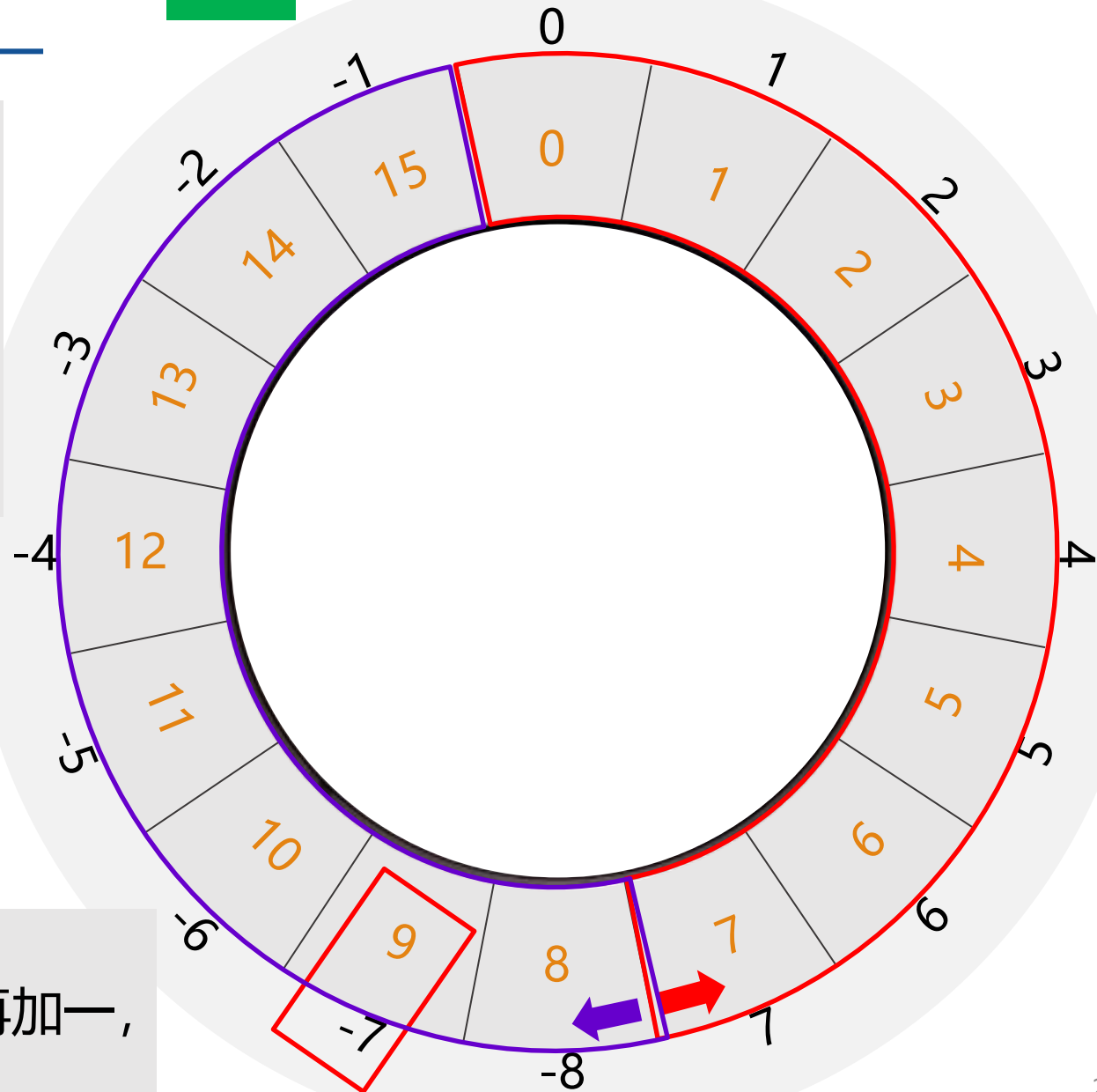
负数A的补数 = 模 - A的绝对值

如：-7的补数 = $16 - 7 = 9$

-x是一个负数，其补数是 $16 - x = 15 - x + 1$
15-x则相当于在4位二进制下对各位取反，再加一，即“取反加一”

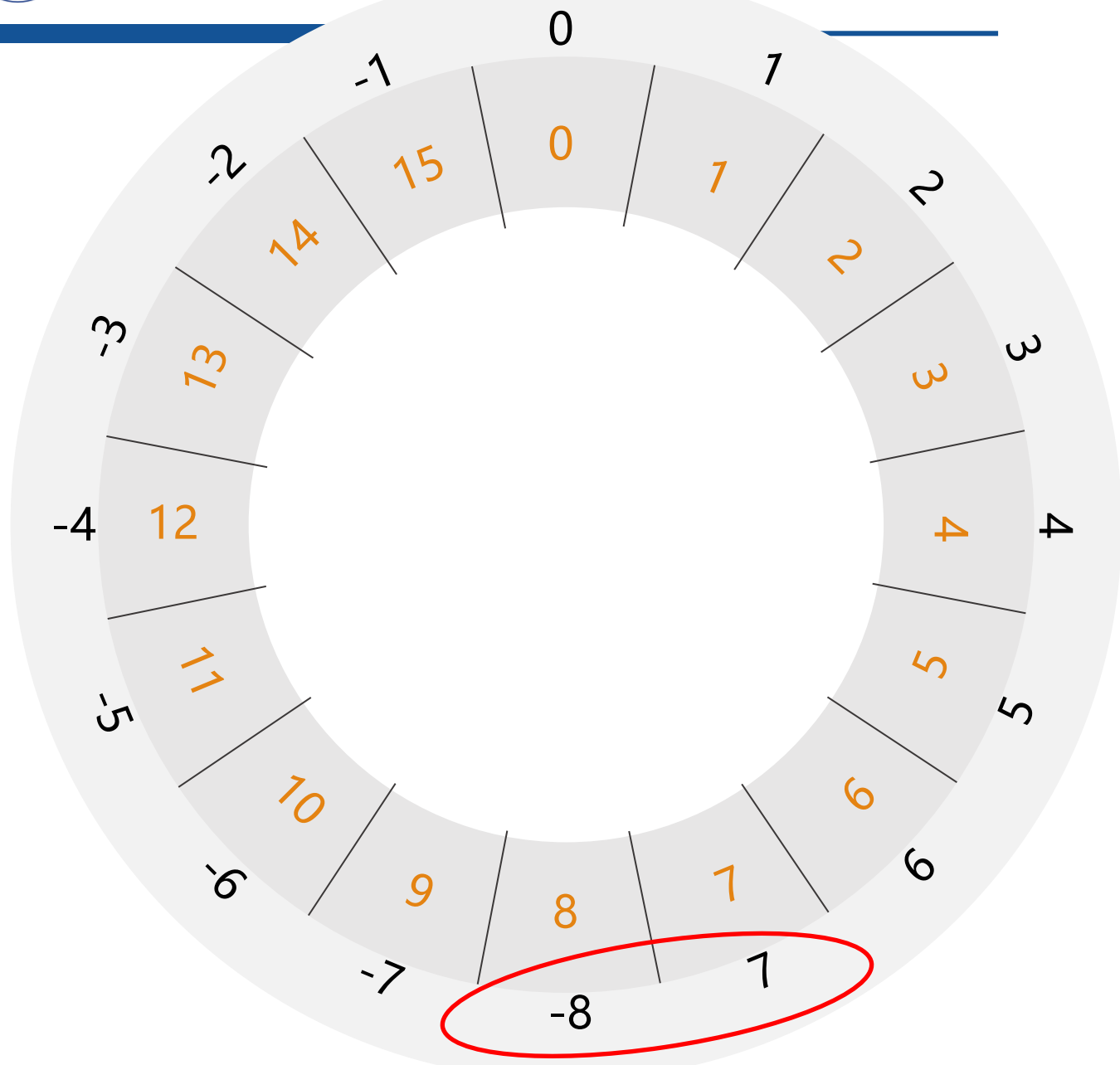
12

$X - 3 \leftrightarrow X + 9$





再说补码：补数



还可看出：

- ✓ 有符号数（补码）表示的正数和负数的范围是不对称的

4位有符号数：

$$-8 \sim 7 \quad (-2^3 \sim 2^3 - 1)$$

8位有符号数：

$$-128 \sim 127 \quad (-2^7 \sim 2^7 - 1)$$

- **无符号数和有符号数的转换**

1. w位有符号数转换成无符号数
(int -> unsigned int)

$$\text{有符号数 } a \begin{cases} \geq 0 & a \\ < 0 & a + 2^w \end{cases}$$

2. w位无符号数转换成有符号数
(unsigned int -> int)

$$\text{无符号数 } a \begin{cases} < 2^{w-1} & a \\ \geq 2^{w-1} & a - 2^w \end{cases}$$



两个特殊的数字编码

以一个字节大小的整数补码表示为例

原码: -0	1	0	0	0	0	0	0
反码:	1	1	1	1	1	1	1
补码:	1	0	0	0	0	0	0

+1

-127~127 :正数就是原码, 负数就是绝对值的原码取反再加1

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

?

~~128~~ 符号位和其他正数不一致

-128 补数: $256-128=128$

二进制数学表示

数值	补码
-128	10000000
-127	10000001
...	... (往上不断减1)
-2	11111110
-1	11111111
0	00000000
1	00000001
2	00000010
...	... (往下不断加1)
126	01111110
127	01111111

✓ 0的表示方式唯一

✓ 表示范围: -128~127



基于补码的运算

◆用补码进行运算，减法可以用加法来实现

✓如： $7-6=1$

+7 补码	0	0	0	0	0	1	1	1
					+			
-6 补码	1	1	1	1	1	0	1	0
1	0	0	0	0	0	0	0	1

人们想出一种方法使得符号位也参与运算。我们知道, 根据运算法则减去一个正数等于加上一个负数, 即:

$1-1 = 1 + (-1) = 0$, 所以机器可以只有加法而没有减法, **这样计算机运算的设计就更简单了。**

对于CPU来说，这是补码最重要的贡献：只要做加法就可以了！



3.3 更多的进制：八进制和十六进制

八进制

0 1 2 3 4 5 6 7

十六进制

0 1 2 3 4 5 6 7 8 9 A B C D E F

10 11 12 13 14 15

例: 15

二进制

0 0 0 0 1 1 1 1
 $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 4 + 2 + 1 = 15$

八进制

0 17
 $1 * 8^1 + 7 * 8^0 = 15$

十六进制

0x F
15

进制	十进制Dec	二进制Bin	八进制Oct	十六进制Hex
基本数字	0 ~ 9	0, 1	0 ~ 7	0 ~9, A~F (or a~f)
基数	10	2	8	16
规则	逢10进1	逢2进1	逢8进1	逢16进1
实例	19	00010011	023	0x13



八进制和十六进制

◆引入八进制或十六进制：二进制位太长，不易使用

八进制

0 1 7

二进制

0 0 0 0 1 1 1 1

十六进制

0 F

每个八进制数字的一位对应

3位二进制位 ($2^3 = 8$)

每个十六进制数字的一位对应

4位二进制位 ($2^4 = 16$)

"二进制"转"八进制"

$$\begin{aligned}(10011)_2 &= (0 \quad \quad \quad)_2 \\ &= (1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0)_8 \\ &= (2 \quad \quad 3)_8\end{aligned}$$

3位构成一组，高位不够补0

023

"二进制"转"十六进制"

$$\begin{aligned}(101111)_2 &= (00 \quad \quad \quad)_2 \\ &= (1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0)_{16} \\ &= (2 \quad \quad F)_{16}\end{aligned}$$

4位构成一组，高位不够补0

0x2F

注意不能写成15



更多的进制

◆十进制与二进制、八进制、十六进制

◆七进制

◆十二进制

◆二十四进制

◆四进制

◆三进制

◆...



3.4 基于二进制的位运算

◆前面的所有表达式运算规则都是基于十进制位运算的

◆**位运算**：基于整数在计算机中存储的**二进制位**定义运算规则

运算符	含义
&	按位与
	按位或
^	按位异或
~	取反
<<	左移
>>	右移

◆ 运算数只能是**整型**（或**字符型**）的数据，不能为实型数据

◆ 位运算符除 ~（取反）外均为二元运算符，~（取反）是一元运算符

位运算效率高、与计算机内部实现密切相关，是高手的秘密武器！

比如，在加密中应用广泛；很多黑客其实就是在经常玩位运算



按位与：&

◆运算规则：两个二进制位同时为1时为1，否则为0

A	B	A&B
1	1	1
0	1	0
1	0	0
0	0	0

例：3&5= 1

3	0	0	0	0	0	1	1
& 5	0	0	0	0	0	1	0
	0	0	0	0	0	0	1

1 保留原来的数值

0 不管原来数值是多少，都置0

应用：可用于实现 “清零” 操作

运算规则可类比串联电路



按位与：&

把数 x 的特定位置为0，其他位保持不变： $x = x \& ?$

	?	?	?	?	?	?	?	x	目标数
&	0	0	1	0	1	0	1	43	操作数
	0	0	*	0	*	0	*	x	

1不变 0清零

上例中保留x的第1, 2, 4, 6位，其他位置为零。更通用的实现方式：
 $x \& (1 \mid 1 \ll 1 \mid 1 \ll 3 \mid 1 \ll 5)$ 【稍后学习左移 \ll 】



按位与：&

◆简单应用：判断数字n的奇偶性

```
if ((a & 1) == 1) // if(a&1)
    printf("%d为奇数.\n", a);
else
    printf("%d为偶数.\n", a);
```



按位或：|

◆运算规则：两个二进制位同时为0时为0，否则为1

A	B	A&B
1	1	1
0	1	1
1	0	1
0	0	0

例：3 | 5 = ?

3	0	0	0	0	0	1	1
5	0	0	0	0	1	0	1
	0	0	0	0	1	1	1

0 保留原来的数值

1 不管原来数值是多少，都置1

运算规则可类比并联电路

应用：可用于实现 “置一” 操作



按位或：|

把 x 的特定位置为1: $x = x | ?$

	?	?	?	?	?	?	?	x	目标数
	0	0	1	0	1	0	1	43	操作数
<hr/>									
	*	*	1	*	1	*	1	x	

0不变 1置一



按位异或： \wedge

◆运算规则：两个二进制位相同时为0，不相同为1

A	B	$A \wedge B$
1	1	0
0	1	1
1	0	1
0	0	0

例： $3 \wedge 5 = ?$

3	0	0	0	0	0	1	1
\wedge 5	0	0	0	0	1	0	1
	0	0	0	0	1	1	0

0 保留原来的数值

1 不管原来数值是多少，都翻转

运算规则：同相斥，异相吸

应用：可用于实现“翻转”操作



按位异或: ^

把 x 的特定位置翻转: $x \wedge *$

	1	0	0	1	0	1	0	1	x	寄存器值
^	0	0	1	0	1	0	1	1	43	操作数
<hr/>										
	1	0	1	1	1	1	1	0		
			翻转		翻转		翻转	翻转		

0不变 1翻转



按位异或: ^

利用异或交换两个变量的值

中间变量 temp

```
temp = a;
```

```
a = b;
```

```
b = temp;
```

```
a = a^b;
```

```
b = b^a;
```

```
a = a^b
```



按位取反：~

◆运算规则：一元运算符，0变1，1变0

例1：将一个数 a 的最低位置为 0，其他位不变

a	?	?	?	?	?	?	?
& ~1	1	1	1	1	1	1	0
	*	*	*	*	*	*	0

$$a = a \& \sim 1$$

例2：对n取相反数 $\sim n + 1$

例3： `while(scanf(...) != EOF){...}`
`while(~scanf(...)){...}`

例：~3=?

3	0	0	0	0	0	0	1	1
~3	1	1	1	1	1	1	0	0



按位左移: \ll

◆运算规则：将数的二进制位全部左移若干位，左边溢出的位舍弃，右边空位补 0

例：若 $a = 15$ ，将 a 的二进制数左移 2 位， $a = a \ll 2$

$a = 15$



$a = a \ll 2$?





按位左移：<<

◆运算规则：将数的二进制位全部左移若干位，左边溢出的位舍弃，右边空位补 0

例：若 $a = 15$ ，将 a 的二进制数左移 2 位， $a = a \ll 2$

$a = 15$



$a = a \ll 2$



||

$$a = 15 \times 2^2 = 60$$

- 高位左移后溢出，舍弃
- 左移一位相当于该数乘以2（超出数据类型表示范围后将造成错误结果）
- 左移比乘法运算快得多



按位右移: $>>$

◆运算规则：将数的二进制位整体右移若干位，右边移出的低位被舍弃，左边空出的高位，可补0（逻辑位移），可补1

（算术位移）

✓无符号数，采用逻辑位移

✓有符号数，根据编译器的具体实现采用逻辑位移或算术位移

例：若 $a = 15$ ，将 a 的二进制数右移 2 位， $a = a >> 2$

$a = 15$



$a = a >> 2$?





按位右移: >>

◆运算规则：将数的二进制位整体右移若干位，右边移出的低位被舍弃，左边空出的高位，可补0（逻辑位移），可补1

（算术位移）

✓无符号数，采用逻辑位移

✓=有符号数，根据编译器的具体实现采用逻辑位移或算术位移

例：若 $a = 15$ ，将 a 的二进制数右移 2 位， $a = a \gg 2$

$a = 15$



$a = a \gg 2$



||

$$a = 15 / 2^2 = 3$$

• 右移一位相当于除以2



位运算的复合赋值运算符

◆二元位运算符也支持复合赋值形式

✓ $a \&= 5$ 等价于 $a = a \& 5$;

赋值运算符	示例表达式	等价含义
$\&=$	$c \&= 1$	$c = c \& 1$
$ =$	$d = 2$	$d = d 2$
$\wedge=$	$e \wedge= 3$	$e = e \wedge 3$
$\ll=$	$f \ll= 4$	$f = f \ll 4$
$\gg=$	$g \gg= 5$	$g = g \gg 5$



C03-01：补码输出

- ◆ C03-01：给定一个整数 n ，输出该整数的二进制补码编码
- ◆ 问题分析：整数在计算机中存储的即为二进制补码形式，直接输出其二进制位即可
 - ✓ 如何获得数的二进制位
 - 最低位的二进制位： $n \& 1$
 - 其他二进制位可以通过移位的方式移到个位
 - ✓ 如何存储二进制位
 - C语言并没有提供二进制数据的表示法，可以用只取0、1两个取值的整数数组来一位位存储二进制位
 - `int bits[32];`



C03-01: 补码输出

```
#include <stdio.h>
#define LEN 32
int main(){
    int n, i;
    int bits[LEN]; //以十进制0、1的形式存储二进制位
    scanf("%d", &n);
    for (i = 0; i < LEN; i++){
        bits[i] = n & 1; //获得n的二进制个位
        n >>= 1;          //右移，将二进制个位移出，二进制十位变个位...
    }
    for (i = LEN - 1; i >= 0; i--){
        //逆序输出二进制位
        printf("%d", bits[i]);
    }
    return 0;
}
```

理解：如何获得整数各个位数的值！
(十进制、二进制...)

思考：如何输出原码？反码？



C03-02: 特定位置赋值

◆C03-02: 用C语言给一个无符号整数 a 的bit7 ~ bit17赋值937, 同时给bit21 ~ bit25赋值17 (位数从0开始)

◆题目分析: 如何设置指定的位?

- ✓清0 (按位与: &) : 先把需要修改的位置清0
- ✓设置值 (按位或: |) : 把这些位置设置为指定的值
- ✓要点

- 937和17的二进制值
- 特定位置清0、设置特定值





C03-02: 特定位赋值

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

1) 7-17位设置为0: 按位与&

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	

16进制值: 0xF F F C 0 0 7 F

即: 0xFFFC007F

2) 7-17位设置为937: 按位或 | 937的二进制: 0011 1010 1001

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	

16进制值: 0x0 0 0 1 D 4 8 0

即: 0x0001D480

同理可设置21-25位为17



C03-02: 特定位赋值

```
#include <stdio.h>
int main()
{
    unsigned int n;
    scanf("%u", &n); // 读入无符号整数
    n &= 0xFFFC007F; // 通过与操作将7-17位设为0
    n |= 0x0001D480; // 通过或操作将7-17位设为937

    n &= 0xFC1FFFFFFF; // 通过与操作将21-25位设为0
    n |= 0x02200000; // 通过与操作将21-25位设为17

    printf("%u", n); // 输出无符号整数
}
```



C03-02: 换个思路实现

- ◆直接按照位数拼凑二进制位原理简单，但不直观、容易出错
- ◆可以借助移位操作来实现
 - ✓以数字1为起点，通过各种移位、取反、与/或、加/减等操作得到需要的数字
- ◆如何做？
 - ✓将7-17位置设为937的操作，其他位为0
 - $937 \ll 7$;
 - ✓将7-17位设为1，其他位设为0的操作
 - 先得到从低位开始的11个1，然后移到7-17位



C03-02: 特定定位赋值 (另一种实现)

```
#include <stdio.h>
int main()
{
    unsigned int n;
    scanf("%u", &n);

    n &= ~( ((1<<11) - 1) << 7 );
    n |= 937<<7;
    n &= ~( ((1<<5) - 1) <<21);
    n |= 17<<21;

    printf("%u", n);
}
```

1. $0x7ff$ 为 $0..0\ 0111\ 1111\ 1111$, 即, 初始化低11 (十一) 位为1, $((1<<11) - 1)$
2. $((1<<11) - 1) << 7$, 得到 $0011\ 1111\ 1111\ 1000\ 0000$
把第1步的十一个1左移7位 (这十一个1变成bit7 ~ bit17)
3. $\sim((1<<11) - 1) << 7$
bit7 ~ bit17的十一个1变成0, 其他位的0变成1, 即变为 $1100\ 0000\ 0000\ 0111\ 1111$
4. $a \&= \sim(0x7ff << 7)$, 保留a的其他位, 但把a的bit7 ~ bit17都置为0
5. $a \mid= (937 << 7)$, 把a的bit7 ~ bit17置为937
6. bit21 ~ bit25赋值为17, 原理同上



更多的位运算应用

求两个数的平均值:

$$(x + y) >> 1$$

计算2的n次方

$$1 << n$$

从低位到高位, 将n的第m位置1

$$n | (1 << (m-1))$$

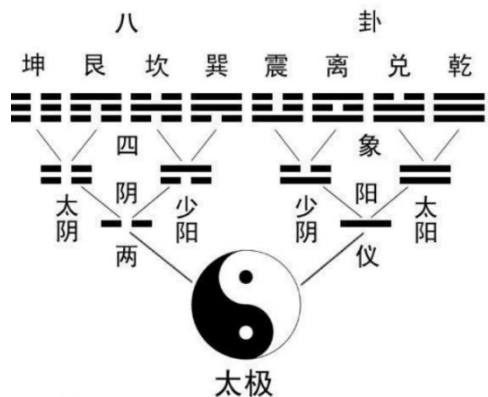
从低位到高位, 将n的第m位置0

$$n \& \sim(1 << (m-1))$$

计算最大、最小值

$$\text{最大值: } x \wedge ((x \wedge y) \& -(x < y))$$

$$\text{最小值: } x \wedge ((x \wedge y) \& -(x > y))$$



一生万物

从 1 出发, 进行位运算, 搞定所有复杂应用!



3.5 浮点数的编码与精度问题

```
#include <stdio.h>
int main()
{
    int a = 625, b = 3;
    printf("%d, %d\n", (a==625), (b==3));
    float x = 0.625, y = 0.3;
    printf("%d, %d", (x==0.625), (y==0.3));
    return 0;
}
```

1, 1
1, 0

0.3不等于0.3!!!

说明：这是dev c下编译运行的结果，其他环境下可能相等！



小数的二进制

进制	十进制	数学意义的二进制表示
实例	19.625	00010011.101

A. "十进制"**整数**转"二进制"数

除以2取余
逆序排列

2	19	18	1	低位
2	9	8	1	
2	4	4	0	
2	2	2	0	
2	1	0	1	高位
	0			

$$(19)_{10} = (10011)_2$$

B. "十进制"**小数**转"二进制"小数

乘以2取整
顺序排列

	整数部分	高位
$0.625 \times 2 = 1.25 \dots$	1	
$0.25 \times 2 = 0.5 \dots$	0	
$0.5 \times 2 = 1 \dots$	1	
$(0.625)_{10} = (0.101)_2$		低位



小数的二进制

进制	十进制	数学意义的二进制表示
实例	19.3	00010011.010011001.....

```
#include <stdio.h>
int main()
{
    int a = 625, b = 3;
    printf("%d, %d\n", (a==625), (b==3));
    float x = 0.625, y = 0.3;
    printf("%d, %d", (x==0.625), (y==0.3));
    return 0;
}
```

B. "十进制"小数转"二进制"小数

乘以2
取整
顺序排列

$$0.3 \times 2 = 0.6 \dots\dots 0$$

$$0.6 \times 2 = 1.2 \dots\dots 1$$

$$0.2 \times 2 = 0.4 \dots\dots 0$$

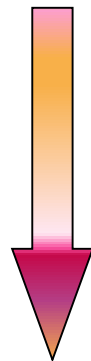
$$0.4 \times 2 = 0.8 \dots\dots 0$$

$$0.8 \times 2 = 1.6 \dots\dots 1$$

$$0.6 \times 2 = 1.2 \dots\dots 1$$

... 循环了!

$$(0.3)_{10} = (0.0\overline{1001} \dots)_2$$



**注意：浮点数表达不精确，
用 == 判断相等时一定要小心！**



小数的二进制

```
float b = 0.3;
if (((int)(b * 1000)) == 300)
{
    printf("b == 0.3\n");
    printf("点火\n");
}
else
{
    printf("b != 0.3\n");
    printf("不点火\n");
}
```



浮点数在关系运算中的思考：
数学问题？
计算机问题？
哲学问题？
工程问题？
安全问题？

b != 0.3
不点火

一行代码引发的惨剧
应该点火，却不点火





小数在计算机中的存储

◆数学表达不等于计算机表达！

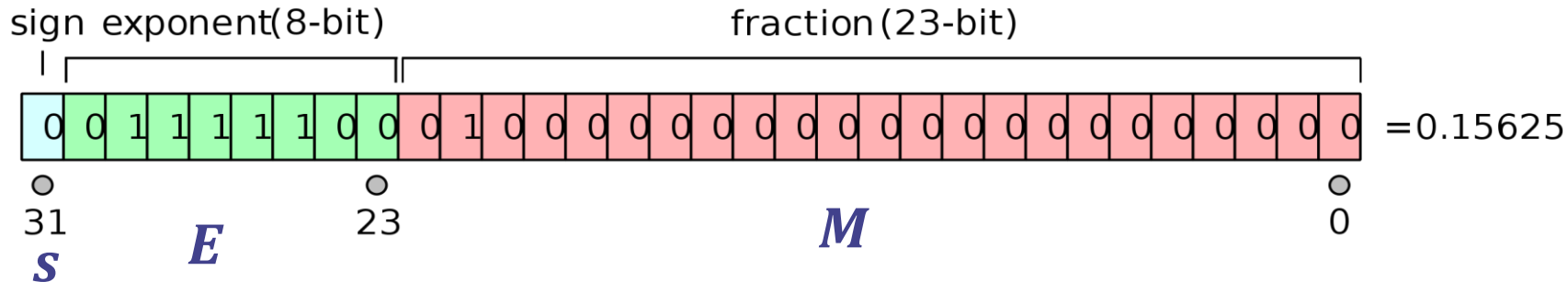
✓不同于整数，小数在计算机中并不是按照编码值直接存储的

◆使用标准数据格式 IEEE-754 存储和表示

✓数值以规范化的二进制数指数形式存放在内存单元中

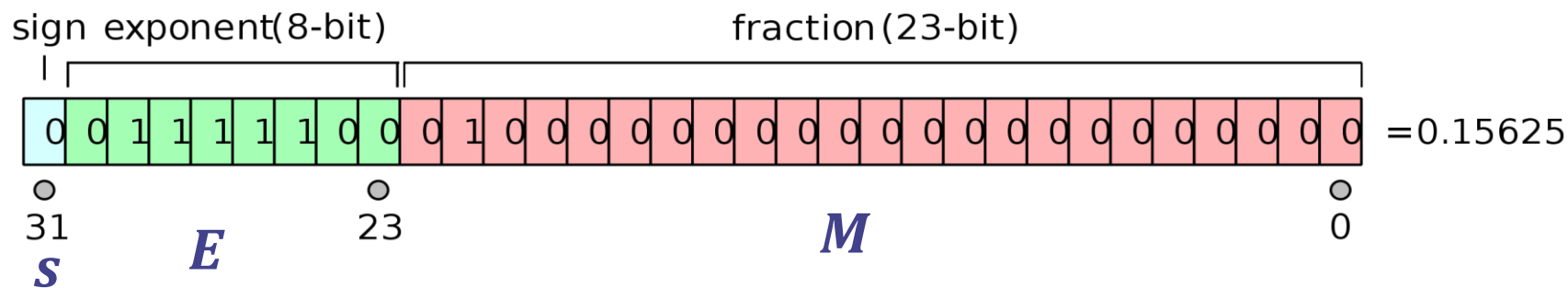
✓存储时分成：符号 (sign)、指数部分 (exponent, E) 和小数部分 (fraction, M) 分别存放

◆例：32位单精度浮点数 (float)





浮点数的存储范围



$$x = (-1)^s \times (1.M) \times 2^{E-127} \quad (\text{float})$$

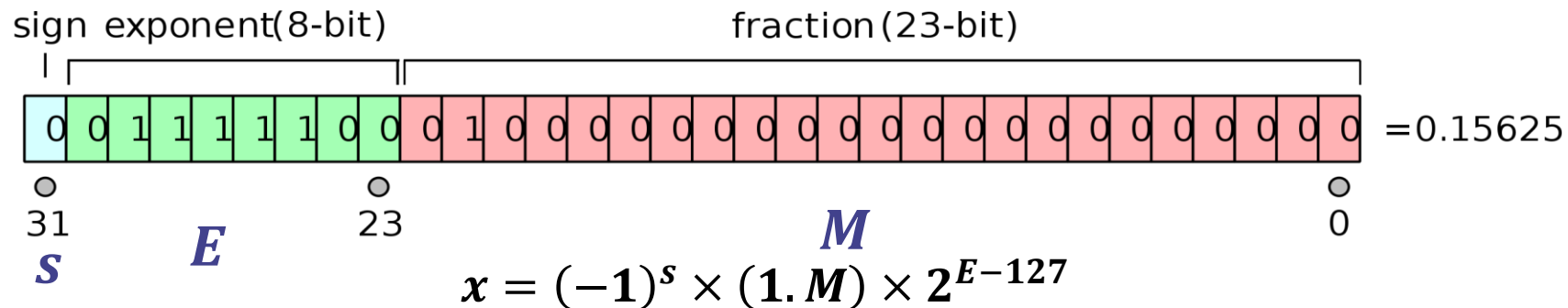
$$x = (-1)^s \times (1.M) \times 2^{E-1023} \quad (\text{double})$$

浮点数类型	符号(+/-)	指数	小数部分
float	1	8	23
double	1	11	52
long double	1	15	112

指数部分 (E) 决定范围, 小数部分 (M) 决定精度!



IEEE-754 标准数据格式（单精度浮点型）



(1) 首先把实数转为二进制的指数形式

$$-3.75 = -\left(2 + 1 + \frac{1}{2} + \frac{1}{4}\right) = -\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right) \times 2 = -(1.111)_2 \times 2^1$$

(2) 整理符号位并进行规范化:

$$-1.111 \times 2^1 = (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^1$$

(3) 进行阶码的移码处理

$$(-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^1 = (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{128-127}$$

(4) $s = 1, M = 1110\ 0000\ 0000\ 0000\ 0000\ 000, E = (128)_{10} = (10000000)_2$



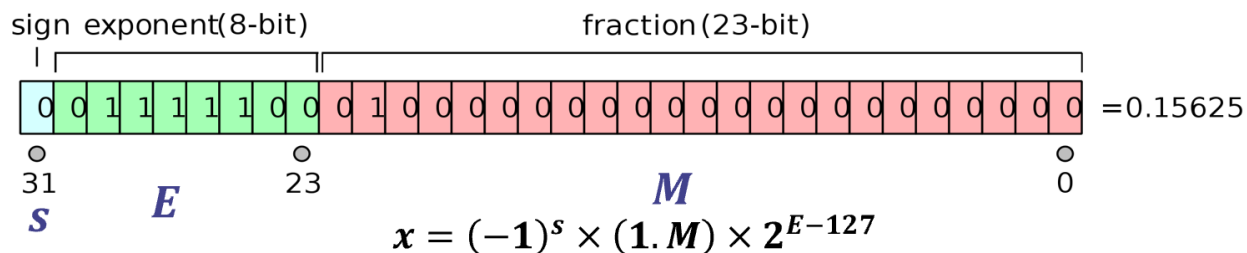
- $$+1.\boxed{000} \times 2^0$$

- ◆ 而比1大的最小双精度浮点数是:

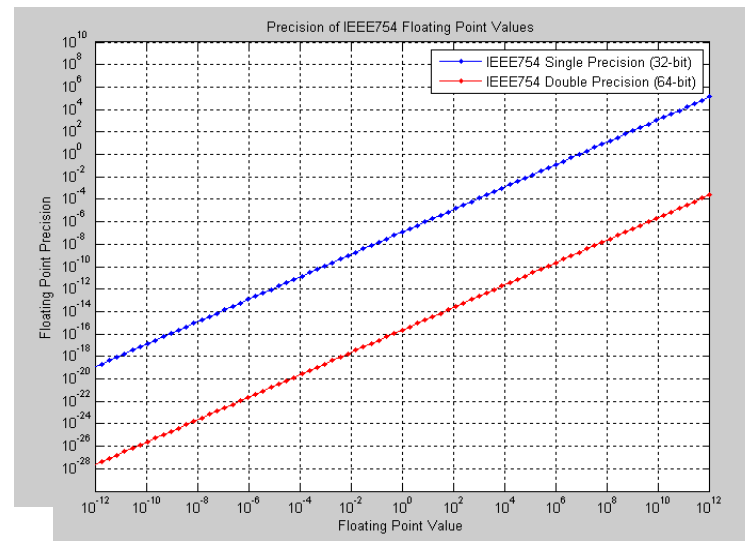
[illegible]

- ◆ 此之差为机器 ϵ : $2^{-52} \approx 2.220446049250313e-16$

- ## ◆ 绝对精度



float和double类型数据的绝对精度





C03-03: 一元二次方程求根

- ◆ 输入 a 、 b 、 c 三个实数，求一元二次方程 $ax^2+bx+c=0$ 的根
- ✓ 1. $a=0$, 方程不是二次方程
 - ✓ 2. $b^2-4ac=0$, 有两个相等的实根
 - ✓ 3. $b^2-4ac>0$, 有两个不相等的实根
 - ✓ 4. $b^2-4ac<0$, 有两个共轭复根



C03-03: 一元二次方程求根

```
#include <stdio.h>
#include <math.h>
int main(){
    double a, b, c, delta, r1, r2;
    scanf("%lf%lf%lf", &a, &b, &c);
    if (a==0) { //a为0, 不是一元二次方程
        printf("Not a quadratic\n");
    }else{
        delta = b * b - 4 * a * c; //根据delta判断根的情况
        if(delta == 0){ //两个相等的实根 ???
            printf("Two equal roots: %8.2f\n", -b/(2*a));
        }else if (delta > 0){ //两个不相等的实根
            r1 = (-b + sqrt(delta)) / (2 * a);
            r2 = (-b - sqrt(delta)) / (2 * a);
            printf("Two roots: %8.2f,%8.2f", r1, r2);
        }else { //两个虚根
            printf("Two imaginary roots");
        }
    }
    return 0;
}
```

由于运算过程中的精度问题, delta
实际值为0, 但存储的不等于0
(可能一个很小的数据)



C03-03: 一元二次方程求根

```
#include <stdio.h>
#include <math.h>
int main(){
    double a, b, c, delta, r1, r2;
    double eps = 1e-9; //定义一个精度值, 小于这个数的即为0
    scanf("%lf%lf%lf", &a, &b, &c);
    if(a==0){ //a为0, 不是一元二次方程? 此处有精度问题吗?
        printf("Not a quadratic\n");
    }else{
        delta = b * b - 4 * a * c;
        if(fabs(delta) < eps){ //两个相等的实根
            printf("Two equal roots: %8.2f\n", -b/(2*a));
        }else if (delta > 0){ //两个不相等的实根
            r1 = (-b + sqrt(delta)) / (2 * a);
            r2 = (-b - sqrt(delta)) / (2 * a);
            printf("Two roots: %8.2f,%8.2f", r1, r2);
        }else { //两个虚根
            printf("Two imaginary roots");
        }
    }
    return 0;
}
```

注意: 1. eps的值可根据题目情况设置, 如1e-6、1e-9、1e-12...

2. fabs: 求double类型的绝对值



浮点数的要点小结

- ◆在C语言中，浮点数有范围，有精度限制。
- ◆浮点数使用标准数据格式（IEEE-754）：
 - ✓float的有效数字大约相当于十进制的7位，表示范围约 $-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$ (2^{128})？能表示的绝对值最小数约为 $1.175494351 \times 10^{-38}$
 - ✓double能表示的范围和精度更大。
- ◆浮点数的表示是近似值
 - ✓如显示的是1.0，计算机中实际可能是0.999999999...，也可能是1.00000001...
- ◆使用浮点数要特别注意范围和精度问题！



数的范围和精度

精度100%
但范围小
如: int, $2^{31}-1$

char, int, short,
long, long long,
unsigned ...

鱼和熊掌
不可兼得

精度可能受损
但范围大
如: float, 3.4×10^{38}

float, double, ...



C03-04: 浮点数的精度

```
#include <stdio.h>
const double eps = 1e-9;
int main()
{
    int x, y;
    double f = 0.0006;
    x = (int)(f * 10000);
    printf("%d\n", x);

    y = (int)((f + eps) * 10000);
    printf("%d\n", y);
    return 0;
}
```

输出

5
6

思考：为什么输出该结果？



注意：精度问题不仅仅是小数部分！

```
#include <stdio.h>
int main()
{
    double f = 1234567890123456789;
    printf("%.0f", f);
    return 0;
}
```

输出

1234567890123456800

思考：为什么整数部分也存在精度问题，IEEE-754？



3.6 一维数组的存储和应用

◆ 数组表示一组相同类型的数，由具有相同名称和相同类型的一组连续内存地址来表示

// 部分初始化，a的后6个元素自动初始化为0

```
int a[12] = {1, 3, 5, -2, -4, 6};
```

```
for (i=0; i<12; i++)  
    printf("%d ", a[i]);
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
1	3	5	-2	-4	6	0	0	0	0	0	0

内存 (Memory)

		60FEF8 a	60FEF9	60FEFA	60FEFB	60FEFC	60FEFD	60FEFE	60FEFF
60FF00	60FF01	...02	...03					60FF23	...24
...26	...27	60FF28							



数组的类型与大小

◆定义数组时，尽量指定数组的长度

✓C99支持不定长数组，但建议尽量使用定长数组

◆sizeof运算符可以获得数据类型或变量的实际字节数

```
#define LENGTH 100  
int a[LENGTH];  
double b[LENGTH];  
char c[LENGTH];
```

sizeof(para) 一元运算符，计算参数para所占的字节数，参数可以是变量、数组、类型等

4, 4

8

1

40, 4

← 输出

```
int i;  
double d;  
char c;  
float f[10];  
printf("%d, %d\n", sizeof(i), sizeof(int));  
printf("%d\n", sizeof(d));  
printf("%d\n", sizeof(c));  
printf("%d, %d\n", sizeof(f), sizeof(f[0]));
```



有关数组大小的问题

- ◆实际处理的问题可能很大，如淘宝数据几亿个用户(M个)，几千万件商品(N件)，数组是否应定义为a[M][N]?
- ◆数组大小多大合适?
 - ✓取决于计算机的能力、程序算法的设计、实际问题的需要
- ◆通常，全局数组可以比较大（但也不宜上百MB），局部数组比较小（通常几十KB）

```
#include <stdio.h>
```

```
//main函数外面，全局数组，可以很大
```

```
int voiceData[1<<20];
```

```
int main()
```

```
{
```

```
//main函数里面，局部数组，较小
```

```
double stuScore[2000];
```

```
...
```

```
}
```

提示：内存资源比较宝贵，根据问题要求定义合适的大小



C03-05: 字符统计

- ◆ 给出标准输入字符序列，统计输入中的每个小写字母出现的次数、所有大写字母出现的总次数、字符总数
- ◆ 问题分析
 - ✓ 统计大写字母的总次数
 - 如何判断大写字母？
 - ✓ 统计每个小写字母出现的次数
 - 如何保存每个小写字母出现次数？
 - 26个成员的数组，对应26个小写字母的次数



C03_05 · 字符统计

```
#include <stdio.h>
#include <ctype.h>
#define N 26
int main(){
    int i, c;
    int upper=0, total=0, lower[N]= {0};
    while((c=getchar()) != EOF ){//读入一个字符
        if(islower(c)) //函数：判断是小写字母
            lower[c - 'a']++;//如果c是'a',存入lower[0], 如此类推
        else if(isupper(c))//函数：判断是大写字母
            upper++;
        total++;
    }
    for ( i=0; i<N; i++ ){
        if(lower[i] != 0)//把下标转换为对应的小写字母
            printf("%c: %d\n", i + 'a', lower[i]);
    }
    printf("Upper: %d\nTotal: %d\n", upper, total);
    return 0;
}
```

这里用法很巧妙

数组元素的下标来表示字母

'a' - 'a' -> 0

'b' - 'a' -> 1, ... ,

数组元素（整形）用于计数

lower[0]计'a'出现次数，

lower[1]计'b'出现次数，

...

lower[c - 'a']++; 等价于
if(c == 'a') lower[0]++; ...

提示：学会使用ctype头文件中各类字符判断和转换函数！！
思考：如何将数组下标与字符对应！



数组的复制和比较

- ◆ 一维数组的元素 $a[i]$ 可以当作普通变量进行相应操作，但数组名 a 代表整个数组，不能参与普通变量的运算
 - ✓ 数组名实际上是地址，不允许对数组进行整体操作，不能整体输入和输出（除了字符串数组有特殊用法）

```
int a[12], b[12];
```

```
...
```

```
b = a;
```

```
if( b == a )
```

```
printf("%d", a);
```

```
...
```

复制数组

比较数组

整体输出



正确的做法

```
for(i=0; i<12; i++)
```

```
    b[i] = a[i];
```

```
for(i=0; i<12; i++)
```

```
    if(a[i] == b[i])
```

```
for(i=0; i<12; i++)
```

```
    printf("%d", a[i]);
```



数组的赋值与比较

◆可使用标准库函数实现数组的整体赋值（头文件：string.h）

函数原型：

```
void *memcpy(void *dest, void *src, size_t count);
```

用法：

```
memcpy(b, a, sizeof(a)); //把数组a的内容复制给数组b
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]

内存 (Memory)									
	60FEF8 a	60FEF9	60FEFA	60FEFB	60FEFC	60FEFD	60FEFE	60FEFF
60FF00	60FF0127	60FF28		



总结：数据处理

◆进制转换

- ✓二进制与十进制
- ✓二进制与八进制、十六进制

◆整数编码：原码、反码和补码

- ✓补码编码规则、数据范围

◆位运算

- ✓基本运算规则、含义和使用

◆浮点数编码：基本编码原理、精度问题

◆一维数组的存储和应用