

## 第三讲(Part3)

# 数据处理基础

## data processing



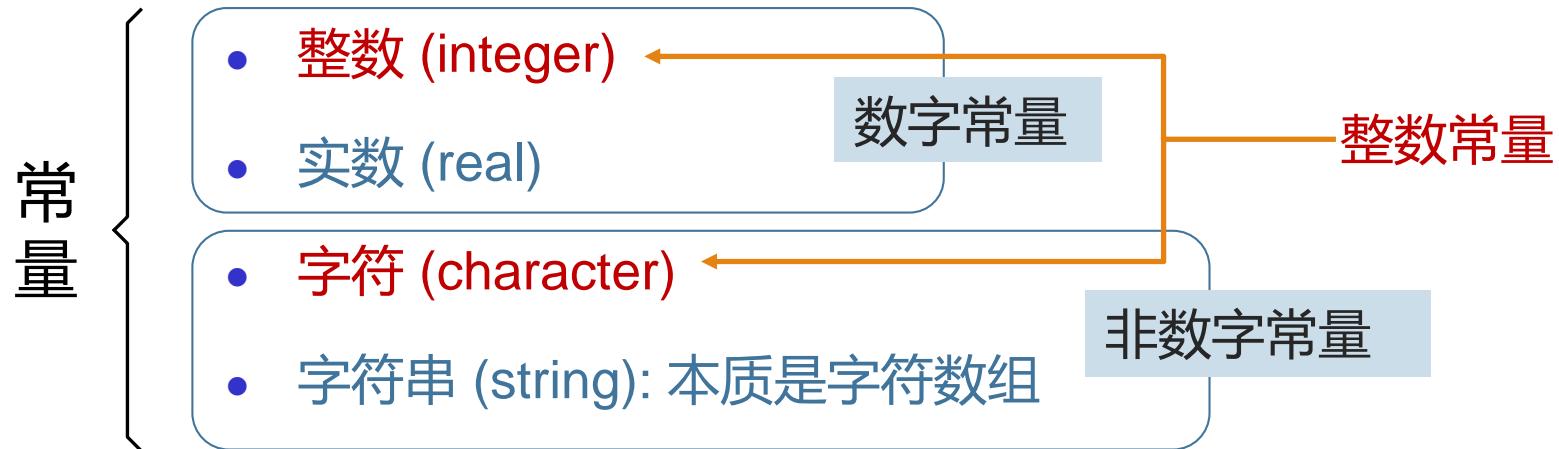
## 前情回顾

- 简单输入、输出、基本运算（算术运算、关系运算、逻辑运算）
- 对什么进行输入、输出、运算？

## 数据

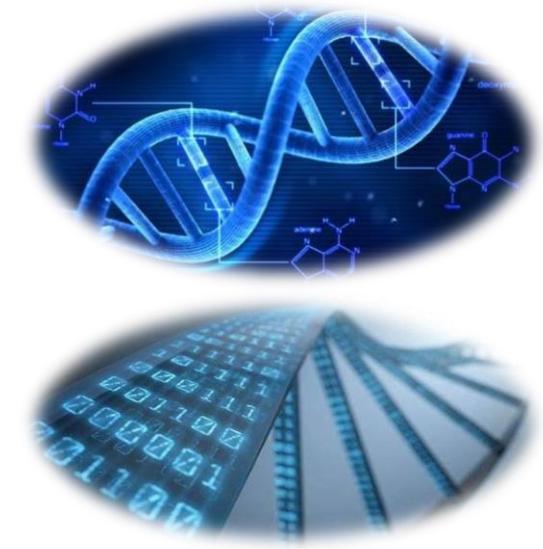
- 数据在计算机里存储、计算的本质是什么？

# 前情回顾



变量

| 数据类型          | 关键字                           |
|---------------|-------------------------------|
| 整型(integer)   | int, short, long, unsigned... |
| 实型(real)      | float, double                 |
| 字符(character) | char                          |
| 字符串(string)   | char型的数组或指针                   |



这些数在计算机中  
如何编码和存储?

# 第三讲 数据处理基础

## 数据 处 理 基 础

- 3.1 数值在计算机中的表示
- 3.2 进制转换
- 3.3 二进制与位运算符
- 3.4 浮点数及数据范围
- 3.5 变量与内存的关系
- 3.6 数组基础
- 3.7 标准输入输出重定向

## 学习要点

1. 数据（整数、浮点数）在计算机中的表示
2. 二进制、位、字节
3. 二进制的原码、反码和补码
4. 二进制与位运算
5. 十进制、二进制、八进制、十六进制之间的转换
6. 变量与内存的关系
7. 各种数据类型的数据范围
8. 数据的范围与精度的相对关系
9. 数组简介
10. 输入输出IO、`freopen()`、IO重定向

## 3.1 | 数值在计算机中的表示

# 两段有点“奇怪”的代码

```
// c3-0-1.c
#include <stdio.h>
int main()
{
    int a, b;
    signed char sum = 0; // 有符号字符
    scanf("%d%d", &a, &b);
    sum = a + b;
    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```

100 100  
100 + 100 = -56

怪象1：100+100 不等于 200？

```
// c3-0-2.c
#include <stdio.h>
int main()
{
    long long 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

说明：这是笔者在codeblocks下编译运行的结果。读者可以亲自测试一下。

怪象2：0.3 等于 0.3 不成立？

# 两段有点“奇怪”的代码

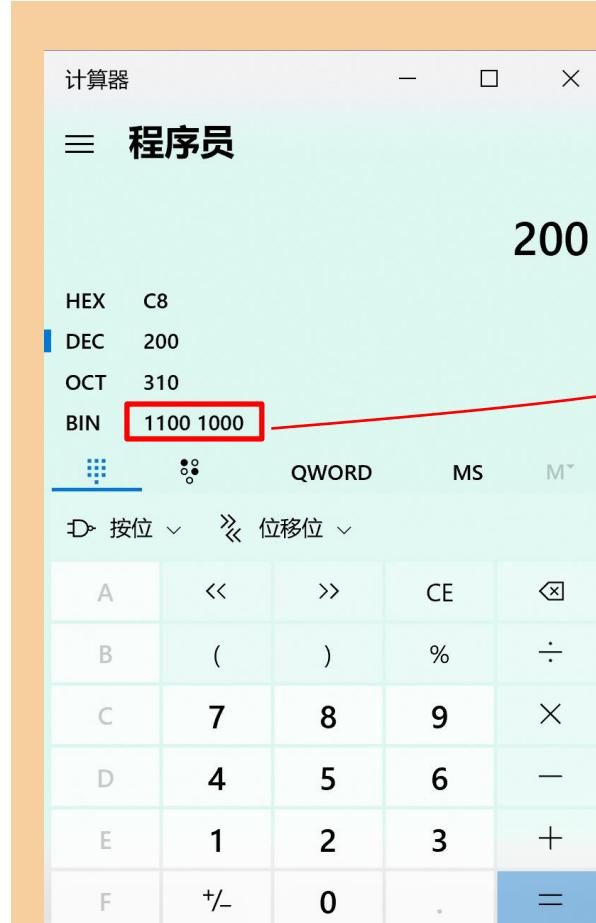
```
// c3-0-1.c
#include <stdio.h>
int main()
{
    int a, b;
    signed char sum = 0; // 有符号字符

    scanf("%d%d", &a, &b);
    sum = a + b;

    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```

100 100  
100 + 100 = -56

怪象1：100+100 不等于 200？



整数200的二进制编码：

00...00 1100 1000

3个字节

1个字节

变量 sum 是 signed char 类型，占1个字节，取值 11001000（整数的高位3个字节被截取掉），有符号数的最高位为符号位（1表示负数），11001000是-56的补码表示（计算机中的整数表示方式）。

何为**补码**表示？请认真听讲！

# 两段有点“奇怪”的代码

```
// c3-0-2.c
#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;
}
```

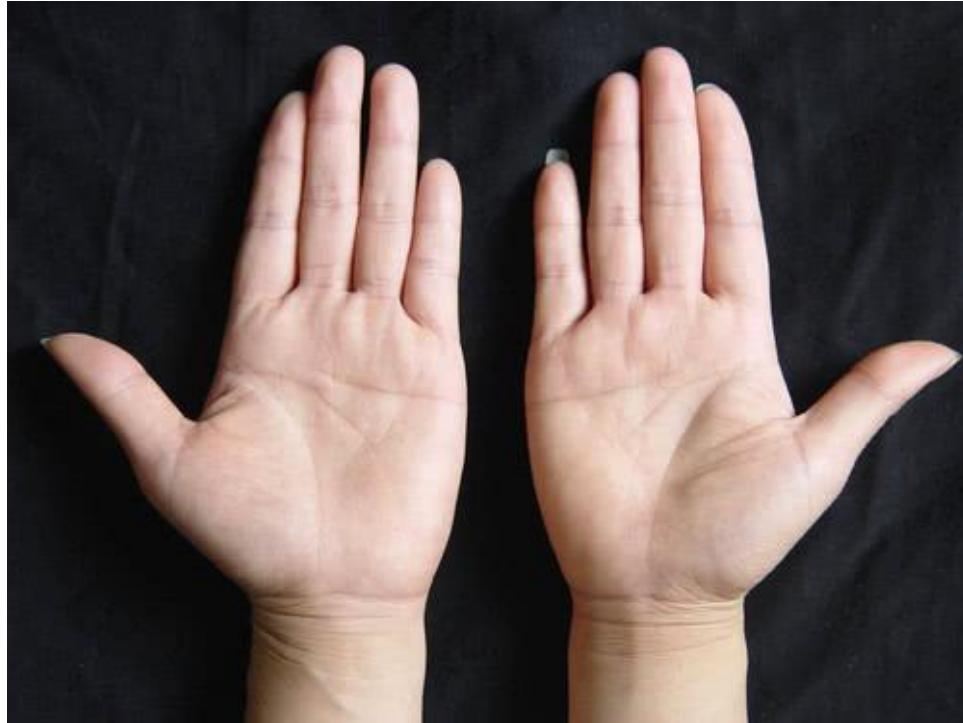
1, 1  
1, 0

怪象2: 0.3 等于 0.3 不成立?

### 3.1 数值在计算机中的表示：二进制

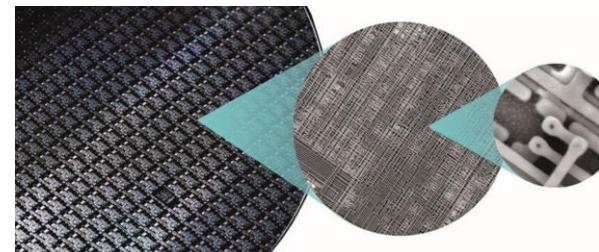
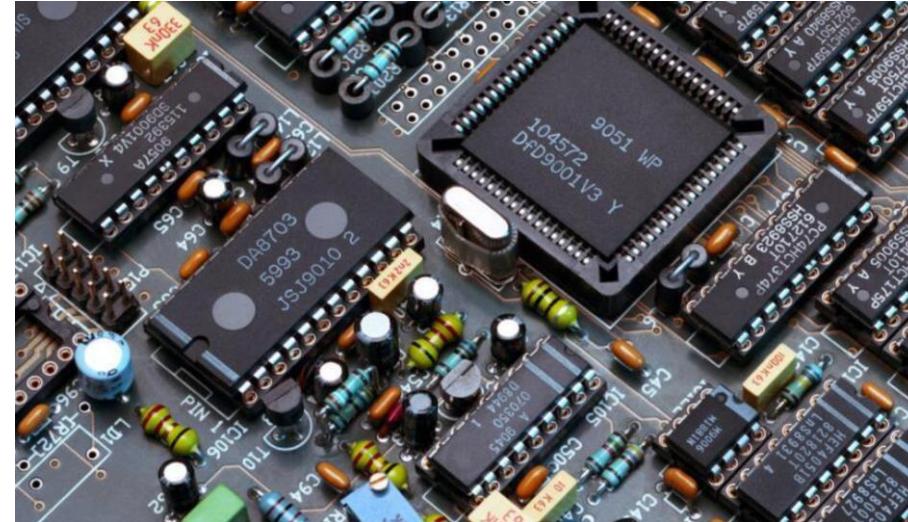
1, 2, ..., 8, 9, 10, 11, 12, ...

十进制：满10进1



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

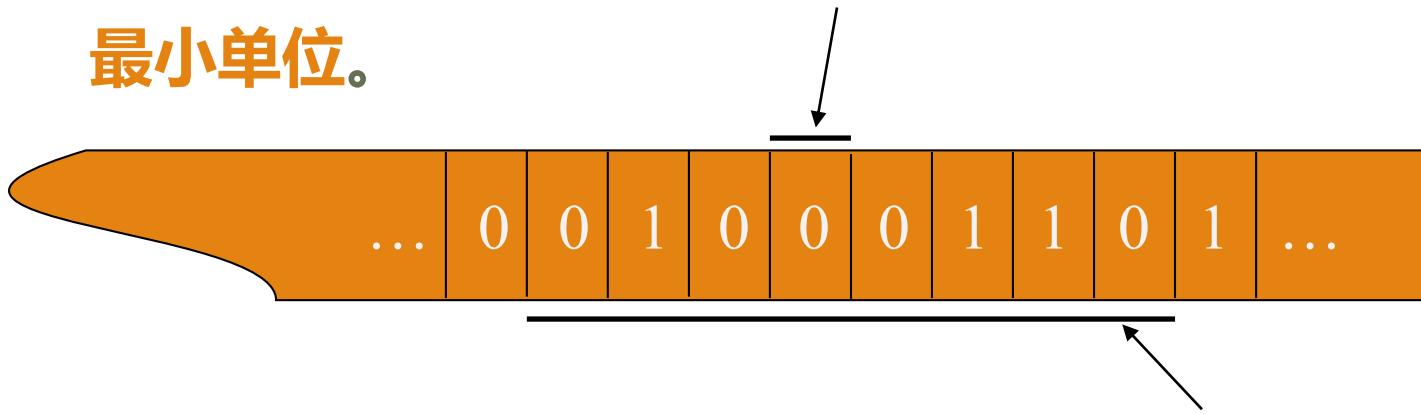
二进制：满2进1



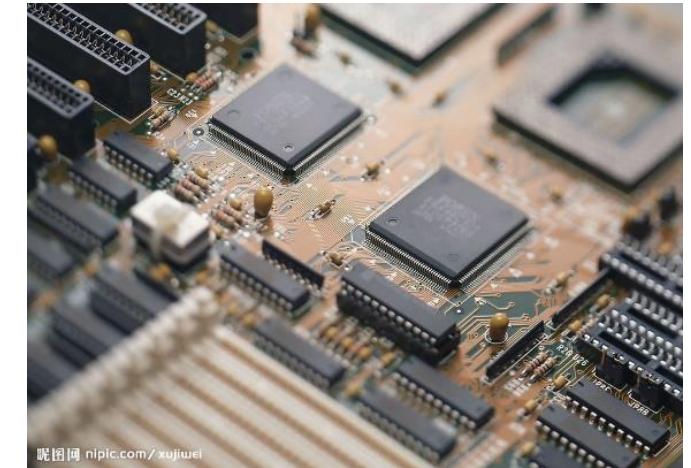
晶体管

# 二进制

- 二进制：数据都是通过 0 和 1 来表示，逢二进一
- 位(bit)：是指二进制中的位，它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（字节）。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。
- 几种数据类型及其通常所占的字节：  
char, 1个字节； int, 4个字节； float, 4个字节； double, 8个字节



# 二进制



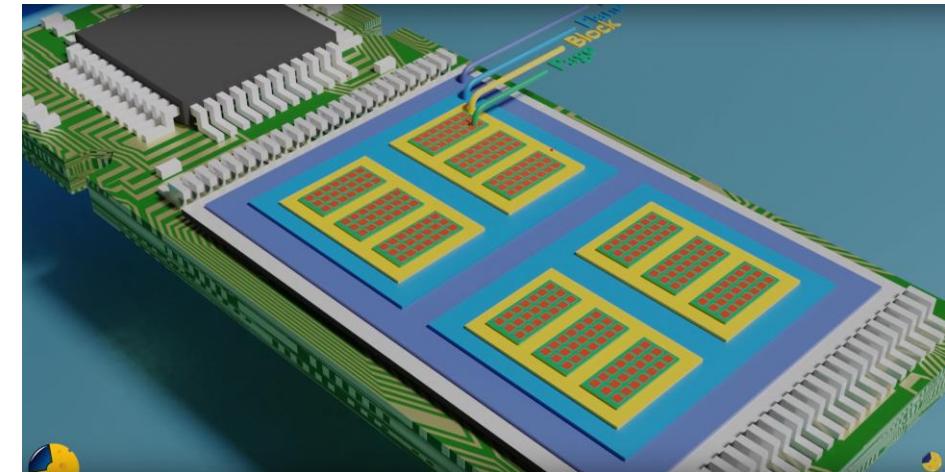
内存地址      内存里存放的数

地址增加 ↓

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| ..1000 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| ..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 |
| ...    | 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 |

1个字节

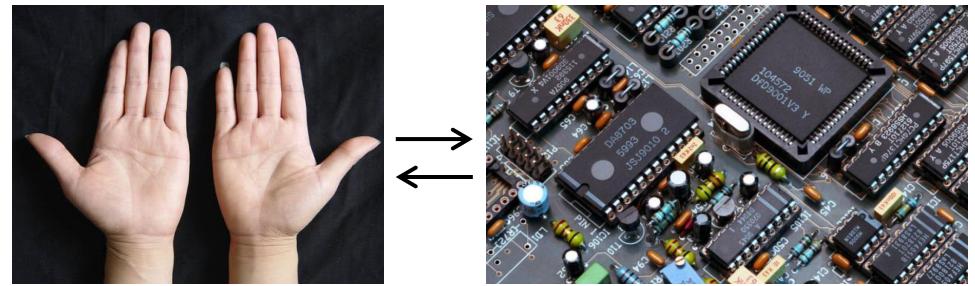
4个字节



- 字符：1个字节
- 整数：4个字节
- ...

# 十进制与二进制

- 人类习惯用十进制表示数值
- 计算机只能对位进行操作，即二进制
- 人与计算机如何交流（十进制vs二进制）



【例3-1】如果已知十进制数  $(19)_{10}$ ，如何用二进制表示？

已知二进制数  $(00010011)_2$ ，如何用十进制表示？

# 十进制转二进制 (10 to 2)

| 进制 | 十进制 | 二进制      |
|----|-----|----------|
| 实例 | 19  | 00010011 |

"十进制"整数转"二进制"数  $(19)_{10} = (10011)_2$

A vertical division algorithm diagram for base 2 conversion of 19. It shows the number 19 at the top, followed by a horizontal line, then the divisor 2, another horizontal line, and the quotient digits 1, 1, 0, 0, 1, 1 below. To the left of the first quotient digit is the remainder 18. To the left of the second quotient digit is the remainder 9. To the left of the third quotient digit is the remainder 4. To the left of the fourth quotient digit is the remainder 2. To the left of the fifth quotient digit is the remainder 1. Below the remainders are the labels "余数" (remainder) and "低位" (low bit). To the right of the quotient digits is the label "高位" (high bit), with a large orange arrow pointing upwards from the bottom to the top of the digits. A red box on the right contains the text "记不住顺序?" (Can't remember the order?).

余数 低位  
高位

除以2取余，逆序排列

"十进制"整数转"十进制"数

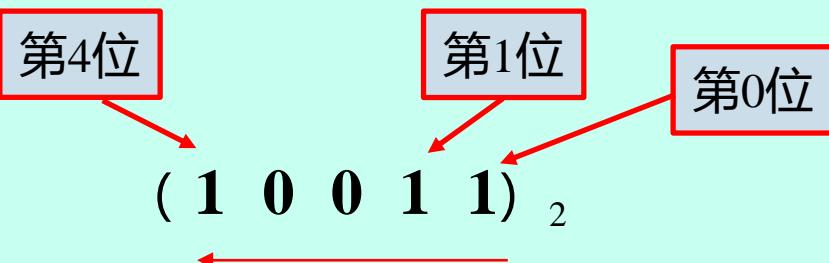
A vertical division algorithm diagram for base 10 conversion of 123. It shows the number 123 at the top, followed by a horizontal line, then the divisor 10, another horizontal line, and the quotient digits 1, 2, 3 below. To the left of the first quotient digit is the remainder 120. To the left of the second quotient digit is the remainder 10. To the left of the third quotient digit is the remainder 1. Below the remainders are the labels "余数" (remainder) and "低位" (low bit). To the right of the quotient digits is the label "高位" (high bit), with a large orange arrow pointing upwards from the bottom to the top of the digits.

余数 低位  
高位

除以10取余，逆序排列

# 二进制转十进制 (2 to 10)

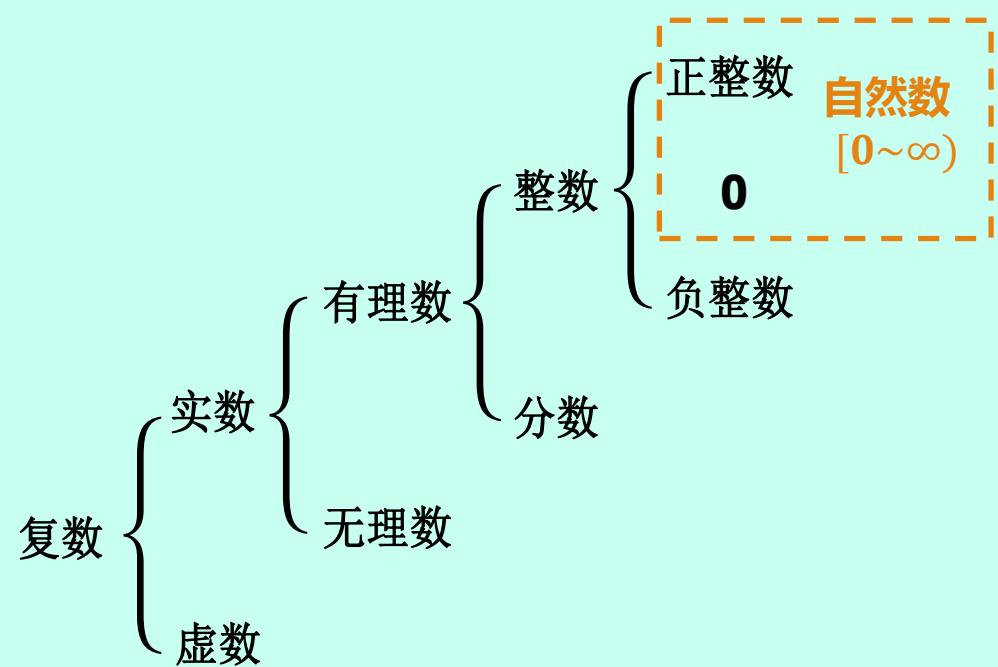
| 进制 | 二进制      | 十进制 |
|----|----------|-----|
| 实例 | 00010011 | 19  |

第4位                  第1位                  第0位  
  
 $(1 \ 0 \ 0 \ 1 \ 1)_2$

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

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

类比十进制:  $123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$



只解决了非负整数的二进制表示?  
负数怎么办?  
小数怎么办?  
复数怎么办?

# 整数的二进制编码：原码，反码，补码

以一个字节，也就是8位二进制表示整数为例：

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

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

## 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：  
 $-127 \sim 127 \Leftrightarrow (-2^{8-1} + 1 \sim 2^{8-1} - 1)$



### 原码的不足：

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。

### (1) 0 的表示不唯一

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

### (2) 加减运算需要识别符号位, 不适合计算机的运算

# 整数的二进制编码：原码，反码，补码

若以两个字节，也就是16位二进制表示整数为例：7与-7的二进制表示

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

高字节 (高8位)                                   低字节 (低8位)

## 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：

$$-32767 \sim 32767 \quad \longleftrightarrow \quad (-2^{16-1} + 1 \sim 2^{16-1} - 1)$$

## 整数的二进制编码：原码，反码，补码

若以四个字节，也就是32位二进制表示整数为例：7 与 -7 的二进制表示

|    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +7 | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0  | 0   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |   |   |
| -7 | <table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table> | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1  | 0   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |   |   |

为简化，后文中许多示例以 1 个字节的数据表示为例进行分析。

# 整数的二进制编码：原码，反码，补码

## 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 原码与反码相同

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

-7 反码：对 7 的原码取反

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

原码：+0

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |
|---|---|---|---|---|---|---|---|

## 反码的不足：

- 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)

同样浪费一个！

# 整数的二进制编码：原码，反码，补码



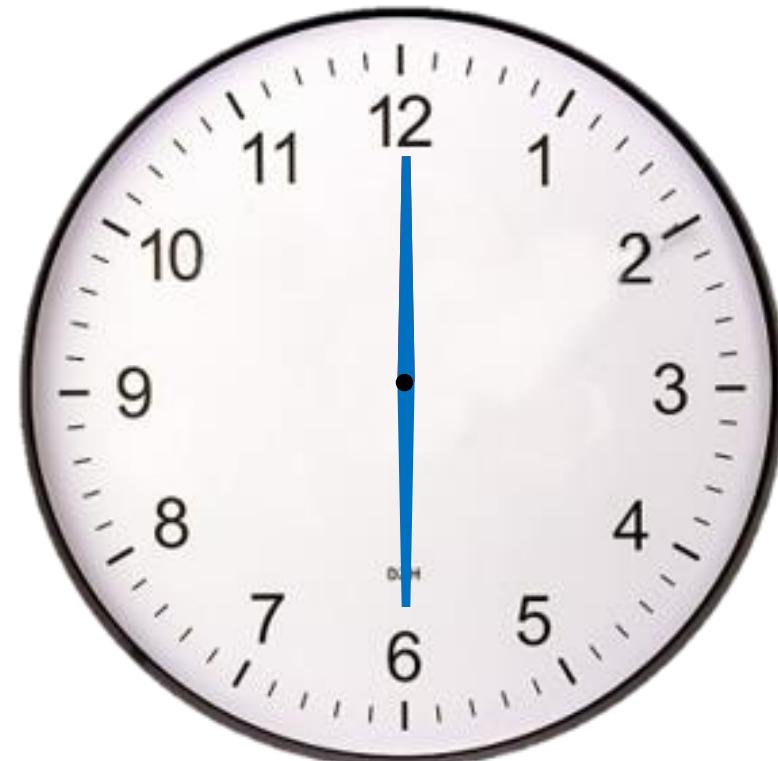
## 从“补数”说起

为了表示负数，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。定义-3和9是关于12的补数。

12

$$X - 3 \leftrightarrow X + 9$$



# 补码

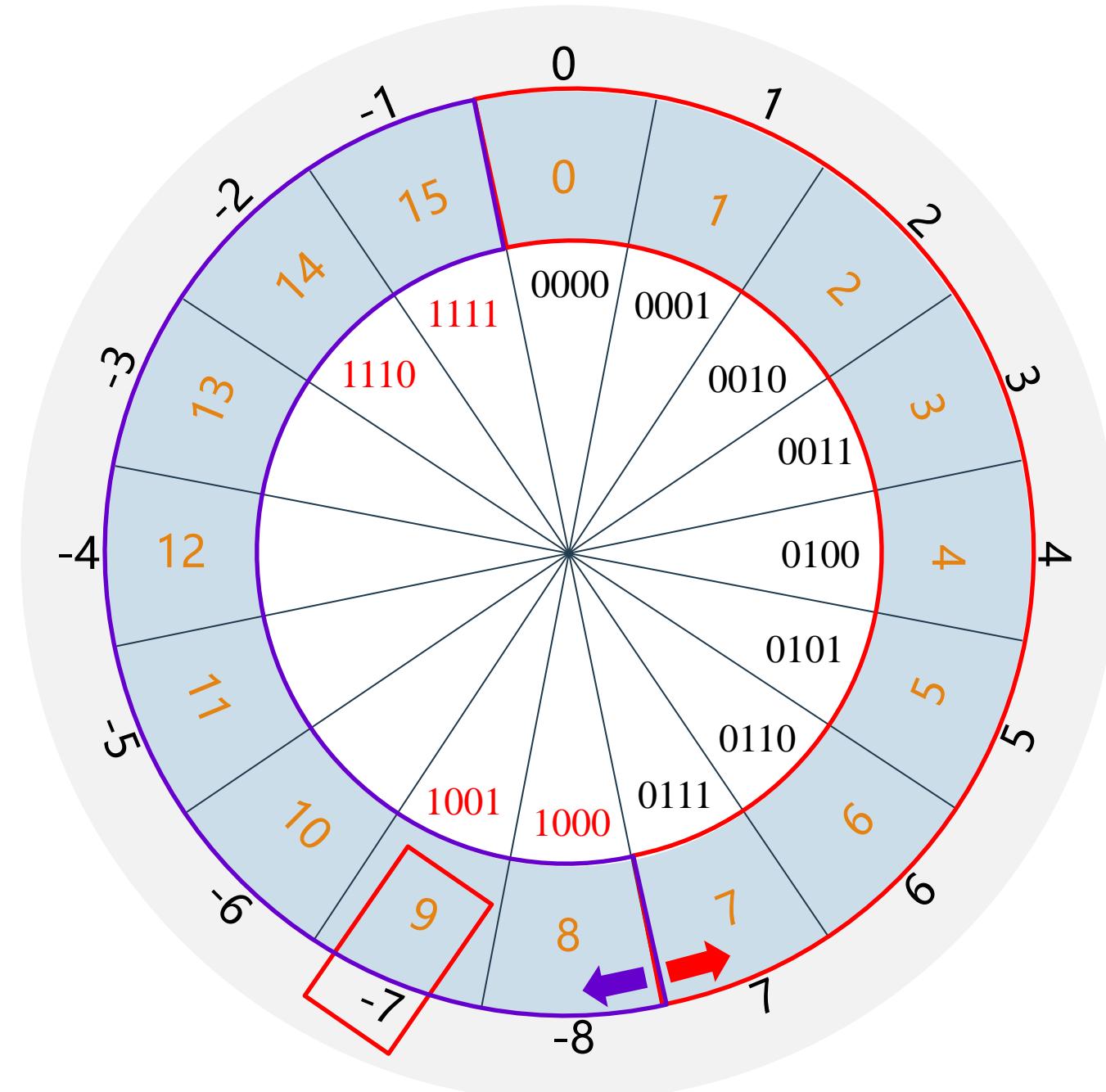
以4位二进制数为例（仅画图示例方便，实际中应为1或2或4或8等字节），共可以表示16个状态，范围从 0000~1111

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 =  $16 - 7 = 9$

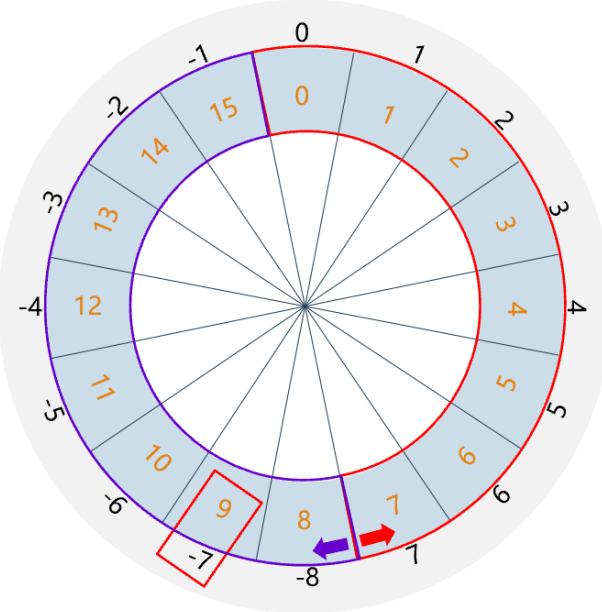
$-x$ 是一个负数，其补数是

$$16-x=\underline{15-x+1}$$

$15-x$ 则相当于在4位二进制下对x各位取反，再加一，即“取反加一”。



# 补码



负数 A 的补数 = 模 - |A|

-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 x 各位取反，再加一，即 “取反加一”。

以 -1 为例

|       |   |   |   |    |   |
|-------|---|---|---|----|---|
| 1     | 1 | 1 | 1 | 15 |   |
| -     | 0 | 0 | 0 | 1  | 1 |
| <hr/> |   |   |   |    |   |
| 1     | 1 | 1 | 0 | 14 |   |
|       |   |   |   |    |   |
|       |   |   |   |    |   |
| 1     | 1 | 1 | 0 | 14 |   |
|       |   |   |   |    |   |
|       |   |   |   |    |   |
| 1     | 1 | 1 | 1 | 15 |   |

15-1, 或者说，对 x 的各位取反

15是-1的补数，在计算机里用这个数的二进制来编码-1，这就是补码

# 补码

- ◆ 正数：原码、反码、补码相同
- ◆ 负数：模减去负数的绝对值，也就是对该负数的绝对值的原码取反，然后对结果加 1（若有进位，则进位被丢弃）（反码 + 1）

+7 原码（反码、补码都一样）

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 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 |
|---|---|---|---|---|---|---|---|

# 补码

- ◆ 正数：原码、反码、补码相同
- ◆ 负数：模减去负数的绝对值，也就是对该负数的绝对值的原码取反，然后对结果加 1 （若有进位，则进位被丢弃）（反码 + 1）

+7 原码（反码、补码都一样）

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 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 |
|---|---|---|---|---|---|---|---|

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

# 补码

- ◆ 正数：原码、反码、补码相同
- ◆ 负数：模减去负数的绝对值，也就是对该负数的绝对值的原码取反，然后对结果加 1（若有进位，则进位被丢弃）（反码 + 1）

+7 原码（反码、补码都一样）

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 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 |
|---|---|---|---|---|---|---|---|

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

# 补码

signed char (带符号的字符) 表示的数的例子, 如 -56

56的原码:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

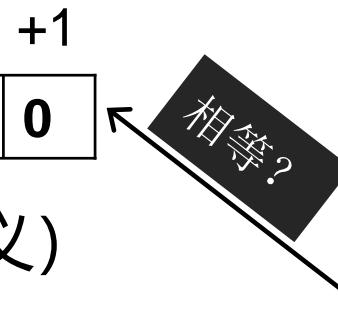
取反:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

加1:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

这是 -56 的补码 (定义)



若用一个字节表示一个整数时, 其模为256 (一个圆周的表盘有256个刻度), -56的补数为200, 即, 逆时针旋转56与顺时针旋转200指向同一个位置 (是同一个数)。

# 补码

signed char (带符号的字符) 表示的数的例子, 如 -56

56的原码:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

取反:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

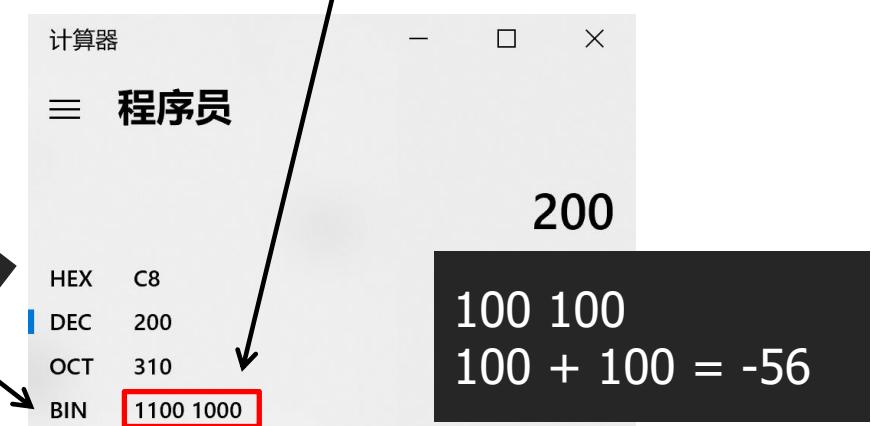
加1:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

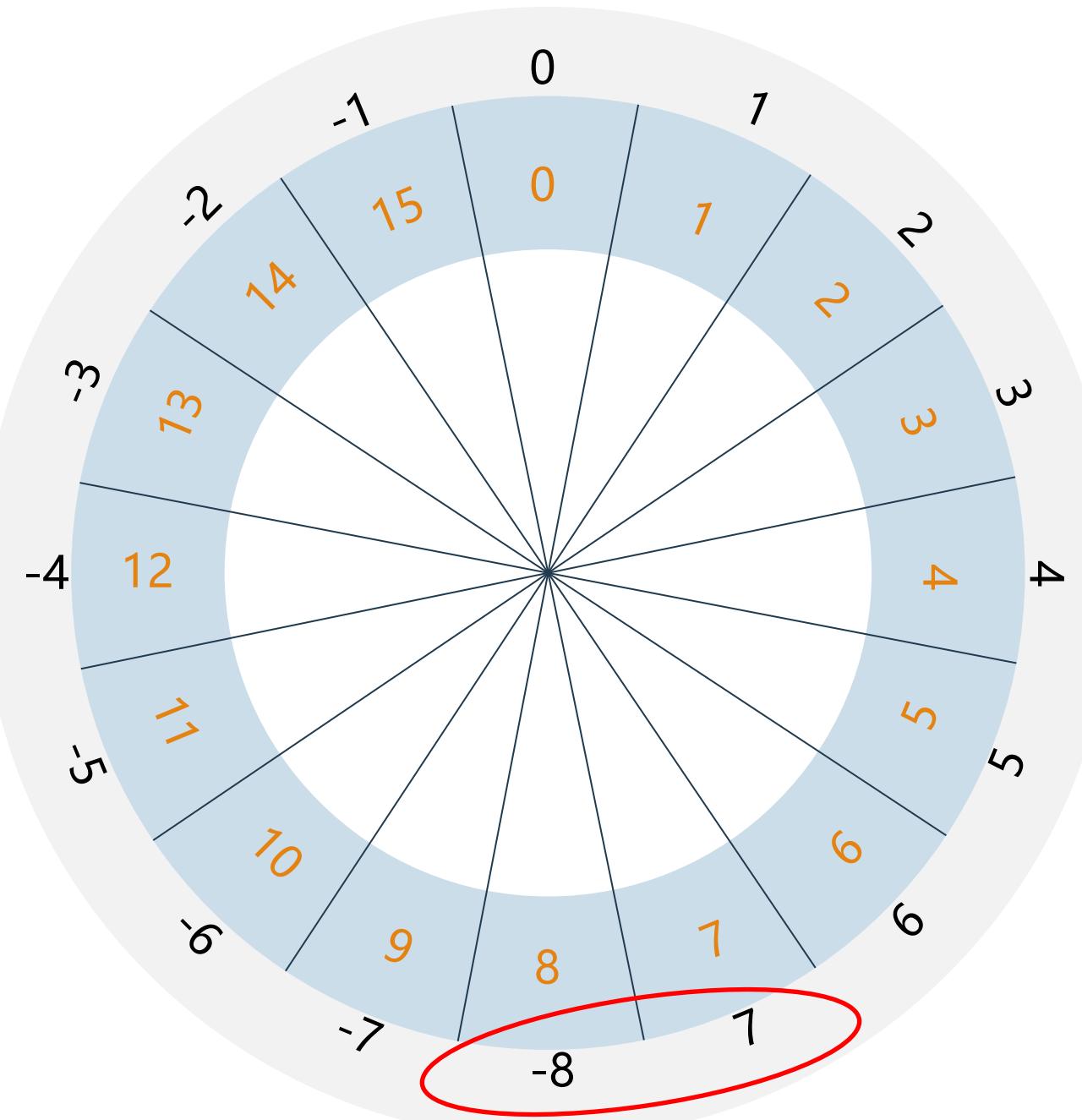
这是 -56 的补码 (定义)

想输出200, 却显示-56的原因:

若用两个字节表示数, 则200的补码为  
00000000 11001000 (计算器上显示没错)  
, 当用8位表示数时 (signed char的位宽)  
, 截取200的低8位, 为11001000, 就是-56



若用一个字节表示一个整数时, 其模为256 (一个圆周的表盘有256个刻度), -56的补数为200, 即, 逆时针旋转56与顺时针旋转200指向同一个位置 (是同一个数)。



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  $\rightarrow$  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  $\rightarrow$  int)

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

## 两个比较特殊的例子

### 0的表示方式唯一

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 | 0 |
|---|---|---|---|---|---|---|---|

反码:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

+1

补码:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

## 两个比较特殊的例子

### -128能有效表达

以一个字节大小的整数补码表示为例

-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 |
|---|---|---|---|---|---|---|---|

?

0也不行! 128也不行!

~~表示 128? 符号位和其他正数不一致~~

表示-128? 补数:  $256 - 128 = 128$

表示128不妥, 但-128的补数是  
128, 就用它来表示-128吧!

# 补码

以一个字节大小 (8位)  
的整数补码表示为例

**表示范围： -128~127**

| 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     |

# 补码

以四个字节大小（32位）  
的整数补码表示为例

表示范围：  $-2^{31} \sim 2^{31}-1$

| 数值          | 补码           |
|-------------|--------------|
| $-2^{31}$   | 1000...0000  |
| $-2^{31}+1$ | 1000...0001  |
| ...         | ... (往上不断减1) |
| -2          | 1111...1110  |
| -1          | 1111...1111  |
| 0           | 0000...0000  |
| 1           | 0000...0001  |
| 2           | 0000...0010  |
| ...         | ... (往下不断加1) |
| $2^{31}-2$  | 0111...1110  |
| $2^{31}-1$  | 0111...1111  |

# 补码

用补码进行运算，减法可以用加法来实现，如  $7-6=1$

|   |   |   |   |   |   |   |   |        |        |
|---|---|---|---|---|---|---|---|--------|--------|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | +7 的补码 |        |
| + | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0      | -6 的补码 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1      | +1 的补码 |
|   |   |   |   |   |   |   |   |        |        |

位溢出，舍去



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！

# 补码

以一个字节大小的整数补码表示为例，

-128表示的另一种理解

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

 -127 的补码

+ 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

 -1 的补码

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

位溢出，舍去

| 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     |

# 二进制编码小结

- 位是计算机处理信息的最小单元
- 位有两种状态0（低电平）和1（高电平）
- 8位构成一个字节，能表达 $2^8$ 种信息（状态）
- 若32位（4个字节）表示一个整数，能表达 $2^{32}$ 种信息（状态）
- 字节是计算机寻址的最小单元
- 二进制是计算机表示数值的方式
- 对于有符号整数，计算机采用补码的形式表示
- 同一个数的补码形式和它占用的字节数有关

## 3.2 | 进制转换

采用八进制（基数8）和十六进制（基数为16）来表示二进制较为方便

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3-2：十进制与八进制和十六进制，如  $(15)_{10}$

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 1 7

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

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                |

采用八进制（基数8）和十六进制（基数为16）来表示二进制较为方便

八进制

|   |   |   |
|---|---|---|
| 0 | 1 | 7 |
|---|---|---|

每个八进制数字的一位对应3位  
二进制位 ( $2^3 = 8$ )

二进制

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

十六进制

|   |   |
|---|---|
| 0 | F |
|---|---|

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"

$$\begin{aligned}(10011)_2 &= (010\overline{011})_2 \\&= (1 \cdot 2^1 + 1 \cdot 2^1 + 1 \cdot 2^0)_8 \\&= (2 \quad 3)_{10}\end{aligned}$$

3位构成  
一组，  
高位不  
够补0

023

"二进制"转"十六进制"

$$\begin{aligned}(10111)_2 &= (0010\overline{111})_2 \\&= (1 \cdot 2^1 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{16} \\&= (2 \quad F)_{16}\end{aligned}$$

4位构成一组，高  
位不够补0

0x2F

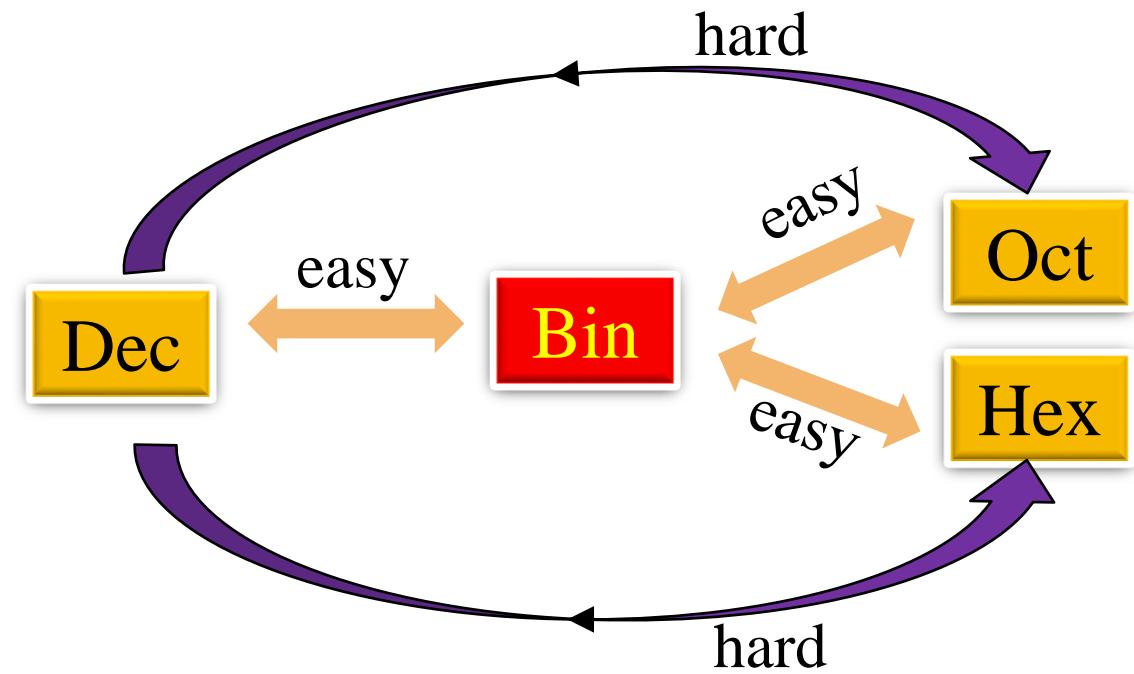
# 十进制 $\leftrightarrow$ 八进制

## A. “十进制” 转 “八进制”



## B. “八进制” 转 “十进制”

$$\begin{aligned}(023)_8 \\ = 2 * 8^1 + 3 * 8^0 \\ = 19\end{aligned}$$



## 其他进制

---

- 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
- ...

### 3.3 | 二进制与位运算

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

### 3.3 二进制与位运算

$$5 \& 3 = 1$$

|       |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|
|       | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 5 |
| &     | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 |
| <hr/> |   |   |   |   |   |   |   |   |   |
|       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

位运算非常重要，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。

# 位运算

**位运算**是直接对数据以**二进制位**为单位进行的运算

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

- 运算对象只能是 **整型** 或 **字符型** 的数据，不能为实型数据
- 位运算符除 ~ (取反) 以外均为二元运算符，~ (取反) 是一元运算符

# & 按位与

## 运算规则

按二进制位进行运算，遵守如下规则

| A | B | A&B |
|---|---|-----|
| 1 | 1 | 1   |
| 0 | 1 | 0   |
| 1 | 0 | 0   |
| 0 | 0 | 0   |

1 保留原来的数值

0 不管原来数值是多少，都置0

运算规则可类比串联电路

例3-3:  $3 \& 5 = 1$

$$\begin{array}{r} 3 \\ & \& 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

应用：可用于实现“清零”操作

## & 按位与

把数  $x$  的特定位置为0，其他位保持不变： $x = x \& ?$

|   |   |   |   |   |   |   |   |   |           |
|---|---|---|---|---|---|---|---|---|-----------|
|   | * | * | * | * | * | * | * | * | x      输入 |
| & | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 43    操作数 |
|   | 0 | 0 | * | 0 | * | 0 | * | * | x      输出 |

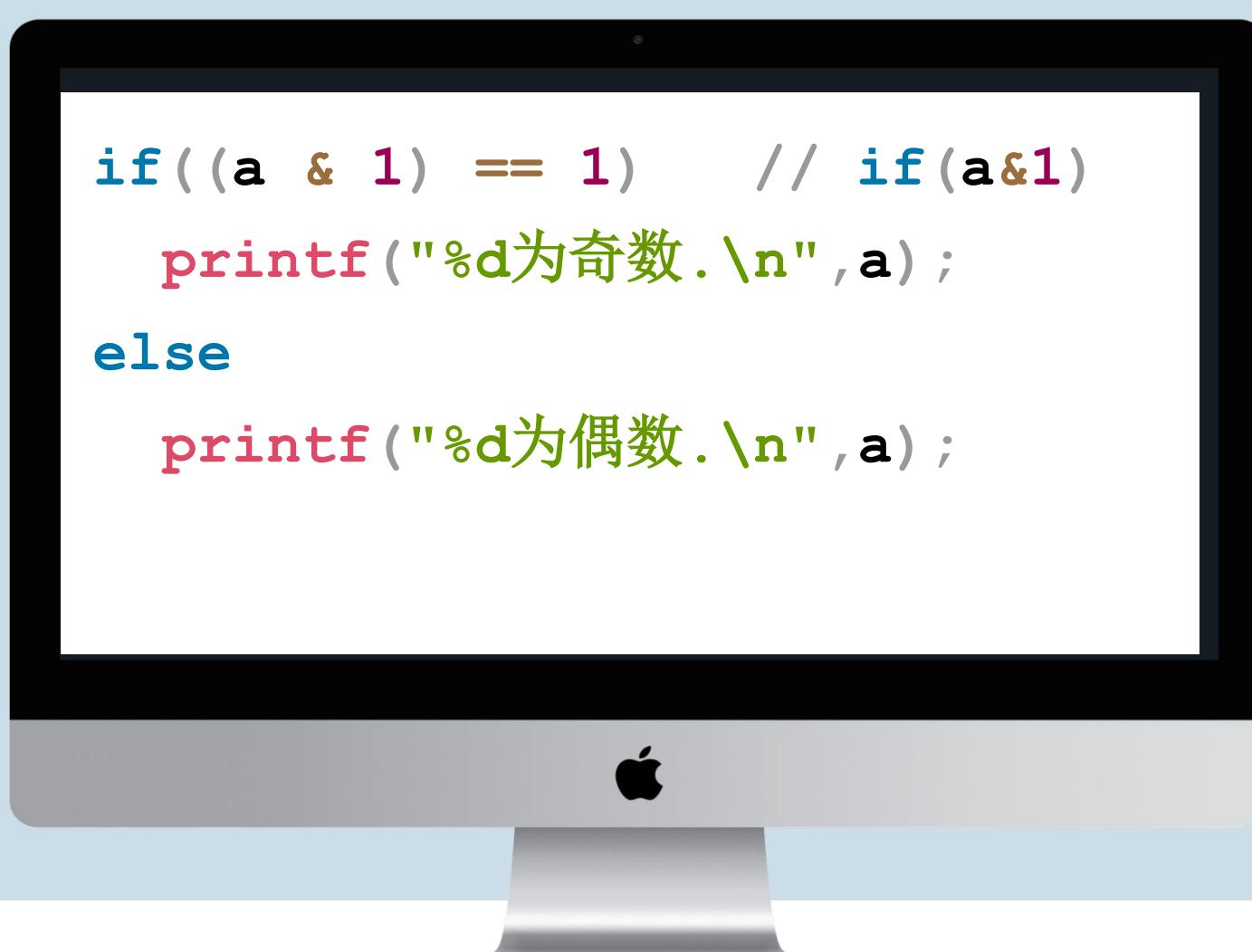
1不变 0清零

上例中保留 $x$ 的第0, 1, 3, 5位，其他位置为零。更通用的实现方式：  
 $x \& (1 | 1 << 1 | 1 << 3 | 1 << 5)$  【稍后学习左移  $<<$ 】

## & 按位与

### 【例】判断奇偶性的一种方法

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



# I 按位或

## 运算规则

| A | B | A B |
|---|---|-----|
| 1 | 1 | 1   |
| 0 | 1 | 1   |
| 1 | 0 | 1   |
| 0 | 0 | 0   |

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

【例3-4】  $3 | 5 = ?$

3      

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

| 5      

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

---

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

应用：可用于实现“置一”操作

# I 按位或

把  $x$  的特定位置为1：  $x = x | ?$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| * | * | * | * | * | * | * | * |
|---|---|---|---|---|---|---|---|

$x$  输入

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

43 操作数

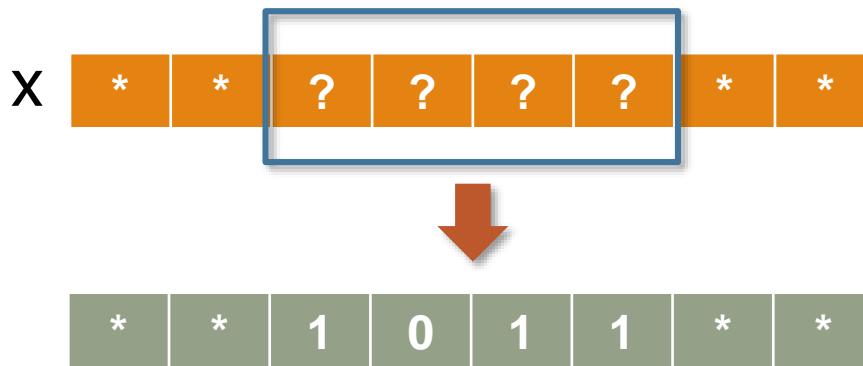
|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| * | * | 1 | * | 1 | * | 1 | 1 |
|---|---|---|---|---|---|---|---|

$x$  输出

0不变 1置一

# I 按位或和按位与的综合范例

【例3-5】把 X 的2至5位设置为特定数，其他位保持不变（以一个字节表示数为例）



移花接木

1.  $X = X \& ((1<<7)|(1<<6)|(1<<1)|1);$
2.  $Y = (1 << 5) | (1 << 3) | (1 << 2);$
3.  $X = X | Y;$

|                   |                                      |
|-------------------|--------------------------------------|
| * * ? ? ? ? *     | X                                    |
| & 1 1 0 0 0 0 1 1 | $(1<<7)   (1<<6)   (1<<1)   1$       |
| <hr/>             | 移花                                   |
| * * 0 0 0 0 * *   | $X = X \& ..$                        |
| 0 0 1 0 1 1 0 0   | $Y = (1 << 5)   (1 << 3)   (1 << 2)$ |
| <hr/>             | 移花                                   |
| * * 1 0 1 1 * *   | $X = X   Y$                          |

## ^ 按位异或

### 运算规则

| A | B | $A \wedge B$ |
|---|---|--------------|
| 1 | 1 | 0            |
| 0 | 1 | 1            |
| 1 | 0 | 1            |
| 0 | 0 | 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

【例3-6】  $3 \wedge 5 = ?$

$$\begin{array}{r} 3 \\ \wedge 5 \\ \hline \end{array} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ \hline \end{array}$$

应用：可用于实现“翻转”操作

^

## 按位异或

例：把 x 的特定位翻转： $x \wedge *$

|   |    |    |    |    |    |    |    |    |             |
|---|----|----|----|----|----|----|----|----|-------------|
|   | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | x      输入   |
| ^ | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 43      操作数 |
|   | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  |             |
|   | 翻转 | 输出          |

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，对二进制按位取反，  
即将 0 变为 1，1 变为 0

例： $\sim 3 = ?$

3    

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

$\sim 3$ 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

例3-7：将一个数 a 的最低位置为 0，其他位不变

|                   |   |   |   |   |   |   |   |   |    |
|-------------------|---|---|---|---|---|---|---|---|----|
| a                 | * | * | * | * | * | * | * | * | 输入 |
| $\& \sim 1$       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |    |
| <hr/>             |   |   |   |   |   |   |   |   |    |
|                   | * | * | * | * | * | * | * | 0 | 输出 |
| $a = a \& \sim 1$ |   |   |   |   |   |   |   |   |    |

例：对 n 取相反数  $\sim n + 1$  前两周，会这样写  $n * (-1)$

例： // 题解里会有这种写法  
`while(scanf(...) != EOF) { ... }`     $\leftrightarrow$     `while(~scanf(...) { ... })`

&lt;&lt;

## 左移

将一个数的二进制编码位全部 **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， $a = a \ll 2$

$a = 15$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

$a = a \ll 2$  ?

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

&lt;&lt;

# 左移

将一个数的二进制编码位全部 **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,  $a = a \ll 2$

$a = 15$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|



$a = a \ll 2$

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|

||

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

- 高位左移后溢出, 舍弃, 右边空位补0
- 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

&gt;&gt;

## 右移

将一个数的各二进位整体右移若干位，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， $a = a >> 2$

$a = 15$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

$a = a >> 2 ?$

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

&gt;&gt;

## 右移

将一个数的各二进位整体右移若干位，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， $a = a >> 2$

$a = 15$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|



$a = a >> 2$

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|

||

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

右移一位相当于除以2

# 位运算符与赋值运算符的结合使用

`&=, |=, >>=, <<=, ^=`

例：  $a \&= b \leftrightarrow a = a \& b$

# 位运算符与赋值运算符的结合使用

【例3-8】给一个无符号整数 a 的  
bit7 ~ bit17位赋值 937,  
bit21 ~ bit25 位赋值17, 其他位不变

```
scanf("%u", &a);
a &= ~((1<<11) - 1) << 7;
a |= 937<<7;
a &= ~((1<<11) - 1) << 7;
a |= 17<<21;
printf("a = 0x%x\n", a);
```

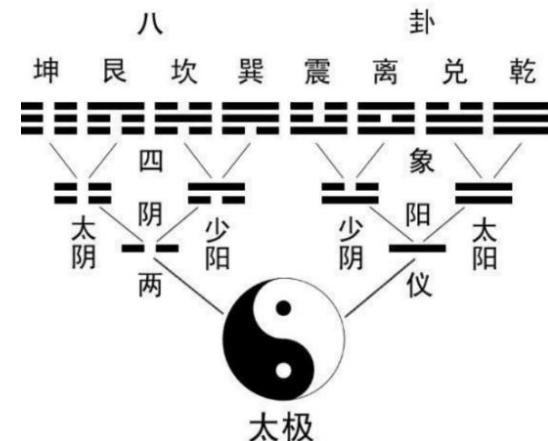
- 1. 初始化低11 (十一) 位为1,  $((1<<11) - 1)$  ,  
得到 0..0111 1111 1111,
- 2.  $((1<<11) - 1) << 7$ ,  
得到 0..011 1111 1111 1000 0000,  
即, 把第1步的十一个1左移7位 (这十一个1变成  
bit7 ~ bit17 )
- 3.  $\sim((1<<11) - 1) << 7)$   
bit7 ~ bit17的十一个1变成0, 其他位的0变成1,  
即变为 1..100 0000 0000 0111 1111
- 4.  $a \&= x$ , 把a的bit7~bit17都置为0, 保留a的其  
他位, ( $x$ 为  $\sim((1<<11) - 1) << 7)$ )
- 5.  $a |= (937 << 7)$ , 把a的bit7 ~ bit17置为937
- 6. bit21 ~ bit25赋值为17, 原理同上, 请填空实现

# 位运算符与赋值运算符的结合使用

【例3-8】给一个无符号整数 a 的  
bit7 ~ bit17位赋值 937，  
bit21 ~ bit25 位赋值17，其他位不变

```
scanf("%u", &a);
a &= ~( ((1<<11) - 1) << 7 );
a |= 937<<7;
a &= ~( ((1<<5) - 1) << 21 );
a |= _____;
printf("a = 0x%x\n", a);
```

无极生太极，太极生两仪，两仪生四象  
，四象生八卦，八卦生万物。



编程与哲理

无生有，有生一，一生二，二生三，三生万物。

从 1 出发，进行位运算，搞定所有复杂应用！

这是一段非常优美的代码！

# 更多的位运算实例

【例】求两个数的平均值：

$$(x + y) \gg 1$$

【例】计算 $2^n$ 次方

$$1 << n$$

【例】从低位到高位，将n的第m位置1

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

【例】从低位到高位，将n的第m位置0

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

【例】计算最大、最小值

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

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

更多位运算应用：<http://graphics.stanford.edu/~seander/bithacks.html#BitReverseObvious>

# 更多的位运算实例

【例】 把一个16位数无符号数的高低字节互换



```
unsigned short a, ans;  
scanf("%u", &a);  
ans = ((a & 0xFF00) >> 8) | ((a & 0xFF) << 8) ;  
printf("%d\n", ans);
```

注:

0xFF 为

0..0 1111 1111

0xFF00 为

0..0 1111 1111 0000 0000

0xFF0000 为

0..0 1111 1111 0000 0000 0000 0000

提示:

按字节整体处理时，用 0xFF 作为基本单元更简单快捷（如本例），若对某些个别位进行处理，“从1出发”更显得逻辑清晰（步骤可能会多些）。

课后思考题：试试这个程序，观察一下结果并分析原因？

```
#include <stdio.h>
int main()
{
    int a = 0x80000001;
    int i;
    for(i=0;i<32;i++)
        printf("left %2d:%08x,%d\n", i, a<<i, a<<i);
    return 0;
}
```

## 实践出真知

养成自己写程序去模拟和  
观察的习惯受用终身。

## 3.4 | 浮点数及数据范围

## 怪象2：0.3 等于 0.3 不成立？

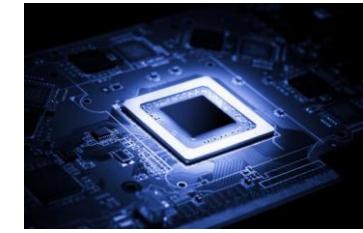
```
// c3-0-2.c
#include <stdio.h>
int main()
{
    int a = 625, b = 3;
    printf("%d, %d\n", (a == 625), (b == 3));

    float x = 0.625, y = 0.3;
    printf("%d, %d", (x == 0.625), (y == 0.3));

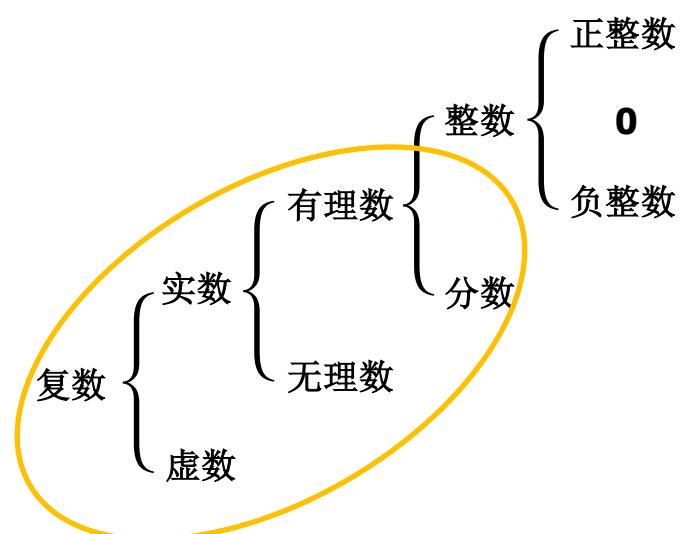
    return 0;
}
```

1, 1  
1, 0

- 计算机的二进制有界，没负数，也没小数
- 十六进制、八进制是用于理解二进制的，所以也有界，没负数
- 但是十进制是人的需求，是无界，有负数和小数的！



- 二进制表达解决了有界非负整数问题
- 补码解决了有界整数（包括负数）问题



**小数怎么表示？  
无穷大怎么办？**

# 小数的二进制表示（数学意义上的表达）

注意：数学表达不等于  
计算机中的编码！

| 进制 | 十进制    | 数学意义的二进制表示  |
|----|--------|-------------|
| 实例 | 19.625 | 00010011101 |

"十进制"整数转"二进制"数

除以2取余逆序排列

$$\begin{array}{r} & \text{余数} \\ \begin{array}{r} 2 \end{array} & \overline{19}_{18} \\ & 1 \\ \begin{array}{r} 2 \end{array} & \overline{9}_8 \\ & 1 \\ \begin{array}{r} 2 \end{array} & \overline{4}_4 \\ & 0 \\ \begin{array}{r} 2 \end{array} & \overline{2}_2 \\ & 0 \\ \begin{array}{r} 2 \end{array} & \overline{1}_0 \\ & 1 \\ & 0 \end{array}$$

(19)<sub>10</sub> = (10011)<sub>2</sub>

小数部分“十进制数”转“二进制数”

乘以2取整顺序排列

$$\begin{array}{r} \text{整数部分} \\ 0.625 \times 2 = 1.25 \dots \quad 1 \\ 0.25 \times 2 = 0.5 \quad 0 \\ \dots \quad 0.5 \times 2 = 1.0 \quad 1 \\ \dots \\ (0.625)_{10} = (0.101)_2 \end{array}$$

# 小数的二进制表示

| 进制 | 十进制  | 数学意义的二进制表示             |
|----|------|------------------------|
| 实例 | 19.3 | 00010011010011001..... |

注意：数学表达不等于  
计算机中的编码！

## 小数部分“十进制数”转“二进制数”

乘以2取整顺序排列

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

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

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

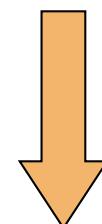
$$0.4 \times 2 = 0.8 \dots \dots 0$$

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

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

...循环了！

高位



低位

$$(0.3)_{10} = (0.0\overline{1001}1 \dots)_2$$

浮点数在计算机中的表达  
不精确，用 == 判断浮点  
数相等时一定要小心！

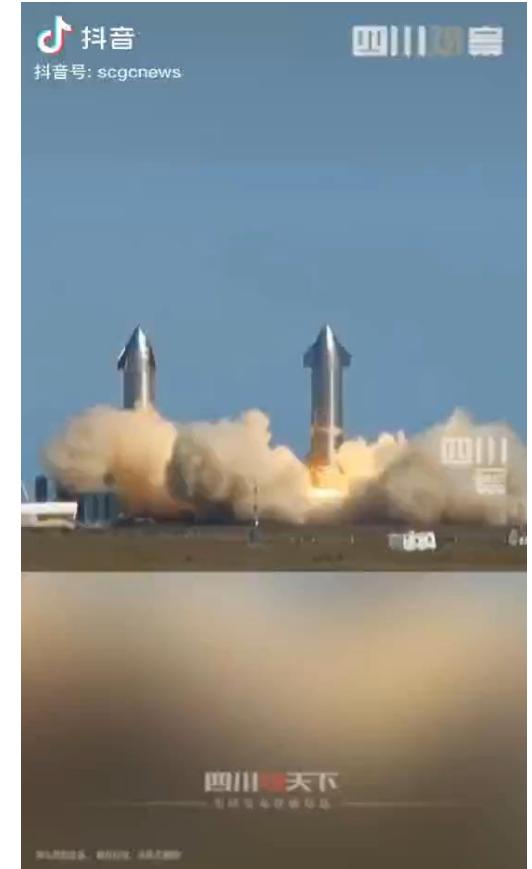
# 小数的二进制表示

```
double b = 0.3;
int a = (int)(b*10);
if( a == 3 )
{
    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

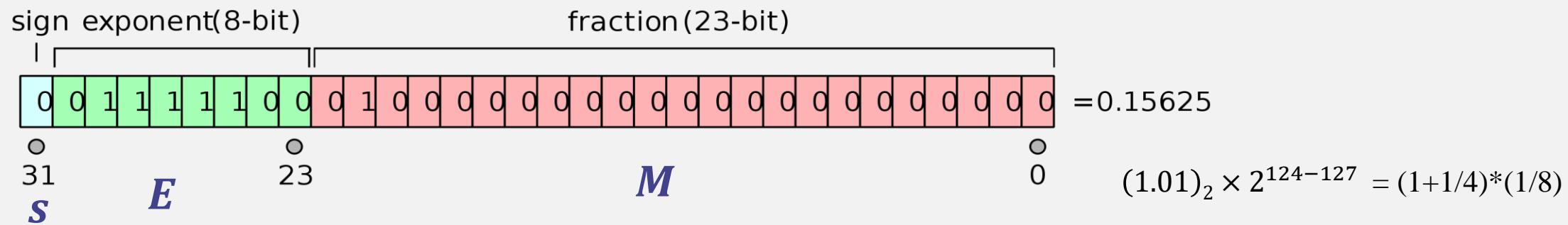
codeblocks  
下编译运行



一行代码可能引发惨剧  
应该点火，却不点火

## \*\*浮点型数据的存储方式与数值范围

小数，又称为浮点数。使用标准数据格式 IEEE-754 进行编码（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中，在存储时将浮点型数据分成：**符号** (sign), **指数部分** (exponent, E) 和 **尾数部分** (mantissa, M)（有的地方也称为fraction “小数”）分别存放。以**32位单精度浮点数**为例：



$$x = (-1)^s \times (\mathbf{1}, M)_2 \times 2^{E-127}$$

浮点数的存储思路是牺牲绝对精度（允许误差）来保证范围。同时，在保证范围的前提下，尽可能保证精度，在精度和范围之间做权衡 Trade Off。

所以，实数编码问题变成了：如何编码才能使得照顾范围的同时让精度尽可能高？

**指数决定范围，尾数决定精度！**

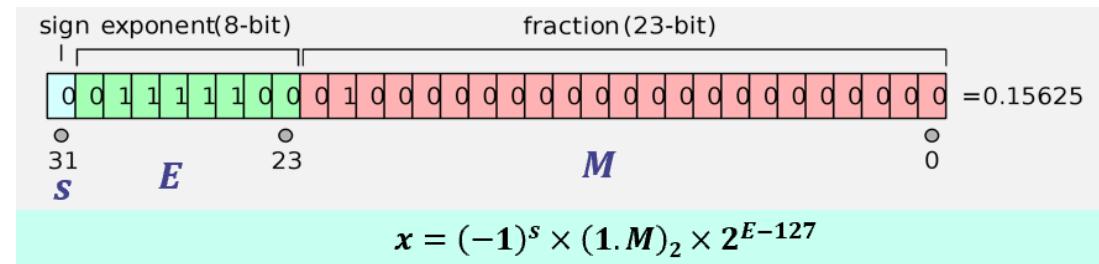
## \*\*浮点型数据的存储方式与数值范围

# 浮点数据类型编码方式

$$x = (-1)^s \times (1.M)_2 \times 2^{E-127} \text{ (float)}$$

$$x = (-1)^s \times (1.M)_2 \times 2^{E-1023} \text{ (double)}$$

上式中  $M$  是二进制,  $E$  是十进制  
!



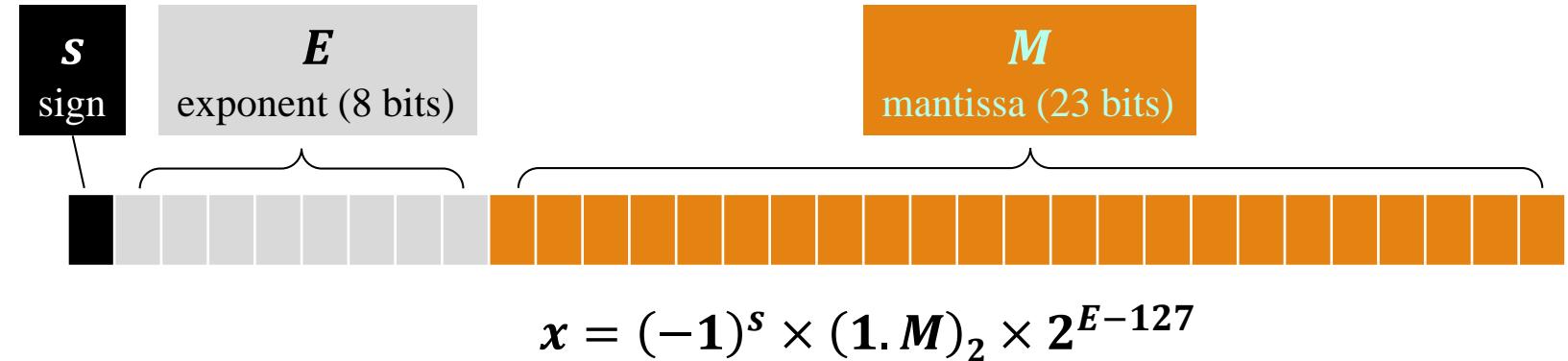
| 浮点数类型  | 符号(+-) | 指数E | 尾数M |
|--------|--------|-----|-----|
| float  | 1      | 8   | 23  |
| double | 1      | 11  | 52  |

**指数决定范围，尾数决定精度！**

范围（数的大小）主要由指数  $E$  决定；精度（小数点的位数）主要由尾数  $M$  决定！

## \*\*浮点型表示实例

IEEE-754 标准数据  
格式 (单精度浮点型)  
)



以 -3.75 为例 (在计算机如何存储)

- (1) 首先把实数转为二进制的  
指数形式
- (2) 整理符号位并进行规范化
- (3) 进行阶码的移码处理
- (4) 因此, -3.75的编码为

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

$$-1.111 \times 2^1 = (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^1$$

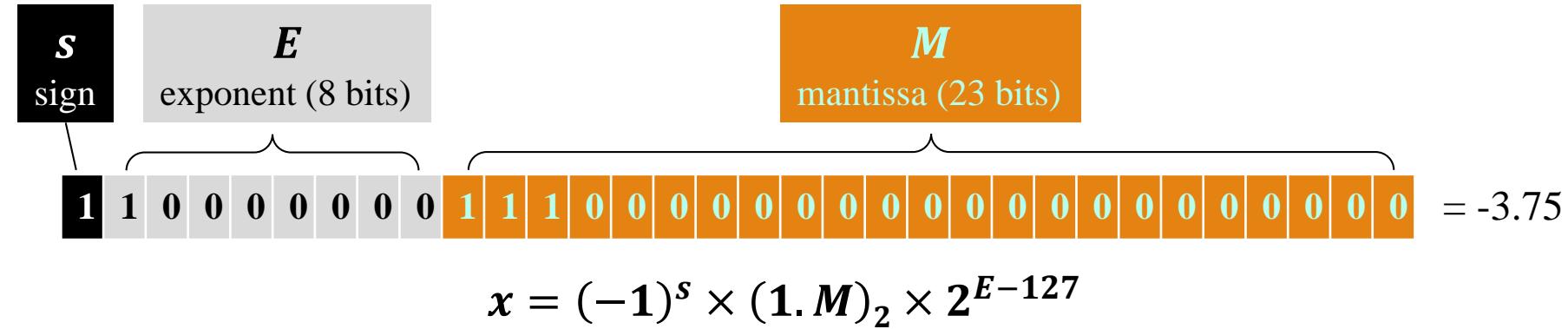
$$(-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^1$$

$$= (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^{128-127}$$

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

## \*\*浮点型表示实例

IEEE-754 标准数据  
格式 (单精度浮点型)  
)



以 -3.75 为例 (在计算机如何存储)

- (1) 首先把实数转为二进制的  
指数形式
- (2) 整理符号位并进行规范化
- (3) 进行阶码的移码处理
- (4) 因此, -3.75的编码为

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

$$-1.111 \times 2^1 = (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^1$$

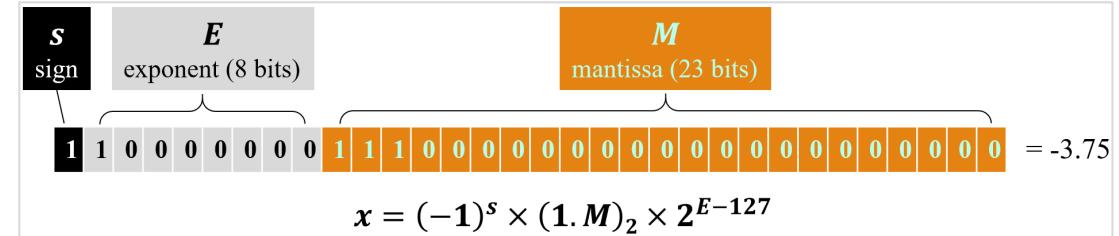
$$(-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^1$$

$$= (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000)_2 \times 2^{128-127}$$

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

## \*\*浮点型数据的精度

**相对精度：机器 $\epsilon$  (machine epsilon)**



表示1与大于1的最小浮点数之差。不同精度定义的机器 $\epsilon$ 不同。以 **double 双精度 (尾数M为52位)** 为例，

数值 1 是：

而比1大的最小双精度浮点数是：

此二者之差为机器 $\varepsilon$ :  $2^{-52} \approx 2.220446049250313e-16$

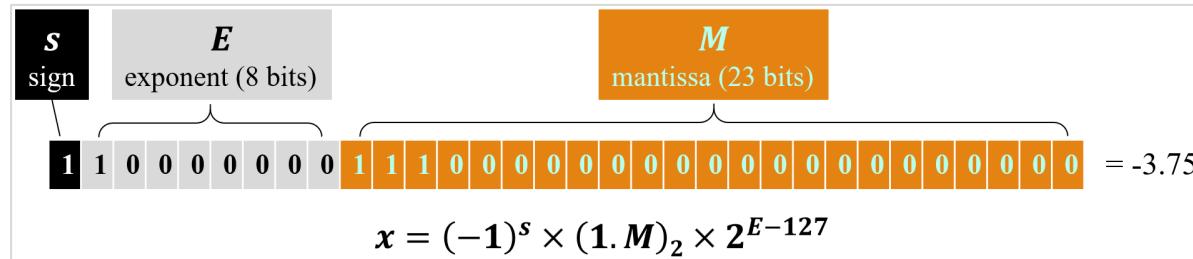
**相对精度是固定的（数值颗粒度）**

# \*\*浮点型数据的精度

## 绝对精度：

$E$  的值 value 越小，此时能够表示的数的范围（或者说，数值的大小）就小，但绝对精度 precision 就高（也就是能够保留的小数点后的数越多）；

反之， $E$  越大，此时能够表示的数的值就越大，但绝对精度就逐渐变小（也就是能够保留的小数点后的数越少）。

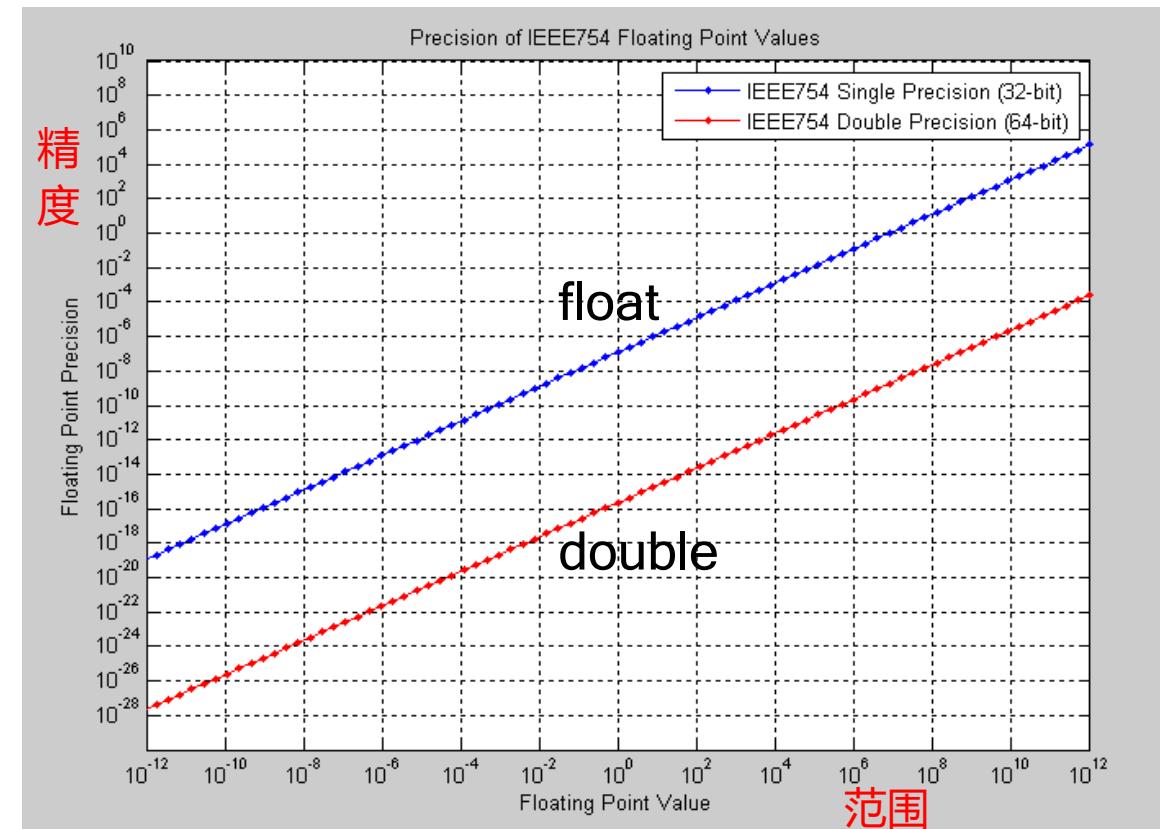


绝对值变化 = 相对精度 \* 范围

范围越大，两个相邻数的绝对值相差（数值颗粒度）

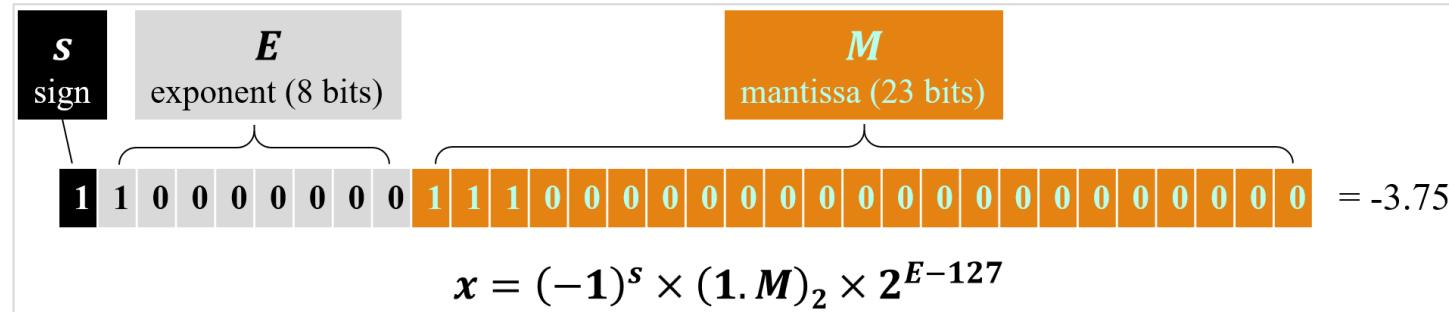
就越大，也就是说，绝对精度就越低。

## float 和 double 类型数据的绝对精度



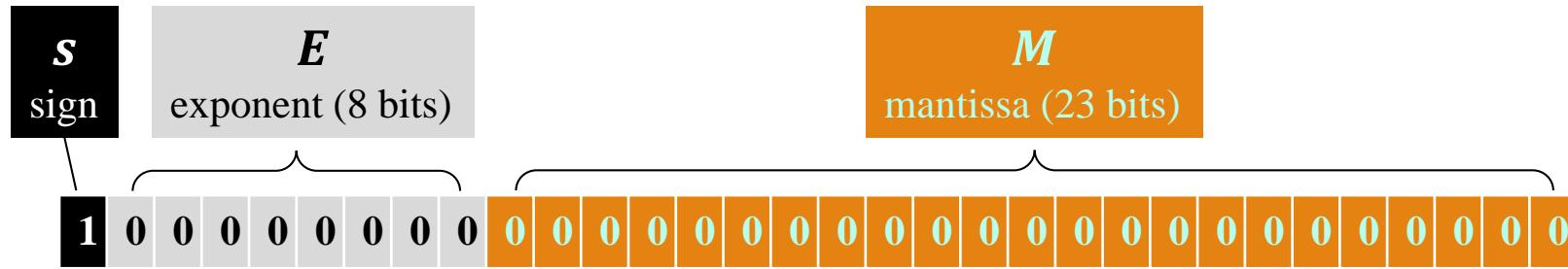
范围小，精度高；范围大，精度低。

# 为什么小数又称为浮点数



- **定点数**: #~#.###~###, 小数点前后的位数是固定的, 位长有限的情况下, 数据表示范围和精度都很小。
- **浮点数**:  $x = (-1)^s \times 1.\# \times 2^{E-127}$ , 任何一个数都被表示为这种形式, 小数点前后的位数不确定, 因此称为浮点数。比如  $s$  为 0,  $M$  全部为 0, 当  $E$  为  $127+5$  时,  $x = 2^5 = 32.0$ , 小数点前有 2 位非 0 数字; 当  $E$  为  $127-5$  时,  $x = 2^{-5} = 0.03125$ , 小数点后精确到 5 位数字。

# 一个特殊的浮点数



$$x = (-1)^s \times (1.M)_2 \times 2^{E-127}$$

$E$  和  $M$  全部取 0 时，

当  $s$  为 0 时， $x = 1 \times 2^{-127}$ ，这表示 +0 的编码

当  $s$  为 1 时， $x = -1 \times 2^{-127}$ ，这表示 -0 的编码

可见，浮点数不能精确表示 0，而是以很小的数（约  $2^{-127}$ ）来近似 0

【例3-9】求  $ax^2+bx+c=0$  方程的解，按如下四种情况处理：

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

浮点型数据在存储和计算时会存在一些微小的误差，因此，对浮点数的大小比较，一般不用“==”或“!=”，而是应该用大于或小于符号。

代码中  $a == 0$  和  $\delta == 0$  这两个地方可能带来问题。  
采取的办法：判别实数之差的绝对值是否小于一个很小的数（比如 $1e-6$ ），则

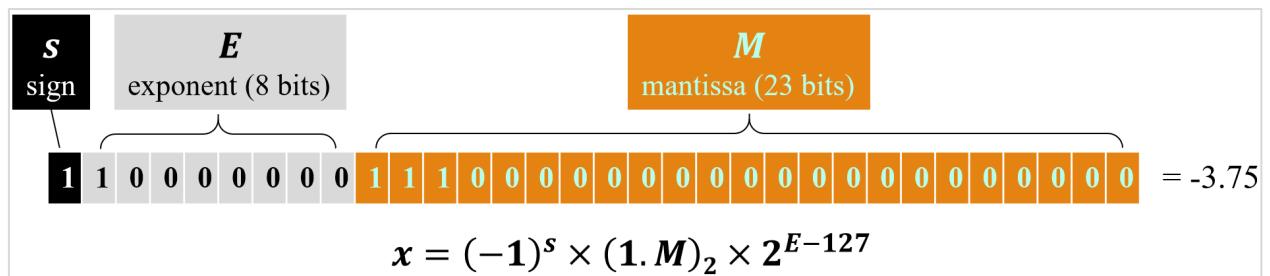
$(\delta == 0)$  可改为  $\text{fabs}(\delta) < \text{eps}$

```
#include <stdio.h>
#include <math.h>
#define eps 1e-10
int main()
{
    double a,b,c,delta,x1,x2,realpart,imapart;
    scanf("%lf%lf%lf",&a, &b, &c);

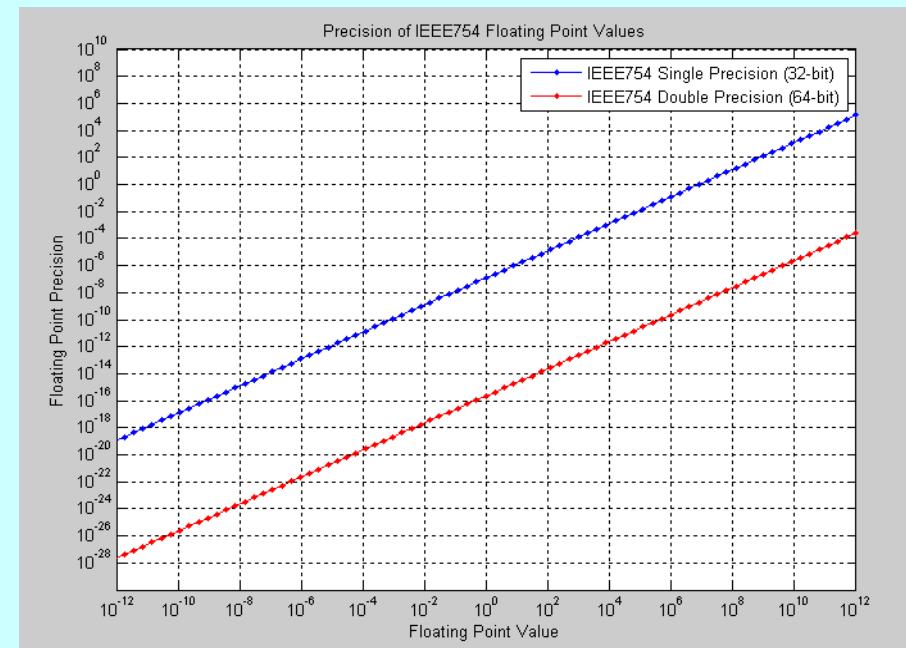
    if(a == 0)
        printf("Not a quadratic");
    else
    {
        delta = b*b-4*a*c;
        if(delta == 0)
            printf("Two equal roots: %8.4f\n",-b/(2*a));
        else if(delta > 0)
        {
            x1 = (-b+sqrt(delta))/(2*a); //a很小时，可能溢出
            x2 = (-b-sqrt(delta))/(2*a);
            // 打印实根(略);
        }
        else
        {
            // 计算、打印虚根(略);
        }
    }
    return 0;
}
```

# 浮点数小结

- 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，表示范围约为：大端  $\pm 3.4 \times 10^{38}$  ( $E$ 为255时)，小端  $\pm 1.1 \times 10^{-38}$  ( $E$ 为0时)？能精确到最小精度约为  $2^{-23} \times 2^{-127} \approx 10^{-44.85}$ ？  
$$\because 2^{10} \approx 10^3 \quad \therefore 128/3.3 \approx 38, 2^{128} \approx 10^{38}$$
- double能表示的范围和精度更大（新标准的C语言基本都抛弃float，只用double了）。
- 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0.99999...999，也可能是1.0000...001。
- 使用浮点数时刻要注意范围和精度问题！



## float和double类型数据的绝对精度



精度高

鱼和熊掌  
不可兼得



范围大

扩大范围损失精度  
照顾精度减少范围

# 整数与浮点数小结



计算机表示数据时，一  
直被两朵乌云笼罩！

鱼和熊掌  
不可兼得  
除非，



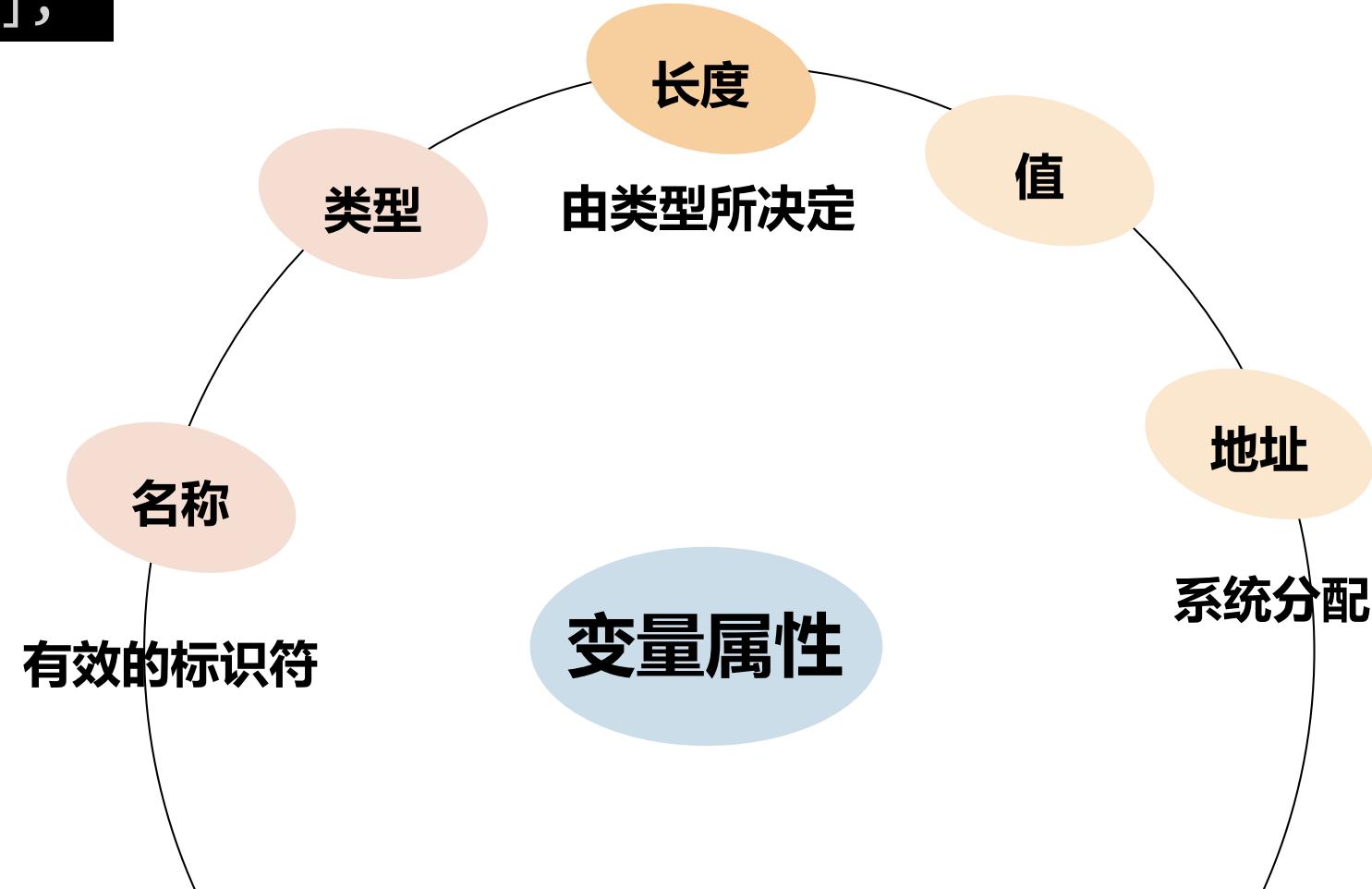
“整数家族”  
char, int, short, long,  
long long, unsigned ...

“浮点数家族”  
float, double, ...

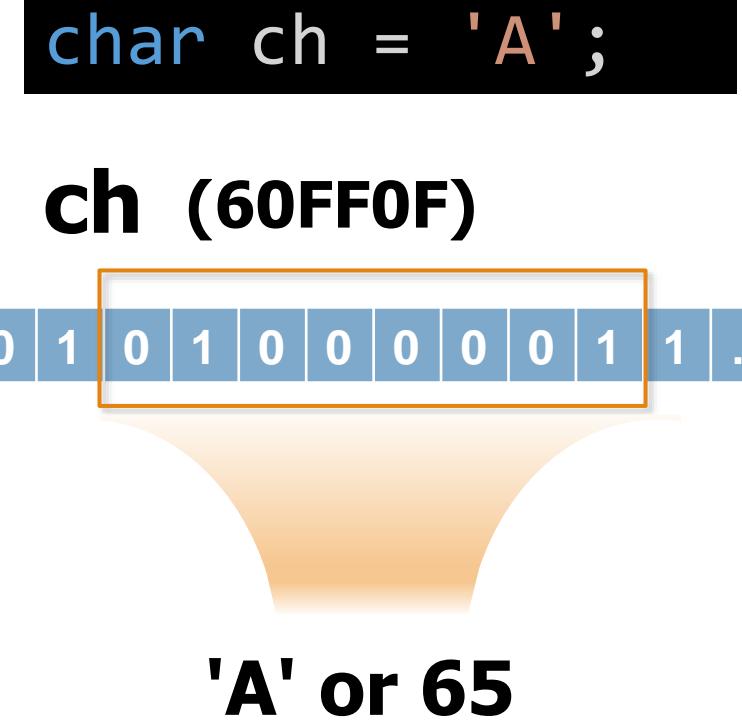
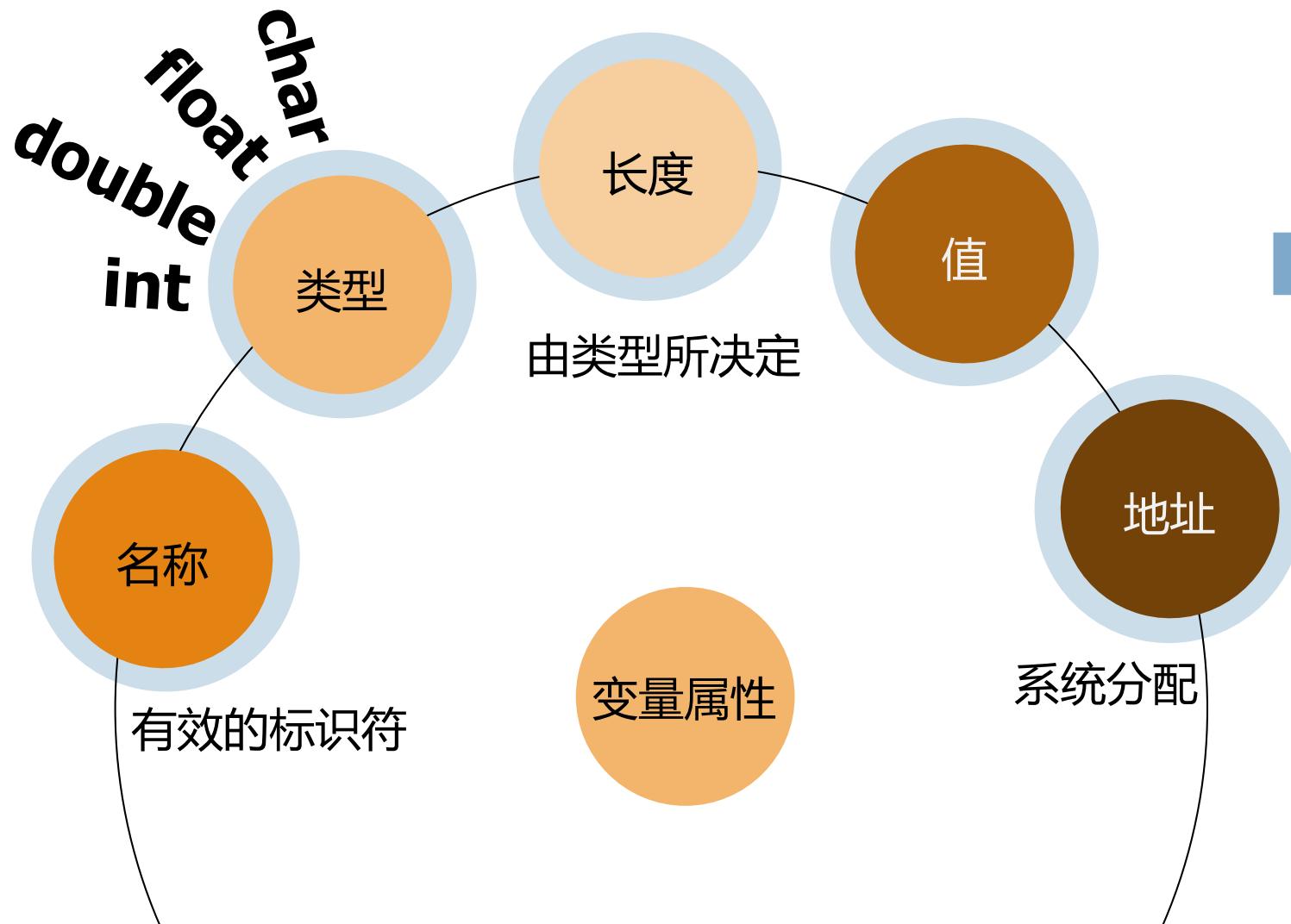
## 3.5 | 变量与内存的关系

# 常用的数据实体：简单变量和数组

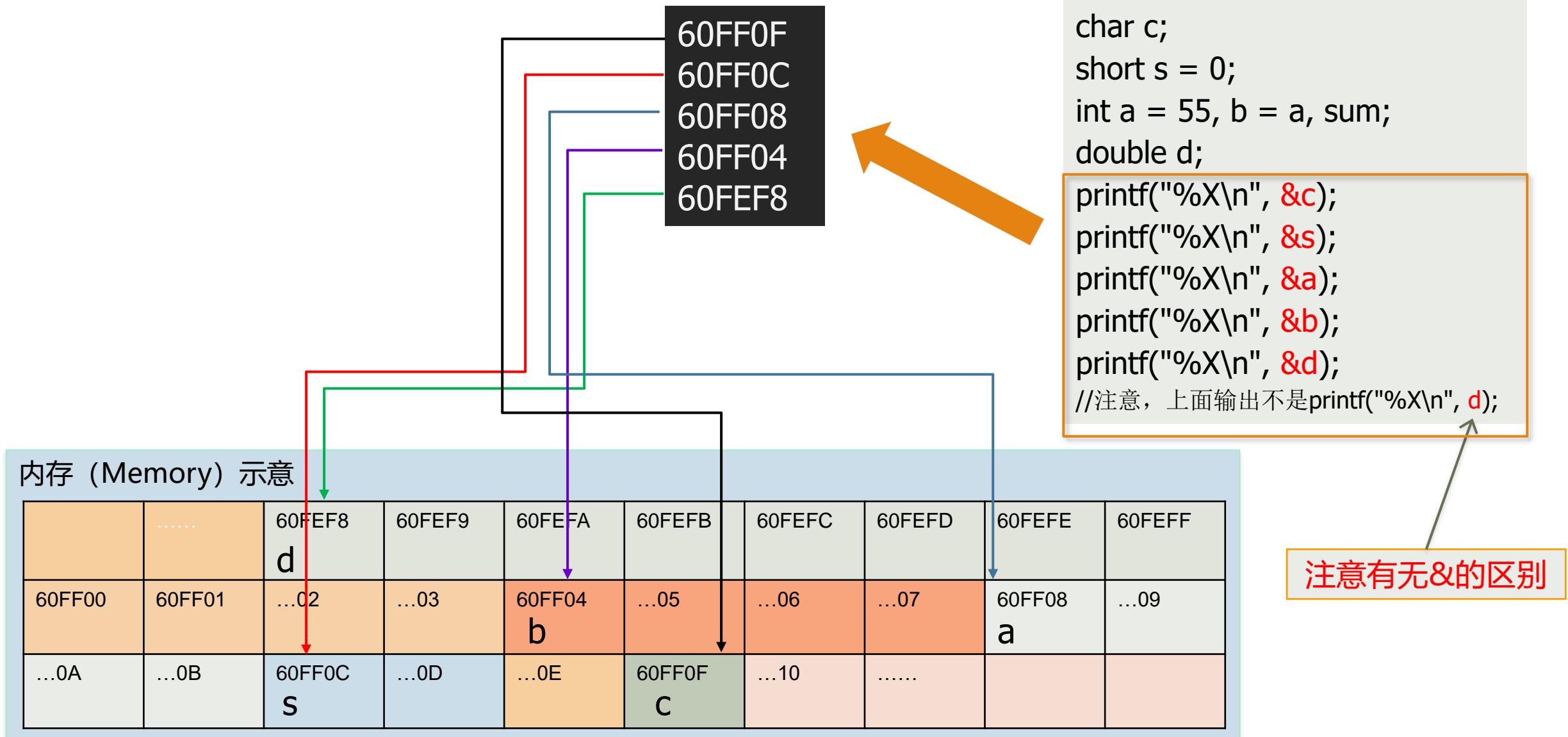
```
char ch = 'A';
float score[1300];
```



# 常用的数据实体：简单变量和数组



# 变量与内存的关系



# 基本数据类型及其通常的存储空间

| 类型            | 字节      | 位  | 有效数字  | 取值范围  |
|---------------|---------|----|-------|---|
| char          | ■       | 8  |       | -128 ~ 127 (- $2^7$ ~ $2^7-1$ )                           |
| int           | ■■■■    | 32 |       | -2147483648 ~ +2147483647 (- $2^{31}$ ~ $2^{31}-1$ )      |
| unsigned int  | ■■■■    | 32 |       | 0 ~ 4294967295 (0 ~ $2^{32}-1$ )                          |
| short int     | ■■      | 16 |       | -32768 ~ 32767  |
| long int      | ■■■■    | 32 |       | -2147483648 ~ 2147483647                                  |
| long long int | ■■■■■■■ | 64 |       | - $2^{63}$ ~ $2^{63}-1$                                   |
| float         | ■■■■    | 32 | 6~7   | 约 $-3.4 \times 10^{38}$ ~ $3.4 \times 10^{38}$ (大端, 详见前文) |
| double        | ■■■■■■■ | 64 | 15~16 | 约 $-1.7 \times 10^{308}$ ~ $1.7 \times 10^{308}$ (大端)     |

从变量内存的角度来  
理解  $100+100=-56$   
的问题

```
int a, b;  
signed char sum = 0;  
  
scanf("%d%d", &a, &b);  
sum = a + b;  
printf("%d + %d = %d\n", a, b, sum);
```

100 100  
100 + 100 = -56

|      |      |      |   |   |   |   |   |   |   |   |
|------|------|------|---|---|---|---|---|---|---|---|
| 0..0 | 0..0 | 0..0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0..0 | 0..0 | 0..0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0..0 | 0..0 | 0..0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

char 类型只占 1 个Byte (8bits), a+b 的高位被截掉

原来如此!

100 a: int, 4 B

100 b: int, 4 B

-56 sum: char, 1 B

# 基本数据类型及其通常的存储空间

| 类型            | 字节       | 位  | 有效数字  | 取值范围  |
|---------------|----------|----|-------|---|
| char          | ■        | 8  |       | -128 ~ 127 (-2 <sup>7</sup> ~ 2 <sup>7</sup> -1)                  |
| int           | ■■■■     | 32 |       | -2147483648 ~ +2147483647 (-2 <sup>31</sup> ~ 2 <sup>31</sup> -1) |
| unsigned int  | ■■■■     | 32 |       | 0 ~ 4294967295 (0 ~ 2 <sup>32</sup> -1)                           |
| short int     | ■■       | 16 |       | -32768 ~ 32767  |
| long int      | ■■■■     | 32 |       | -2147483648 ~ 2147483647  |
| long long int | ■■■■■■■■ | 64 |       | -2 <sup>63</sup> ~ 2 <sup>63</sup> -1                             |
| float         | ■■■■     | 32 | 6~7   | 约 -3.4×10 <sup>38</sup> ~ 3.4×10 <sup>38</sup> (大端, 详见前文)         |
| double        | ■■■■■■■■ | 64 | 15~16 | 约 -1.7×10 <sup>308</sup> ~ 1.7×10 <sup>308</sup> (大端)             |

1. 在C语言中，数据是有范围的；
2. 在不同的系统平台，同一数据类型（如int）范围可能不同。但有个原则是：  
短整型(short)不能长于普通整型(int)；长整型(long)不能短于普通整型(int)。
3. 浮点数使用标准数据格式（IEEE-754）编码。

## 3.6 | 数组基础

数组是在内存中连续存储的一组同类型变量，这些变量统一以**数组名+下标**的形式访问。

```
int a[12] = {1, 3, 5, -2, -4}; // 定义数组，并部分初始化，数组后面的元素自动初始化为0  
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   | 0    | 0    | 0    | 0    | 0    | 0     | 0     |

内存 (Memory)

|        |        |                 |        |        |        |                 |        |        |        |
|--------|--------|-----------------|--------|--------|--------|-----------------|--------|--------|--------|
|        |        | 60FEF8<br>&a[0] | 60FEF9 | 60FEFA | 60FEFB | 60FEFC<br>&a[1] | 60FEFD | 60FEFE | 60FEFF |
| 60FF00 | 60FF01 | ...02           | ...03  | ...    | ...    | ...             | ...    | 60FF24 | ...25  |
| ...26  | ...27  | 60FF28          |        |        |        |                 |        |        |        |

# 数组的类型与大小

```
#define Len 100  
...  
int main  
{  
    int a[Len];  
    double b[Len];  
    char c[Len];  
    ...
```

跟变量一样，可以定义不同类型的数组，如double, char, ...

`sizeof(a)` is `sizeof(int)*Len`

`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。

| 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   | 0    | 0    | 0    | 0    | 0    | 0     | 0     |

|                  |        |        |                 |        |        |        |                 |        |        |        |
|------------------|--------|--------|-----------------|--------|--------|--------|-----------------|--------|--------|--------|
| 内存示意<br>(Memory) |        |        | 60FEF8<br>&a[0] | 60FEF9 | 60FEFA | 60FEFB | 60FEFC<br>&a[1] | 60FEFD | 60FEFE | 60FEFF |
|                  | 60FF00 | 60FF01 | ...02           | ...03  | ...    | ...    | ...             | ...    | 60FF24 | ...25  |
|                  | ...26  | ...27  | 60FF28          |        |        |        |                 |        |        |        |

## sizeof(para) 使用范例

```
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]));
```

输出结果：

```
4, 4  
8  
1  
40, 4
```

实际的程序中，可能涉及到很多数组，而每个数组的数量不一，巧用 sizeof 可以比较方便地维护程序。如定义宏：

```
#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))
```

# 例

```
#include <stdio.h>
#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))
#define FOR(i,s,N) for(i=s; i<N; i++)
int main()
{
    int i;
    double d[7];
    char c[26];
    float f[10];
    for(i=0; i < ArrayNum(d); i++)
    {
        d[i] = sqrt(i+10);
        printf("%f\n", d[i]);
    }
    for(i=0; i < ArrayNum(c); i++)
    {
        c[i] = i + 'a';
        printf("%c ", c[i]);
    }
    printf("\n");
    for(i=0; i<ArrayNum(f); i++) //可用宏 FOR 代替
    {
        f[i] = i * i;
        printf("%f\n", f[i]);
    }
    ...
}
```

在每个循环中，控制循环次数的语句都一样（替换为相应需要处理的数组名），而不用关心每个循环中的实际次数（不需要每个循环处用相应的常量）。程序维护方便，可读性强。

妙用define，但**初学者慎用**，用得太多，程序的可读性可能也不好！

此处的 for 循环头可以用宏  
**FOR(i,0,ArrayNum(f))**  
来代替，但可读性不一定适合每一个人。

## 【\*\*】 #define是一把双刃剑

70年代后期，Steve Bourne在贝尔实验室编写UNIX第7版的shell（命令解释器）时，决定采用C预处理器使C语言看上去更像Algol-68，于是他用了以下一些宏定义：

一段C代码：

```
int compare(char *s1, char *s2)
{
    while(*s1++ == *s2) {
        if(*s2++ == 0) return (0);
    }
    return (*--s1 - *s2);
}
```

丰富的define

```
#define STRING char *
#define IF if(
#define THEN )
#define ELSE }else{
#define FI ;
#define WHILE while(
#define DO ){
#define OD ;
#define INT int
#define BEGIN {
#define END }
```

这样，他就可以像下面这样编写代码：

```
INT compare(s1, s2)
STRING s1;
STRING s2;
BEGIN
    WHILE *s1++ == *s2
    DO IF *s2++ == 0
        THEN return(0);
    FI
OD
return (*--s1 - *s2);
END
```

结果，Bourne shell的影响远远超出了贝尔实验室的范围，这也使得这种类似Algol-68的C语言变型名声大噪。但是有些C程序员对此感到不满。他们抱怨这种记法使得别人难以维护代码。

# 定义数组大小的讨论

- 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（比如几MB），局部数组比较小（通常几十KB）

```
#define LSize 1000000
#define ssize 1000
#include <stdio.h>

int voiceData[LSize]; //函数之外，全局数组

int main()
{
    double stuScore[ssize]; //局部数组
    ...
}
```

**内存是宝贵的计算资源，应合理规划**

# 定义数组大小的讨论【课后读物\*】

c语言中的全局数组和局部数组：“今天做一道题目的时候发现一个小问题，在main函数里面开一个int[1000000] 的数组会提示stack overflow，但是将数组移到main函数外面，变为全局数组的时候则ok，就感到很迷惑，然后上网查了些资料，才得以理解。对于全局变量和局部变量，这两种变量存储的位置不一样。对于全局变量，是存储在内存中的静态区（static），而局部变量，则是存储在栈区（stack）。”这里，顺便普及一下程序的内存分配知识，C语言程序占用的内存分为几个部分：

- (1) 堆区 (heap)：由程序员分配和释放，比如malloc函数；
- (2) 栈区 (stack)：由编译器自动分配和释放，一般用来存放局部变量、函数参数；
- (3) 静态区 (static)：用于存储全局变量和静态变量；
- (4) 代码区：用来存放函数体的二进制代码。

在C语言中，一个静态数组能开多大，决定于剩余内存的空间，在语法上没有规定。所以，能开多大的数组，就决定于它所在区的大小了。

在WINDOWS下，栈区的大小为2M，也就是 $2 * 1024 * 1024 = 2^{21}$  字节，一个int占2个或4个字节，那么可想而知，在栈区中开一个int[1000000]的数组是肯定会overflow的。我尝试在栈区开一个 $2000000 / 4 = 500000$ 的int数组，仍然显示overflow，说明栈区的可用空间还是相对小。所以在栈区（程序的局部变量），最好不要声明超过int[200000]的内存的变量。而在静态区（可以肯定比栈区大），用vs2010编译器试验，可以开 $2^{32}$ 字节这么大的空间，所以开int[1000000]没有问题。

总而言之，当需要声明一个超过十万级的变量时，最好放在main函数外面，作为全局变量。否则，很有可能overflow。

# 用变量定义数组大小\*

```
int n;  
scanf("%d", &n);  
double s[n];  
double x[]; X
```

C89标准数组定义时长度不能是变量，应为常量。也不能定义长度为空的数组。

例：输入整数n，接着输入n个实数，然后把输入逆序输出

```
int main()  
{  
    int n, i;  
    scanf("%d", &n);  
    double s[n];  
  
    for(i=0; i<n; i++)  
    {  
        scanf("%lf", &s[i]);  
    }  
  
    for(i=n-1; i>=0; i--)  
    {  
        printf("%.2f\n", s[i]);  
    }  
    ...
```

|         |       |
|---------|-------|
| 输入样例:   | 输出样例: |
| 3 6 8 9 | 9.00  |
|         | 8.00  |
|         | 6.00  |

用变量定义数组长度，可能有时正确，但有隐患。不同的编译器由于版本不同，有很多扩展功能，可能造成跟C标准并不完全一致。

问：本程序测试样例数据时正确，但提交到OJ却错了？

答：因为，真实的测试数据集，其输入的n可能比较大（如10M个）！

# 用变量定义数组大小\*

**【课后读物\*】** C语言 (C89标准) 不支持动态数组，即数组的长度必须在编译时确定下来，而不是在运行中根据需要临时决定。但C语言提供了动态分配存储函数，利用它可实现动态申请空间。

(1) 在 ISO/IEC9899 标准的 6.7.5.2 Array declarators 中明确规定了数组的长度可以为变量的，称为变长数组 (VLA, variable length array)。（注：这里的变长指的是数组的长度是在运行时才能决定，但一旦决定，在数组的生命周期内就不会再变。）

(2) 在 GCC 标准规范的 6.19 Arrays of Variable Length 中指出，作为编译器扩展，GCC 在 C90 模式和 C++ 编译器下遵守 ISO C99 关于变长数组的规范。

(3) C89是美国标准，之后被国际化组织认定为标准C90，除了标准文档在印刷编排上的某些细节不同外，ISO C(C90) 和 ANSI C(C89) 在技术上完全一样。

一种常见的用法：  
先定义宏常量，以宏常量  
作为数组长度

```
#define LEN 10  
  
int main()  
{  
    double s[LEN];  
    .....  
}
```

## 数组应用实例

---

**【例3-10】** 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。（很有趣的例子！）

```

#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)
    {
        if(islower(c))
            lower[c-'a']++; // if c is 'a', lower[0]++
        else if(isupper(c))
            upper++;
        total++;
    }
    for ( i=0; i<N; i++ )
    {
        if(lower[i] != 0)
            printf("%c: %d\n", i+'a', lower[i]);
    }
    printf("Upper: %d\nTotal: %d\n", upper, total);
    return 0;
}

```



## 妙用两个函数

### getchar()

### islower()

hash变换，把数映射到字符

## 这里用法很巧妙

数组元素的下标与字母的关系

'a' - 'a' → 0 // hash变换，把字符映射到数  
 'b' - 'a' → 1, ... ,

数组元素（整型）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' == 0)
    lower[0]++;
if(c - 'a' == 1)
    lower[1]++;
...

```

课后练习：给出标准输入字符序列，统计有多少个单词？

# 数组应用实例

**【例3-10】**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。（很有趣的例子！）

运行该程序，并以该程序的源代码作为输入，统计该程序的字符数。

```
选择命令提示符
Microsoft Windows [版本 10.0.22000.556]
(c) Microsoft Corporation。保留所有权利。
C:\Users\DELL>d:
D:\>cd a1ac
D:\a1ac>cd example
D:\a1ac\example>cd chap3
D:\a1ac\example\chap3>c3-10 < c3-10.c
a: 9
c: 11
d: 7
e: 21
f: 8
g: 1
h: 4
i: 25
l: 14
m: 1
n: 13
o: 12
p: 13
r: 17
s: 5
t: 16
u: 7
w: 7
y: 1
Upper: 8
Total: 490
```

## 3.7 | 标准输入输出的重定向

# 标准输入/输出 (IO) 的重新定向

- 标准IO在默认情况下均对应控制台（从标准设备层面看，则分别对应键盘和显示器，也可以把标准IO重新定向为文件）。
- 当程序需要对标准输入/输出进行大量读写时，如：需反复从键盘输入大量数据（输入10000个以内的整数，求和、求平均）



大量数据反复测试  
多次重新输入  
重复劳动，且极易出错

.....

```
int i, n, sum=0;  
for (n=0; scanf("%d", &data[n]) != EOF; n++);  
for (i=0; i<n; i++) // show your input  
    printf("%d\n", data[i]);  
printf("\n\n");  
  
// get the sum and average  
for (i=0; i<n; i++)  
    sum += data[i];  
  
printf("  num: %d\n", n);  
printf("  sum: %d\n", sum);  
printf("average: %.2f\n", (float)sum/n );
```

96

85

73

91

...



## 文件重定向

```
freopen("c3-11.dat","r", stdin);
```

```
for (n=0; scanf("%d", &data[n]) == 1; n++);
```

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

```
    printf("%d\n", data[i]);
```

```
printf("\n\n");
```

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

```
    sum += data[i];
```

```
printf("  num: %d\n", n);
```

```
printf("  sum: %d\n", sum);
```

```
printf("average: %.2f\n", (float)sum/n );
```

```
fclose(stdin);
```



## 对标准输入输出文件重新定向

- 示例中不再从键盘读入数据，而是从文件 c3-11.dat 中读入数据，c3-11.dat 就相当于新的 stdin（标准输入）。
- IO 文件重定向的含义：在不对程序输入/输出语句做任何改动的情况下，使程序从指定的文件中读入测试数据（整体读入，分批处理），并将结果写入指定的文件。如：将对键盘和屏幕的读写改为对指定文件的读写操作。
- 好处：对 C 程序内部处理逻辑无任何影响，可避免重复键入测试数据。

# 标准IO的重新定向的3种方法

(1) 在IDE中，可以进行输入输出的设置（略）。

(2) 在命令行模式下，使用重新定向操作符 < 和 >

```
C:\..>programName < data.in > file.out
```

语句作用：在运行 `programName` 时，将 `data.in` 指定为该程序的标准输入文件，将 `file.out` 指定为标准输出文件。

(3) 在程序中使用标准库函数`freopen()`进行标准输入/输出重新定向

```
FILE * freopen(const char *path, const char *mode, FILE *fp)
```

语句作用：关闭由参数`fp`指向的已打开的输入/输出文件，按参数`mode`打开由参数`path`指定的文件，并将其与参数`fp`相关联。

`mode`: "r"、"w" 分别表示重定向后的文件用于"读"、"写"

`fp`: `stdin`、`stdout`，分别表示标准输入和标准输出

例：

```
freopen("file.out", "w", stdout)
```

执行成功后，`printf`、`puts`等函数的输出将不再写到屏幕上，而是写入文件 `file.out` 中。

注意：测试完成，程序正确后，记得把该语句注释或删除（不然OJ 上通不过，因为 OJ 上的输入输出重定向位置跟你的程序中不一样）！

# 标准I/O重定向实例

随机产生数据，输出到文件

c3-11.dat 中，给右边程序用作输入

```
// c3-11-gen-data.c
```

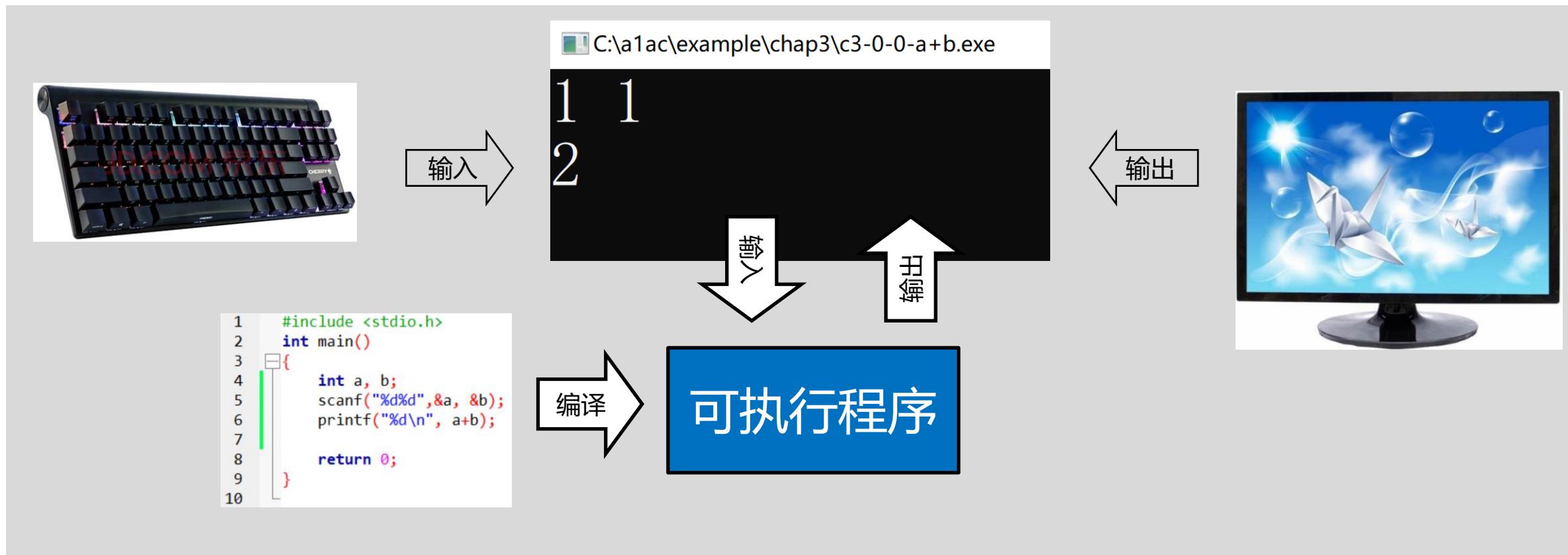
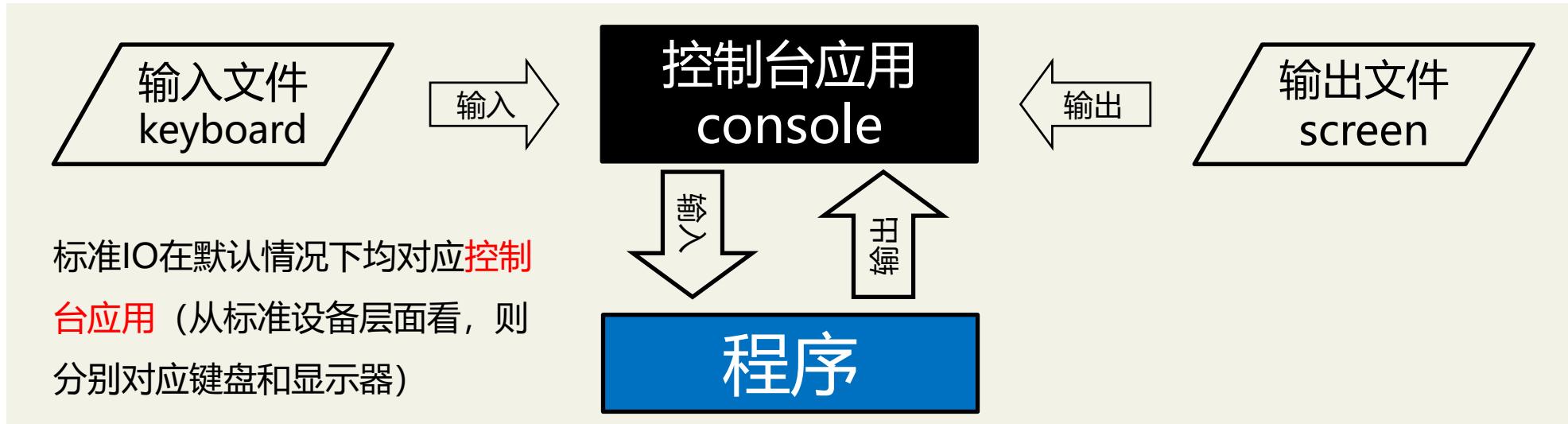
```
freopen("c3-11.dat", "w", stdout); // c3-11.dat 定向为 stdout  
scanf("%d", &n); // 如，输入 1000000 (1M)  
for(i=0; i<n; i++)  
{  
    data = rand()%101; // 随机产生0~100之间的一个数  
    printf("%d\n", data); // 输出到 c3-11.out 中，不是屏幕！  
}  
  
fclose(stdout);
```

当一个程序需要成千上万个输入数据时，手输不现实，用该程序产生随机数据是个好办法！

从文件而不是键盘输入

```
// c3-11-sum.c  
int i, n, sum = 0;  
  
freopen("c3-11.dat", "r", stdin);  
  
for (n = 0; scanf("%d", &data[n]) != EOF; n++)  
  
for (i = 0; i < n; i++)  
    printf("%d\n", data[i]);  
printf("\n\n");  
  
for (i = 0; i < n; i++)  
    sum += data[i];  
  
printf("    num: %d\n", n);  
printf("    sum: %d\n", sum);  
printf("average: %.2f\n", (float)sum / n);  
fclose(stdin);
```

# 标准IO 的重新 定向



# 标准IO的重新定向

也可以把标准IO  
重新定向为文件

```
1 #include <stdio.h>
2 int main()
3 {
4     int a, b;
5     freopen("file.in", "r", stdin);
6     freopen("file.out", "w", stdout);
7
8     scanf("%d%d", &a, &b);
9     printf("%d\n", a+b);
10
11    fclose(stdin);
12    fclose(stdout);
13
14    return 0;
15 }
```

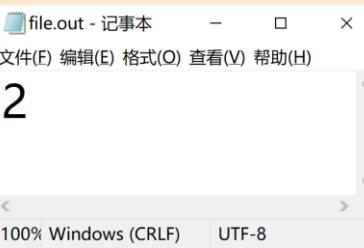
编译



输入

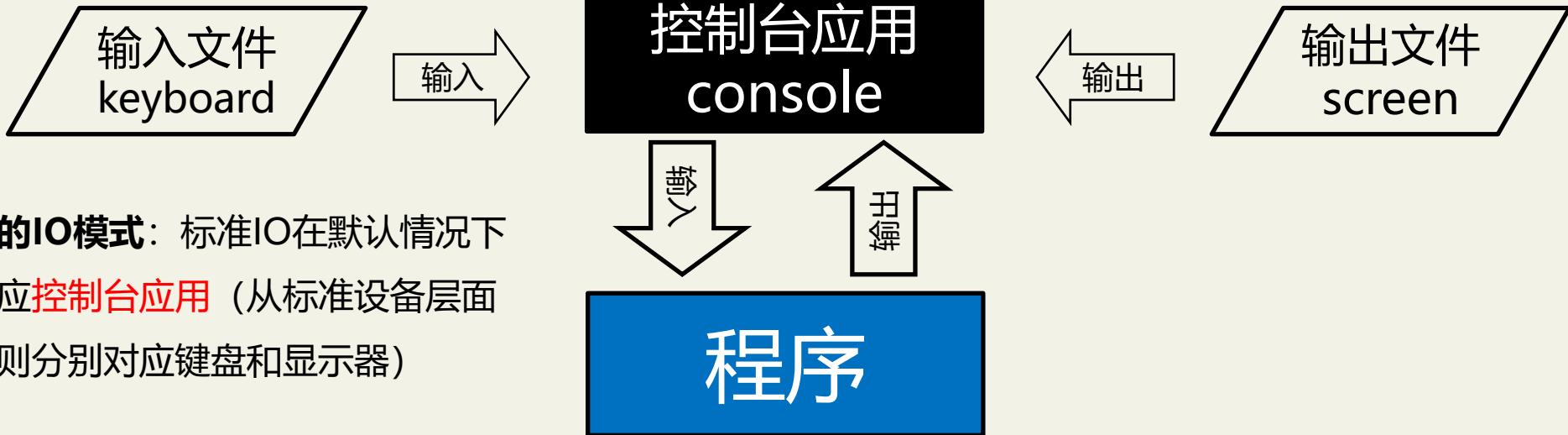
可执行程序

| 名称       | 类型     | 大小   | 修改日期      |
|----------|--------|------|-----------|
| file.out | OUT 文件 | 1 KB | 2021/3/16 |
| file.in  | IN 文件  | 1 KB | 2021/3/16 |



输出

# 标准IO的重新定向



**默认的I/O模式：**标准I/O在默认情况下均对应**控制台应用**（从标准设备层面看，则分别对应键盘和显示器）

**重新定向I/O模式：**也可以把标准I/O重新定向为文件



# 本章小结

---

3.1 数的二进制表示：掌握整数在计算机中的表示（补码）

3.2 进制转换关系：掌握各种进制之间的转换

3.3 位运算：位运算符的含义及功能

3.4 浮点数的表达：初步了解浮点数在计算机中的表示

3.5 变量与内存：掌握变量与内存的关系及各种数据类型的数据范围

3.6 数组基础：了解数组的存储与读取方式

3.7 IO与重定向：理解IO重定向的含义，可运用其更方便地进行程序调试

## 第三讲作业

---

- 为什么float的表示的最大范围约 $-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$  ?
- float能表示的绝对值最小数约为多少?
- 看书, 复习PPT (从开始 ~ 第3讲结束)
- 习题: all
- 预习结构化编程 (判断、循环)
- 上机实践题
  - ◆ 把本课件和书上的所有例程输入计算机, 运行并观察、分析与体会输出结果。
  - ◆ 编程练习课后习题内容。