

第三讲

数据处理基础 data processing



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

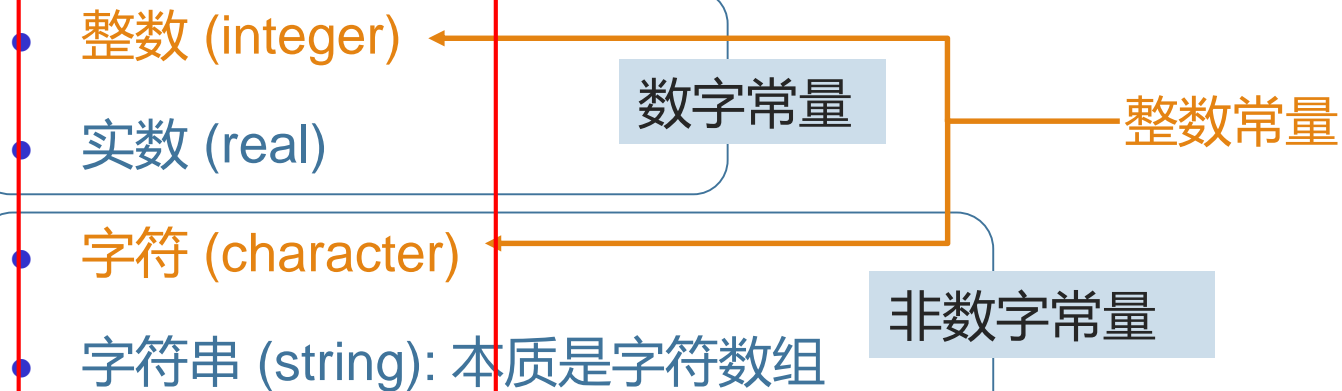
3.7 标准输入输出重定向

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

前情回顾

常量



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



3.1 | 数值在计算机中的表示

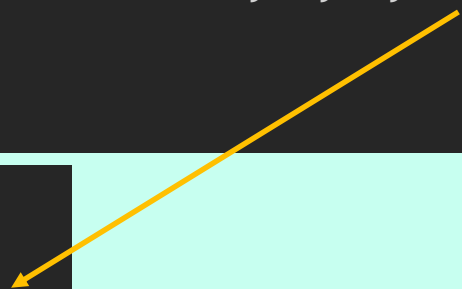
两段有点“奇怪”的代码

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

    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()
{
    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?

两段有点“奇怪”的代码

```
#include <stdio.h>
int main()
{
    int a, b;
    signed char sum = 0;

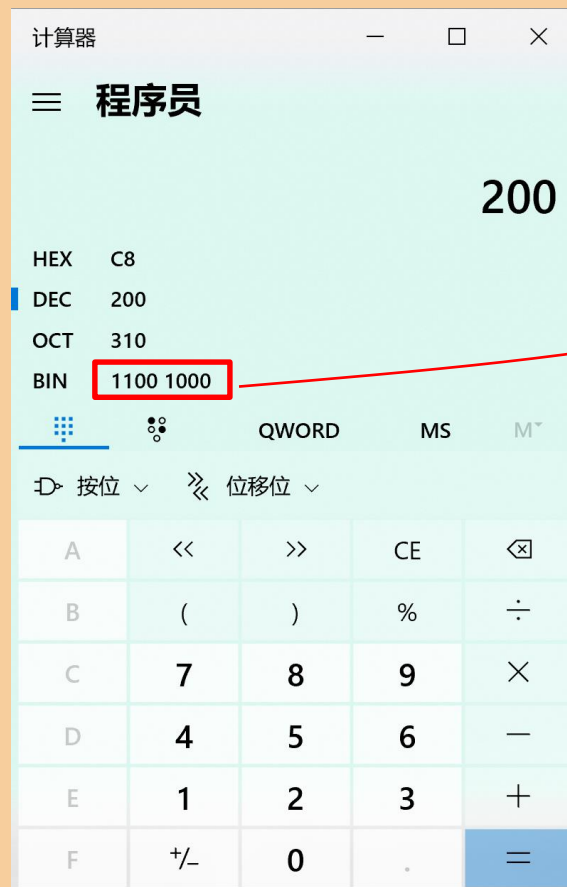
    scanf("%d%d", &a, &b);
    sum = a + b;

    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 (整数的前3个字节被截取掉了), 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;
}
```

更复杂！请认真听讲！

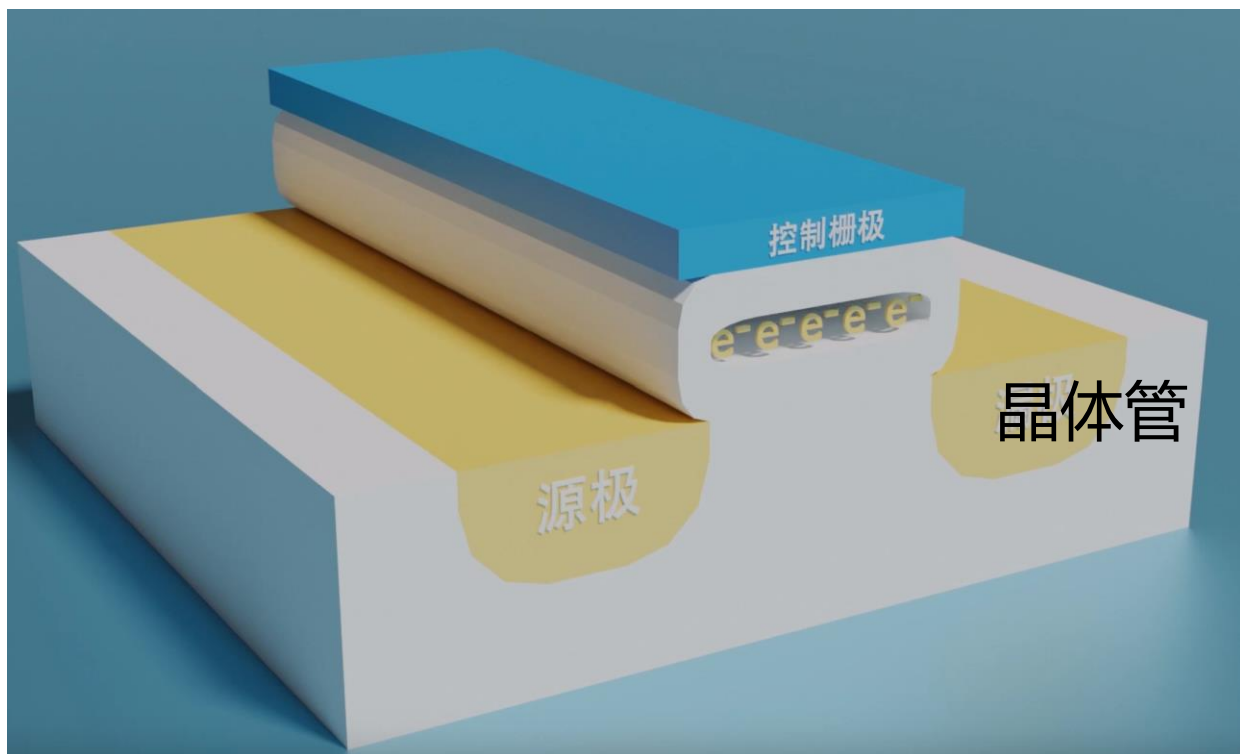
1, 1
1, 0

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）

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

二进制：满2进1



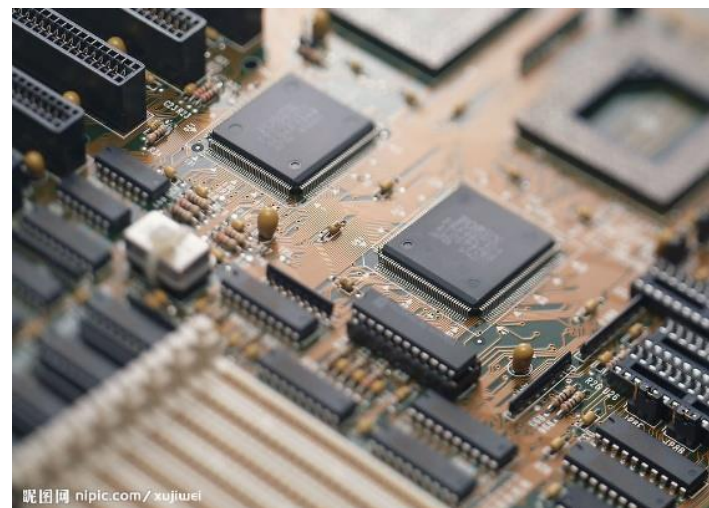
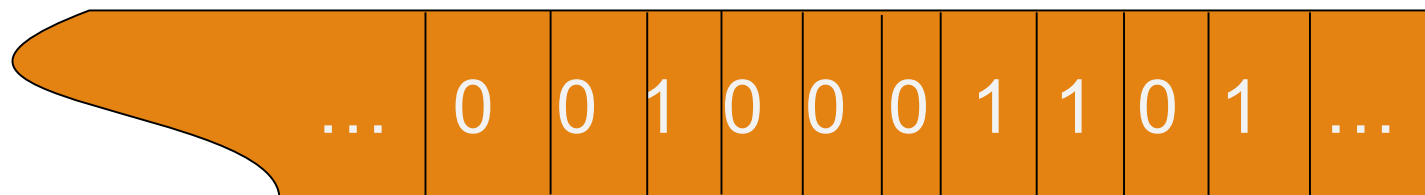
7, 8, 9, 10, 11, 12, 13...

十进制：满10进1



3.1 数值在计算机中的表示（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。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。

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

3.1 数值在计算机中的表示



内存地址

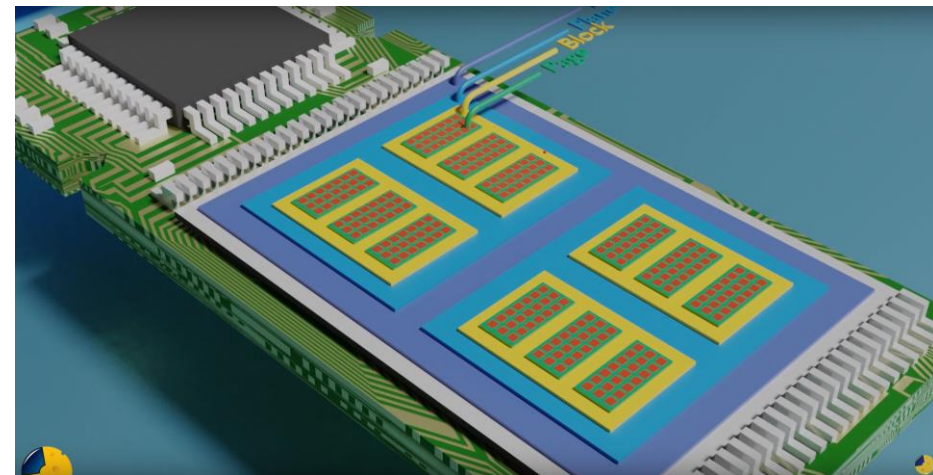
内存里存放的数

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
- ...

二进制编码

- 人类习惯用十进制表示数值
- 计算机只能对位进行操作和理解，即二进制
- 因此需要建立用二进制表达十进制的方法

【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 $(19)_{10} = (10011)_2$

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

记不住顺序?

除以2取余，逆序排列

"十进制"整数转"十进制"数

10	123 ₁₂₀	余数 3	低位
10	12 ₁₀	2	
10	1 ₀	1	
	0		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

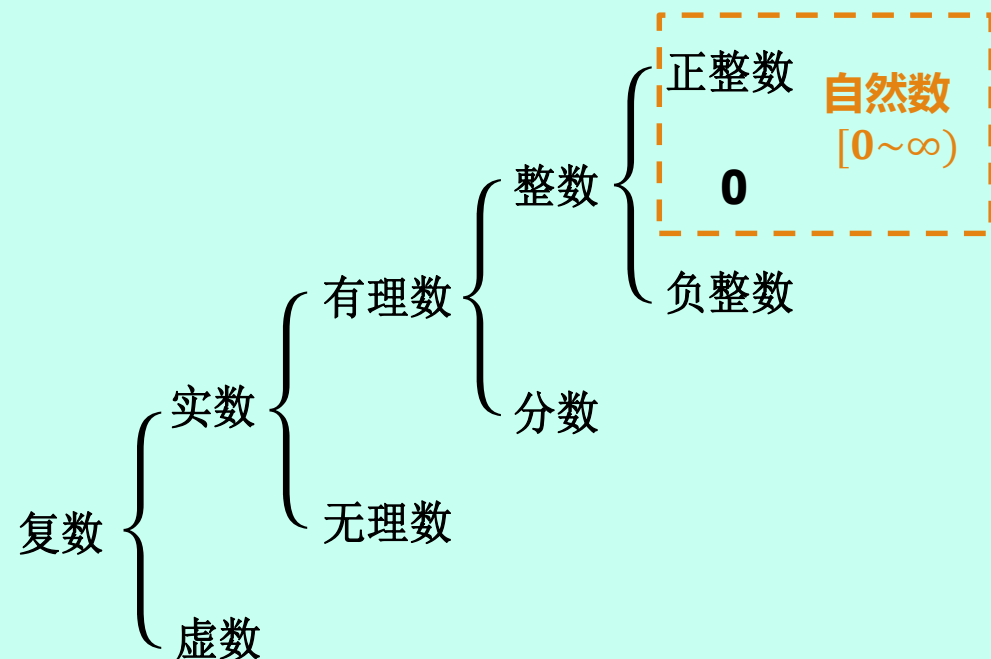
第0位

(1 0 0 1 1)₂

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

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

十进制: $1*10^2 + 2*10^1 + 3*10^0 = 123$



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?

原码，反码，补码

$7_{(10)}$ 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：
 $-127 \sim 127 \iff (-2^{8-1} + 1 \sim 2^{8-1} - 1)$

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。

- 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

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
---	---	---	---	---	---	---	---

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!

原码，反码，补码

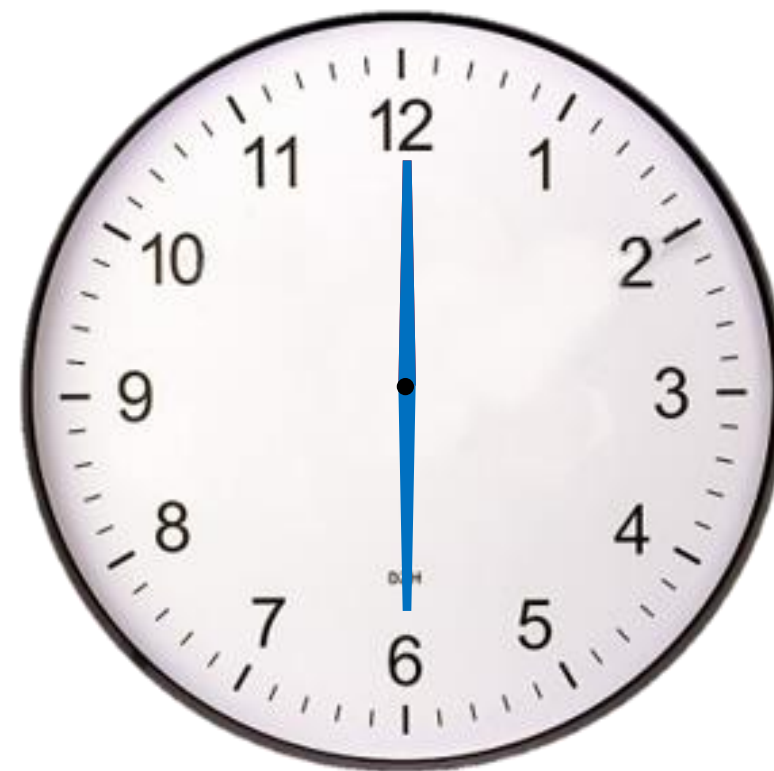


从“补数”说起

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。所以-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，共可以表示16个状态，范围从 0000~1111

正数补数即为本身，

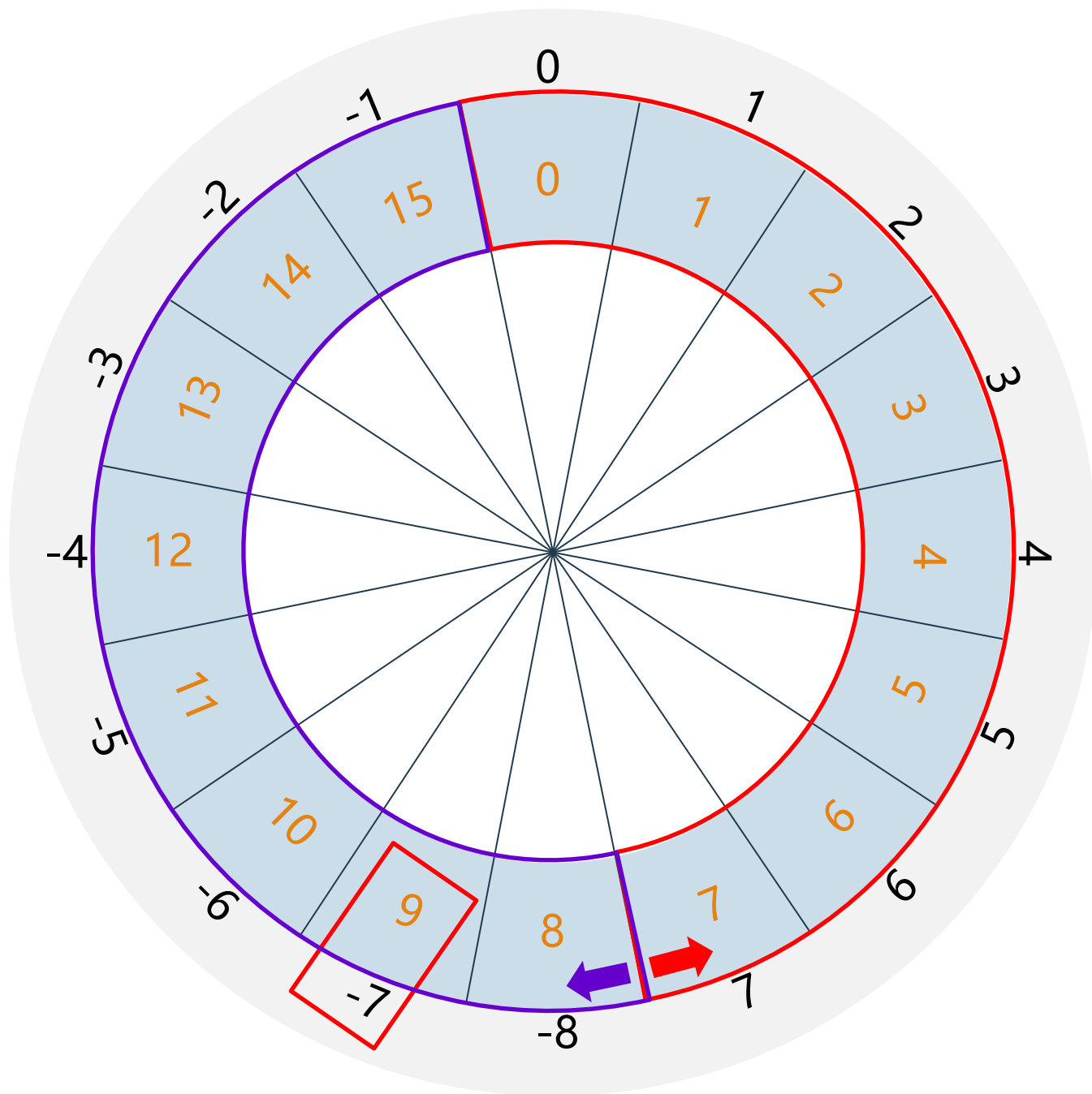
负数A的补数 = 模 - A的绝对值，

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

正数补数即为本身，
负数A的补数 = 模 - A的绝对值

-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”。

以 -1 为例

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

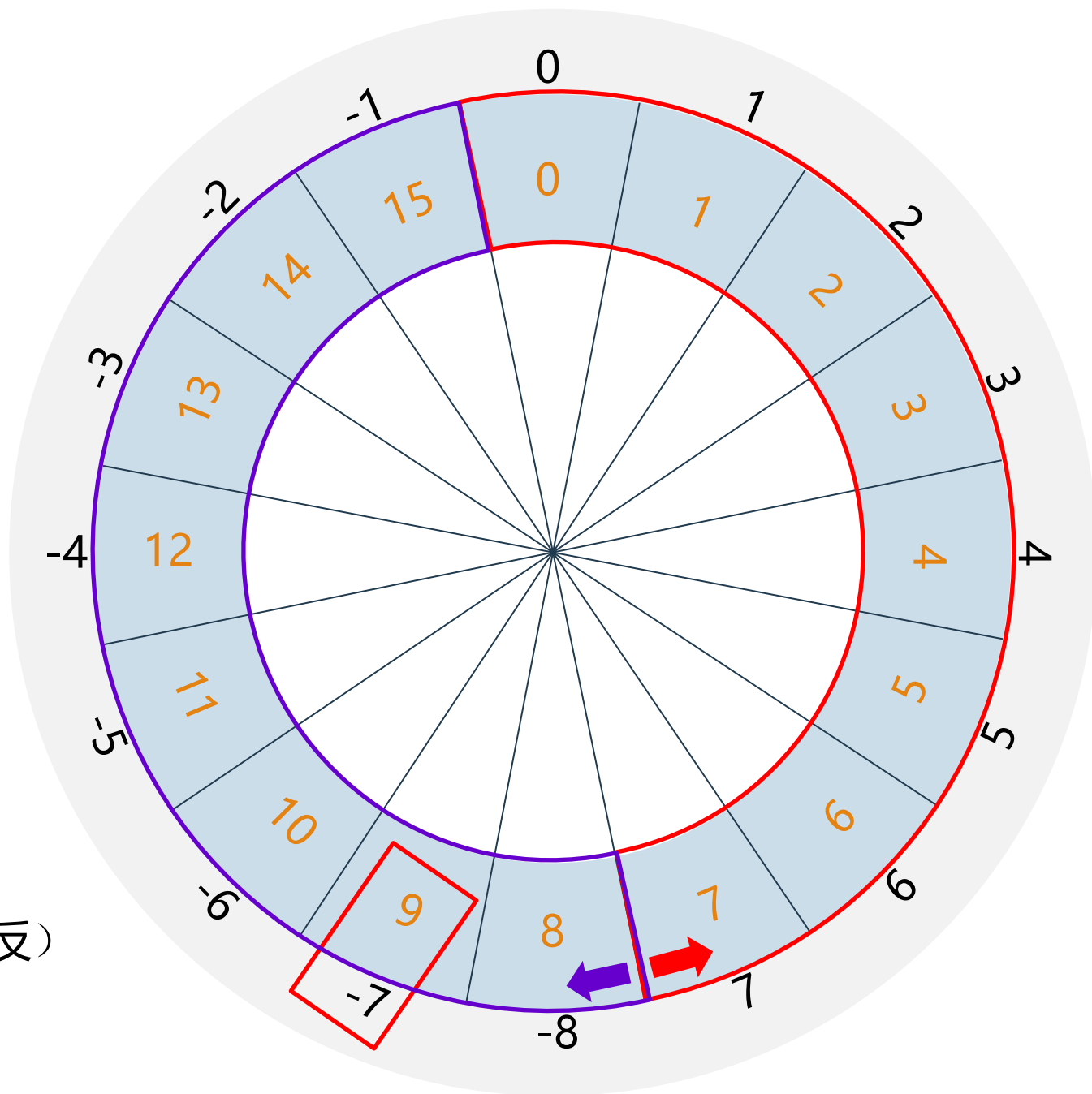
1

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

14 (15-1, 对 1 各位取反)

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

-1 (14加1=15)



原码，反码，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，也就是对其绝对值的原码取反，然后对整个数加 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
---	---	---	---	---	---	---	---

原码，反码，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，也就是对其绝对值的原码取反，然后对整个数加 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
1	1	1	1	1	0	0	0

原码，反码，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，也就是对其绝对值的原码取反，然后对整个数加 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
---	---	---	---	---	---	---	---

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

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

原码，反码，补码

signed char 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
---	---	---	---	---	---	---	---

有符号的视角

这是 -56 的补码

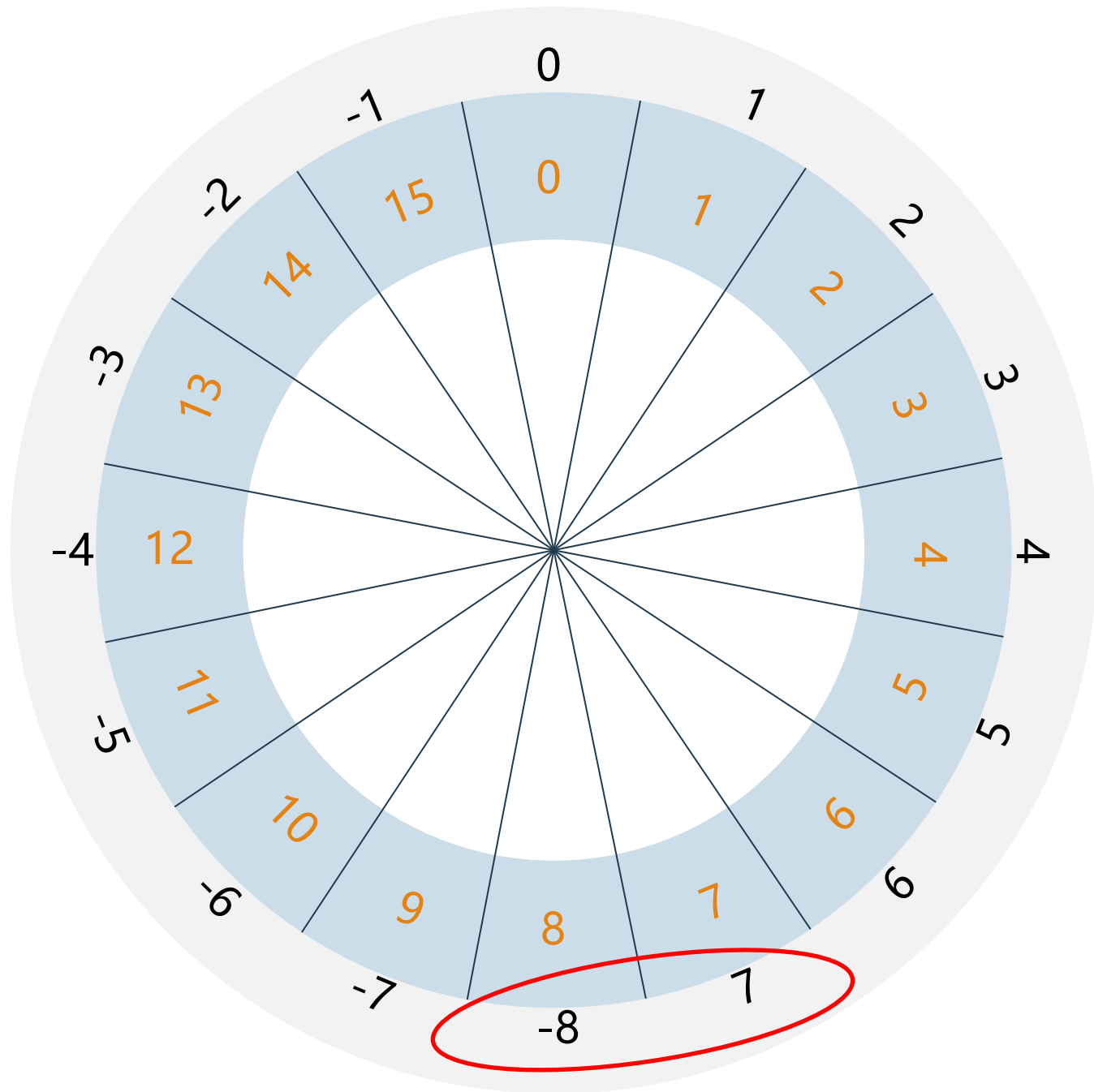
相等?



无符号的视角（或者说，其余高位都是0，省略显示）

100 100
100 + 100 = -56

模为256的意义下（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

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

✓ 0 的表示方式唯一

-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: $256-128=128$

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

✓ 表示范围: -128~127

原码, 反码, 补码

补码: 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!

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3-2：十进制转为八进制与十六进制，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 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

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

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"

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

023

3位构成
一组，
高位不
够补0

"二进制"转"十六进制"

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

0x2F

4位构成一组，高
位不够补0

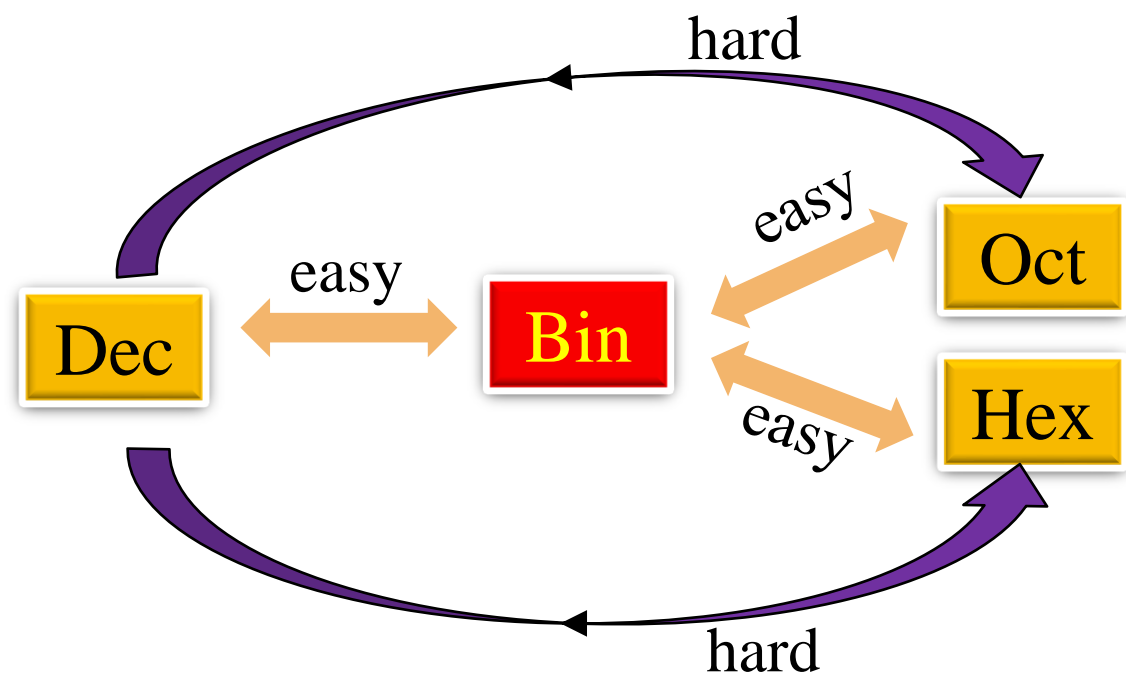
十进制 ↔ 八进制

A. “十进制”转“八进制”

8	19	余数
8	2	3
	0	2
		023

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

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

$$5 \& 3 = 1$$

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

- 运算量只能是 **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: $3 \& 5 = 1$

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

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	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 】

【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

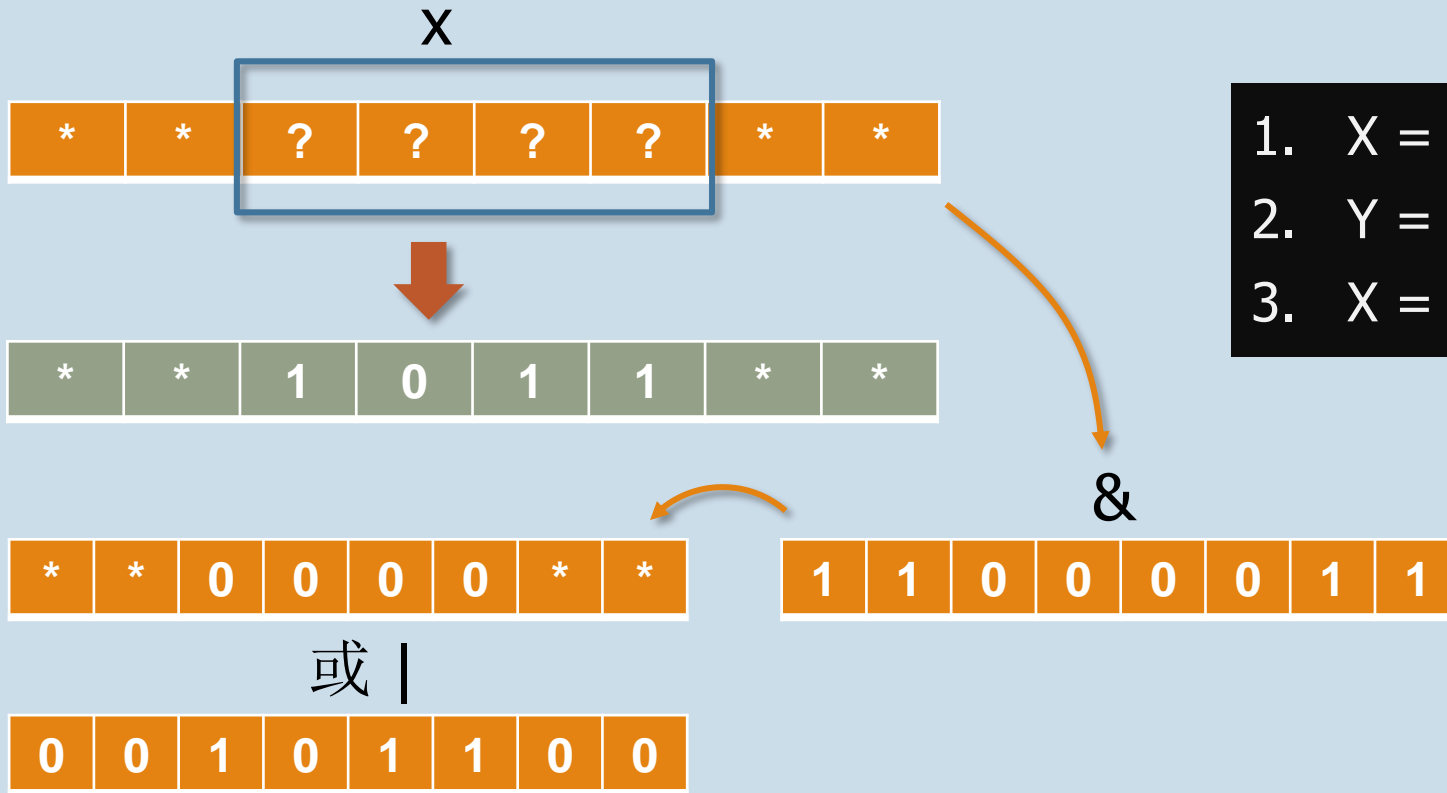
把 x 的特定位置为1: $x = x \mid ?$

	?	?	?	?	?	?	?	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

I 例3-5：按位或和按位与的综合范例，把x的2至5位设置为特定数

移花接木： $x = ?$



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

^ 按位异或

运算规则

A	B	A^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: $3^5=?$

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

应用：可用于实现“翻转”操作



按位异或

把 x 的特定位翻转: $x \wedge *$

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

x 寄存器值

\wedge

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

43 操作数

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

翻转 翻转 翻转 翻转

0不变 1翻转



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

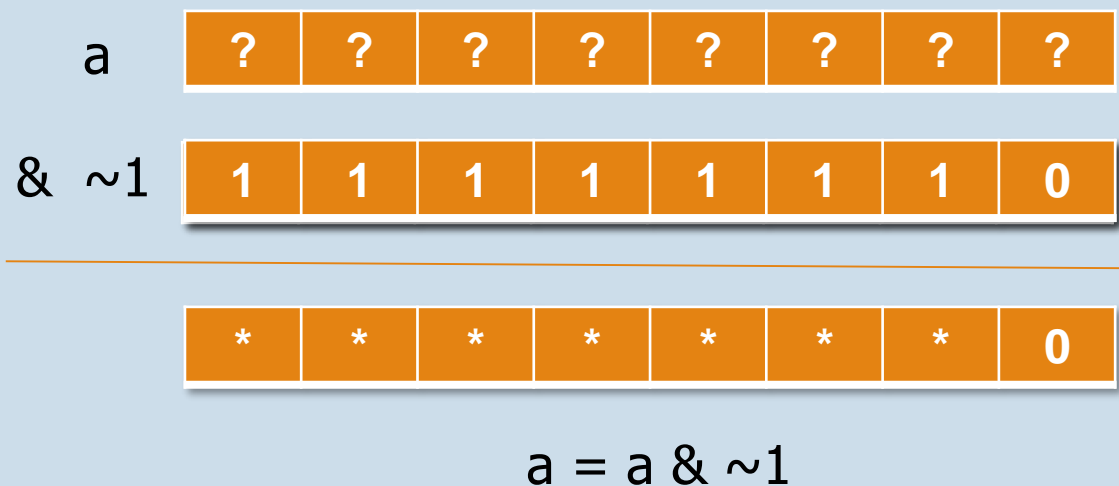
一元运算符，对二进制按位取反，
即将 0 变为 1，1 变为 0

例： $\sim 3 = ?$

3	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---

~ 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



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

例： `while(scanf(...) != EOF){...}`

`while(~scanf(...)){...}` // 题解里会有这种写法



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
---	---	---	---	---	---	---	---

$a = a \ll 2$?

--	--	--	--	--	--	--	--



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， $a = a \ll 2$

$a = 15$



$a = a \ll 2$



||

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

- 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



右移

将一个数的各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， $a = a >> 2$

$a = 15$

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

$a = a >> 2$?

--	--	--	--	--	--	--	--



右移

将一个数的各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

位运算符与赋值运算符的结合使用

$\&=$, $|=$, $>>=$, $<<=$, $\wedge=$

例: $a \&= b \iff a = a \& b$

例: 给一个整数 a 的 $\text{bit}7 \sim \text{bit}17$ 赋值937, 同时给 $\text{bit}21 \sim \text{bit}25$ 赋值17



位运算符与赋值运算符的结合使用

例3-8：给一个整数 a 的 bit7 ~ bit17 位赋值 937，给 bit21 ~ bit25 位赋值 17

```
unsigned int a = 0xc305bad3;  
a &= ~( ((1<<11) - 1) << 7 );  
a |= 937<<7;  
a &= ~( ((1<<5) - 1) <<21);  
a |= 17<<21;  
printf("a = 0x%x.\n", a);
```

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 $0.011\ 1111\ 1111\ 1000\ 0000$ ，
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. $a \mid= (937 << 7)$ ，把a的bit7 ~ bit17置为937
6. bit21 ~ bit25赋值为17，原理同上

计算器

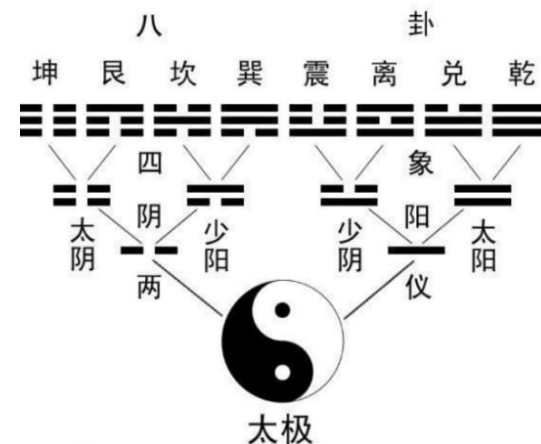
程序员

HEX	7FF
DEC	2,047
OCT	3 777
BIN	0111 1111 1111

位运算符与赋值运算符的结合使用

例3-8：给一个整数 a 的 bit7 ~ bit17 位赋值 937，给 bit21 ~ bit25 位赋值 17

```
unsigned int a = 0xc305bad3;  
a &= ~( ((1<<11) - 1) << 7 );  
a |= 937<<7;  
a &= ~( ((1<<5) - 1) <<21);  
a |= 17<<21;  
printf("a = 0x%x.\n", a);
```



编程与哲理

一生万物

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: $(x + y) >> 1$

【例】计算2的n次方 $1 << n$

【例】从低位到高位, 将n的第m位置1
 $n | (1 << (m-1))$

【例】从低位到高位, 将n的第m位置0

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

【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>

3.4 | 浮点数及数据范围

怪象：“0.3 等于 0.3” 不成立？

```
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));
```

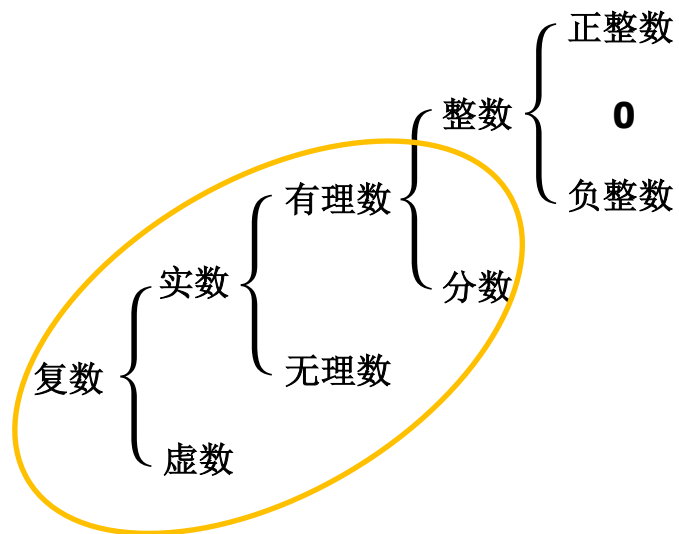
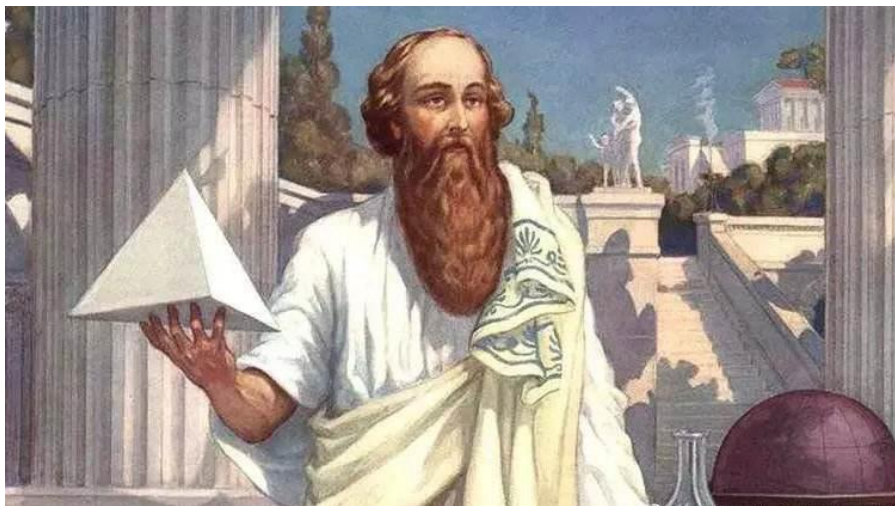


1, 1
1, 0

- ◆ 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 数学表达

注意：数学表达不等于
计算机表达！

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

"十进制"**整数**转"二进制"数

除以2取余
逆序排列

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

(19)₁₀ = (10011)₂

小数部分"十进制数"转"二进制数"

乘以2取整
顺序排列

	整数部分	高位
0.625 x 2 = 1.25.....	1	↓ 低位
0.25 x 2 = 0.5	0	
0.5 x 2 = 1	1	
(0.625) ₁₀ = (0.101) ₂		

小数的二进制数学表达

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

注意：数学表达不等于计算机表达！

```
#include <stdio.h>
int main()
{
    double a = 0.625, b = 0.3;
    printf ("%d, %d", (a == 0.625), (b == 0.3));
    return 0;
}
```

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

$$(0.3)_{10} = (0.010011 \dots)_2$$

浮点数表达不精确，用 == 判断浮点数相等时一定要小心！

小数的二进制数学表达

```
double 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");
}
```

codeblocks
下编译运行

b != 0.3
不点火

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？

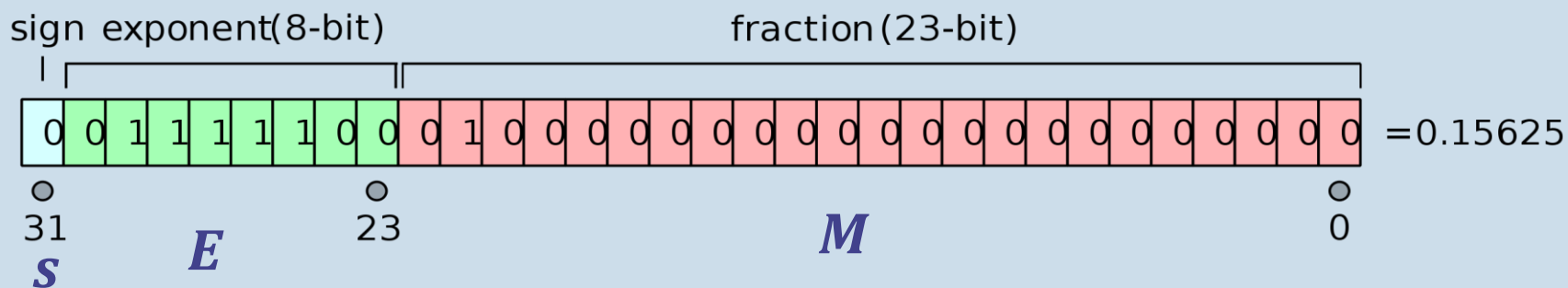
一行代码引发的惨剧

应该点火，却不点火



**浮点型数据的存储方式与数值范围

使用标准数据格式 **IEEE-754** 的进行存储和表示。数值以规范化的二进制数指数形式存放在内存单元中，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)分别存放。以**32位单精度浮点数**为例：



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

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，为了保证范围的前提下，尽可能保证精度，**在精度和范围之间做Trade Off**。

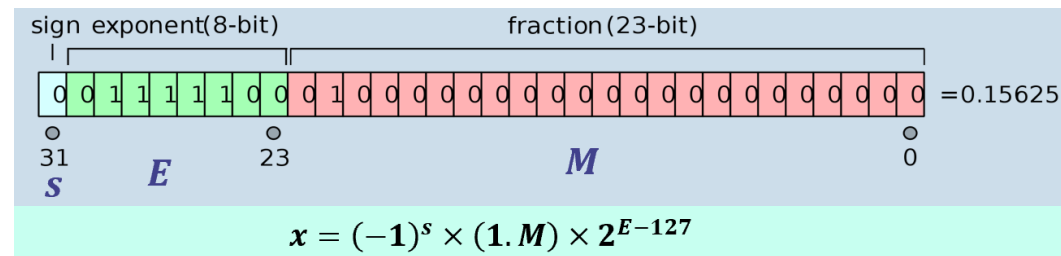
所以，实数编码问题变成了：如何编码才能使得照顾范围的同时让精度尽可能高？

指数决定范围，小数决定精度！

**浮点型数据的存储方式与数值范围

各浮点数据类型存放方式

M 是二进制, E 是十进制!



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

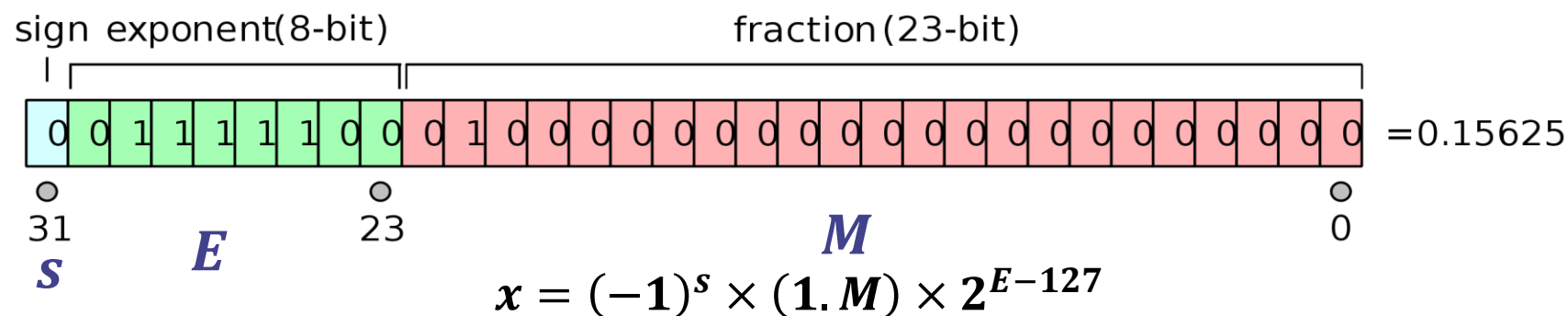
$$x = (-1)^S \times (1.M) \times 2^{E-1023} \quad (\text{double})$$

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

指数决定范围, 尾数决定精度!

**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

(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:

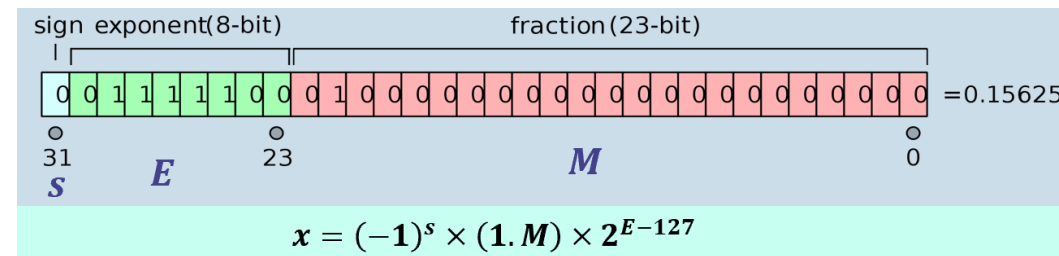
$$-1.111 \times 2^1 = (-1)^1 \times (1 + 0.1110\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^1$$

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

**浮点型数据的精度



相对精度: 机器 ϵ (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 ϵ 不同。以 **double 双精度** 为例，数值 1 是：

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

而比1大的最小双精度浮点数是:

[illegible]

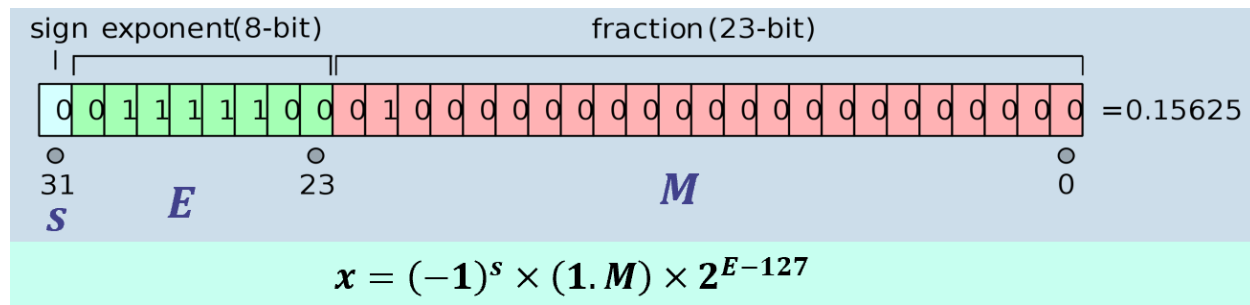
此二者之差为机器 ϵ : $2^{-52} \approx 2.220446049250313e-16$.

**浮点型数据的精度

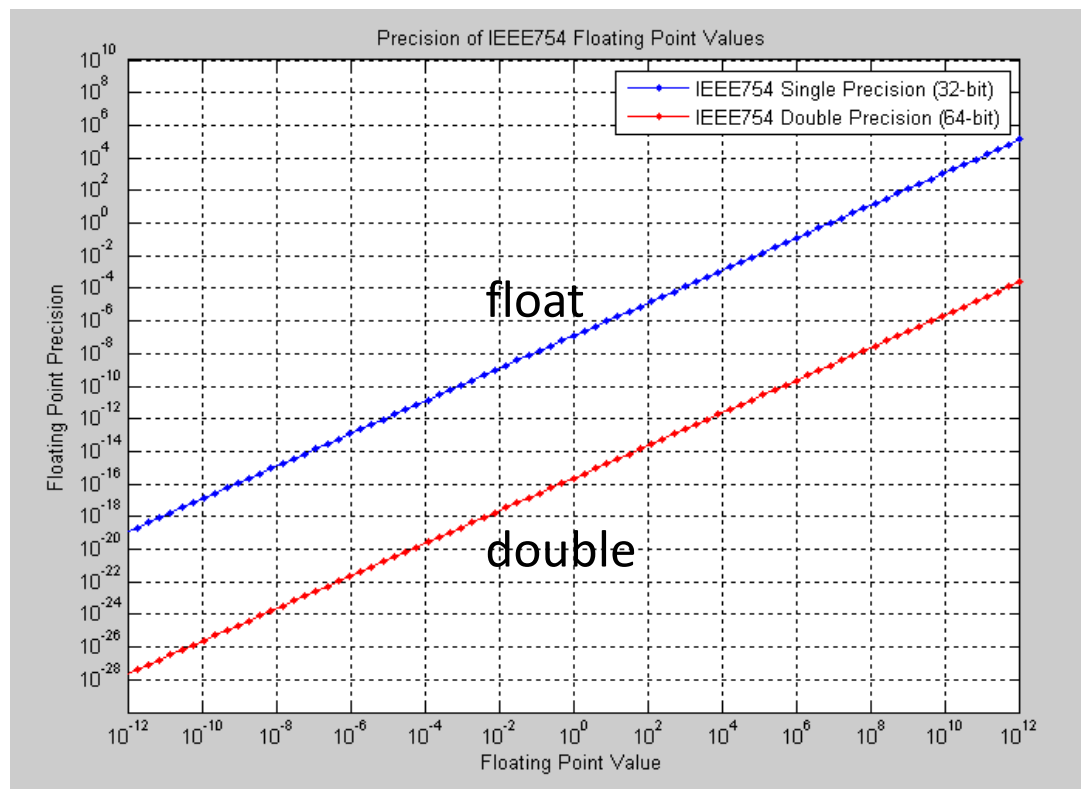
绝对精度:

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）。



float和double类型数据的绝对精度



【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 $disc == 0$ 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

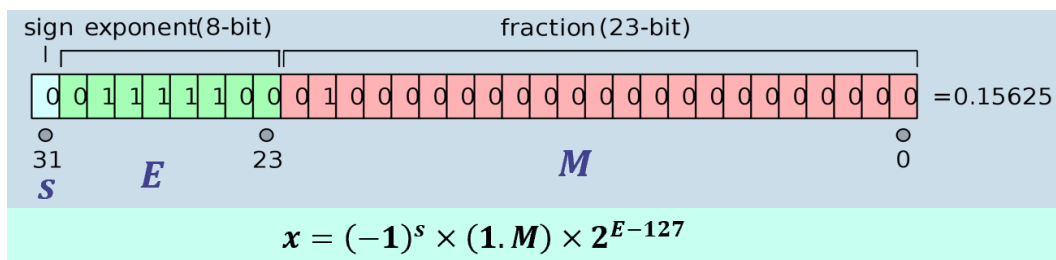
$(disc == 0)$ 可改为 **$fabs(disc) < eps$**

```
// file: c3-9.c
#include <stdio.h>
#include <math.h>
int main(){
    float a,b,c,d,disc,x1,x2,realpart,imapart;
    scanf("%f%f%f",&a, &b, &c);
    if(a == 0)    printf("Not a quadratic") ;
    else{
        disc=b*b-4*a*c;
        if(disc == 0)
            printf("Two equal roots: %8.4f\n",-b/(2*a));
        else if( disc>0){
            x1=(-b+sqrt(disc))/(2*a); // a很小时，溢出
            x2=(-b-sqrt(disc))/(2*a);
            打印实根(略);
        }
        else{ 计算、打印虚根;
        }
    }
}

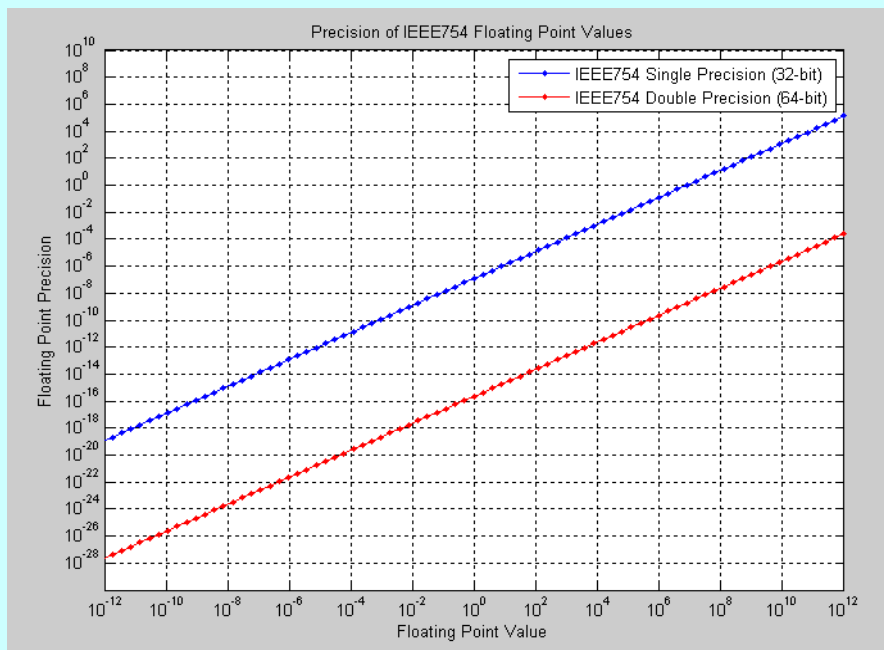
/* 注意：本代码直接拷贝并不能成功编译，还需要修改 */
```


浮点数小结

1. 在C语言中，浮点数有范围，有精度限制。
2. 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 $\pm 3.4 \times 10^{38}$ ，小端 $\pm 1.1 \times 10^{-38}$ ？能表示的绝对值最小数约为 $2^{-23} \times 2^{-127} \approx 10^{-44.85}$ ？
 $\because 2^{10} \approx 10^3 \quad \therefore 128/3.3 \approx 38, 2^{128} \approx 10^{38}$
3. double能表示的范围和精度更大。
4. 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0.99999999...，也可能是1.00000001...。
5. 使用浮点数要特别注意范围和精度问题！



float和double类型数据的绝对精度



鱼和熊掌
不可兼得



扩大范围损失精度
照顾精度减少范围

整数与浮点数小结

精度100%
但范围小
如: int, $2^{31}-1$

鱼和熊掌
不可兼得

精度可能受损
但范围大
如: float, 3.4×10^{38}

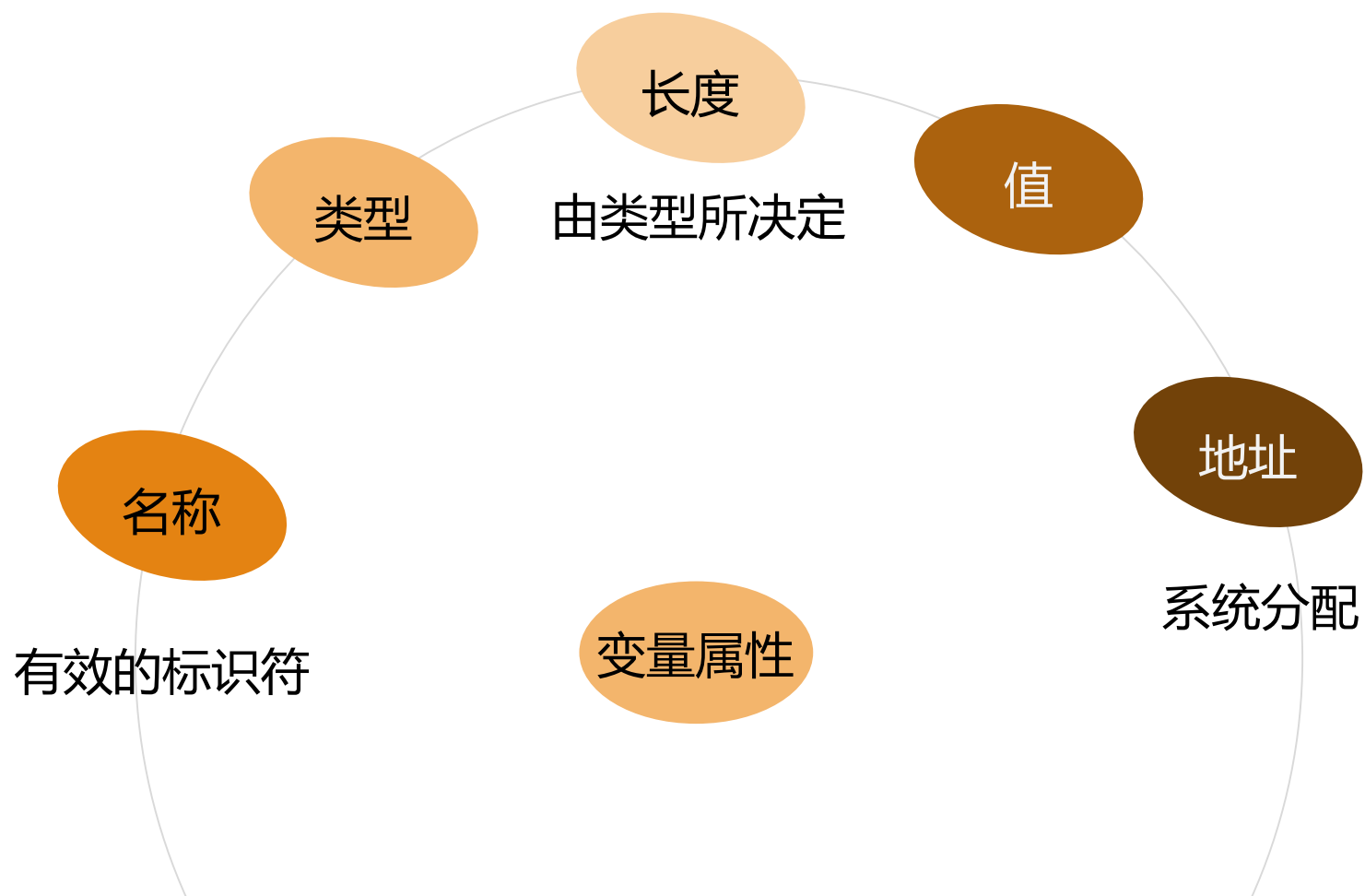
“整数家族”
char, int, short, long,
long long, unsigned ...



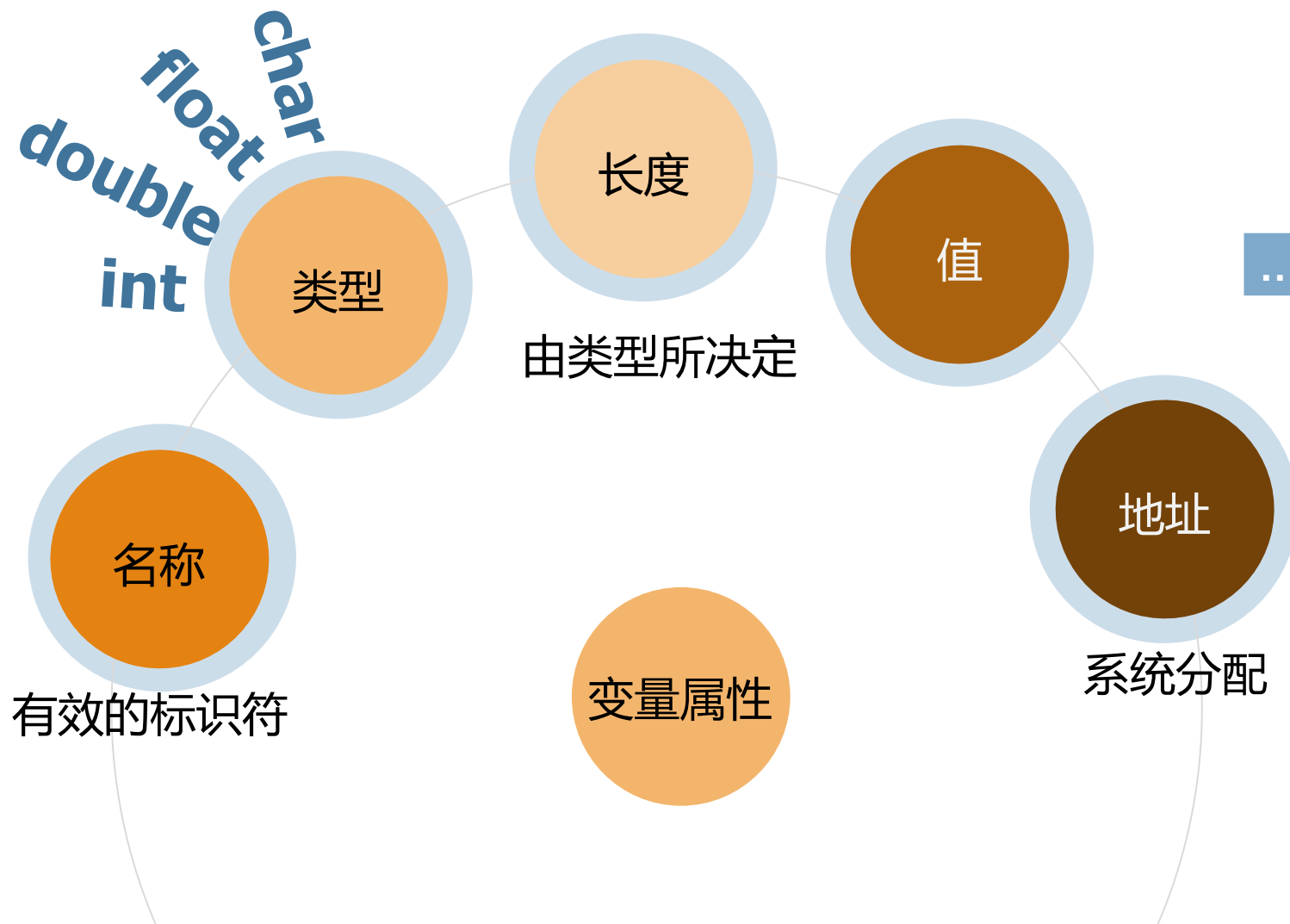
“浮点数家族”
float, double, ...

3.5 | 变量与内存的关系

常用的数据实体：简单变量和数组



常用的数据实体：简单变量和数组



char a;

a (60FF0F)



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

60FF0F
60FF0C
60FF08
60FF04
60FEF8

内存 (Memory)

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

基本数据类型及其内存存储空间

类型	字节	位	有效数字	取值范围
char	■	8		-128 ~ 127
int	■■■■	32		-2147483648 ~ +2147483647
unsigned int	■■■■	32		0 ~ 4294967295
short int	■■	16		-32768 ~ 32767
long int	■■■■	32		-2147483648 ~ +2147483647
long long int	■■■■■■■■	64		$-2^{63} \sim +2^{63}-1$
float	■■■■	32	6~7	约 $-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$ (大端, 详见P68)
double	■■■■■■■■	64	15~16	约 $-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$ (大端)

```

#include <stdio.h>
int main()
{
    int a, b;
    signed char sum = 0;

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

    return 0;
}

```

100 100
100 + 100 = -56

100

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

100

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

-56

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

基本数据类型及其内存存储空间

类型	字节	位	有效数字	取值范围
char	■	8		-128 ~ 127
int	■■■■	32		-2147483648 ~ +2147483647
unsigned int	■■■■	32		0 ~ 4294967295
short int	■■	16		-32768 ~ 32767
long int	■■■■	32		-2147483648 ~ +2147483647
long long int	■■■■■■■■	64		$-2^{63} \sim +2^{63}-1$
float	■■■■	32	6~7	约 $-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$ (大端, 详见P68)
double	■■■■■■■■	64	15~16	约 $-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$

- 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）表示。

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。

```
// 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				60FF24	...25
...26	...27	60FF28							

数组的类型与大小

跟变量一样，可以定义不同类型的数组

```
int a[LENGTH];
double b[LENGTH];
char c[LENGTH];
```

// sizeof(a) is sizeof(int)*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。

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				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]))
```

例如

```
#define ArrayNum(x) (sizeof(x)/sizeof(x[0]))
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++ )
    {        f[i] = i*i;        printf("%f\n", f[i]);        }
    ...
}
```

```
#define FOR(i,s,N) for(i=s; i<N; i++)
...
FOR(i,0,ArrayNum(d))
// 若FOR定义为如上宏，则for代码可以替换为这条
```

...

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，但初学者慎用，用得太多，程序的可读性可能也不好！

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（但也不宜上百MB），局部数组比较小（通常几十KB）

```
#include <stdio.h>

int voiceData[1<<20]; //函数之外，全局数组

int main()
{
    double stuScore[2000]; //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

- 实际问题中的数据可能很大，如电商数据几亿用户M，几千万商品N，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、算法设计、实际需要。
- 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）。
- 内存是宝贵的计算资源，应合理规划。

```
#define LSize 1000000
#define ssize 1000
double globalArray[LSize];

int main()
{
    int localArray[ssize];
    ...
}
```

【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=2097152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[1000 000]的数组是肯定会overflow的。我尝试在栈区开一个2000 000/4=500 000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[];
```



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。

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。

一种常见的用法：先定义宏常量，以宏常量作为数组长度

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

【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 中明确说明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。

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！）



请按下暂停键 **||**，思考**1分钟**，再继续播放，看后面的程序



新的知识点

getchar()

islower()

```
#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;
}
```

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]++; ...

课后练习

给出标准输入字符序列，统计有多少个单词？

数组的复制与比较

数组元素`a[i]`可以当作普通变量进行相应操作，但数组名`a`不可以（数组名实际上是地址）。不允许对数组进行整体操作。

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

```
...
```

```
b = a;
```

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

```
...
```

复制数组

比较数组



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])
```

```
...
```

数组的复制与比较**

标准库函数可以实现数组整体复制：

函数原型:

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

用法：

```
memcpy(b, a, sizeof(a));
```

[illegible][illegible]

数组的直接整体处理**

数组复制

- 方法一：通过循环逐一复制数组中元素
- 方法二：通过内置函数memcpy()实现整体复制

```
int a[5] = {1, 2, 3, 4};  
int b[5], i;  
b = a; ✗ // 错误  
b[5] = {1, 2, 3, 4}; ✗ // 错误  
for (i=0; i<5; i++)  
    b[i] = a[i]; // 正确  
memcpy(b, a, sizeof(a)); // 正确
```

```
char s[15] = "1234567890";  
memset(s, 'A', 6);  
printf("%s", s);
```

输出

AAAAAA7890

- b = a, 语法错误, 不能把数组整体赋值给另一个数组。
- b[5] = {1, 2, 3, 4}, 语法错误, 数组除定义时初始化外, 不能用 {数值列表} 进行整体赋值。
- 两个数组赋值需要通过循环逐一赋值数组元素。

`void *memcpy(void *dest, void *src, size_t count);`
将src中的count个字节拷贝到dest, 内存拷贝, 效率高!

`void *memset(void *s, int ch, size_t n);`
将s中当前位置后面的n个字节用 ch 替换并返回 s, 常用于清零等。

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1000个以内的整数，求和、求平均）

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



大量数据反复测试
多次重新输入

.....

96
85
73
91
82
68




```
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

- 作用：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调试数据（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。
- 对C程序内部处理逻辑无任何影响，可避免重复键入调试数据。
- 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）。

标准IO的重新定向

(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 `>`

```
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：调试完成，程序正确后，记得把该语句注释或删除（不然OJ上通不过）！

标准IO重定向实例



随机产生数据，输出到文件中，
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);
```

```
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]) == 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];
```

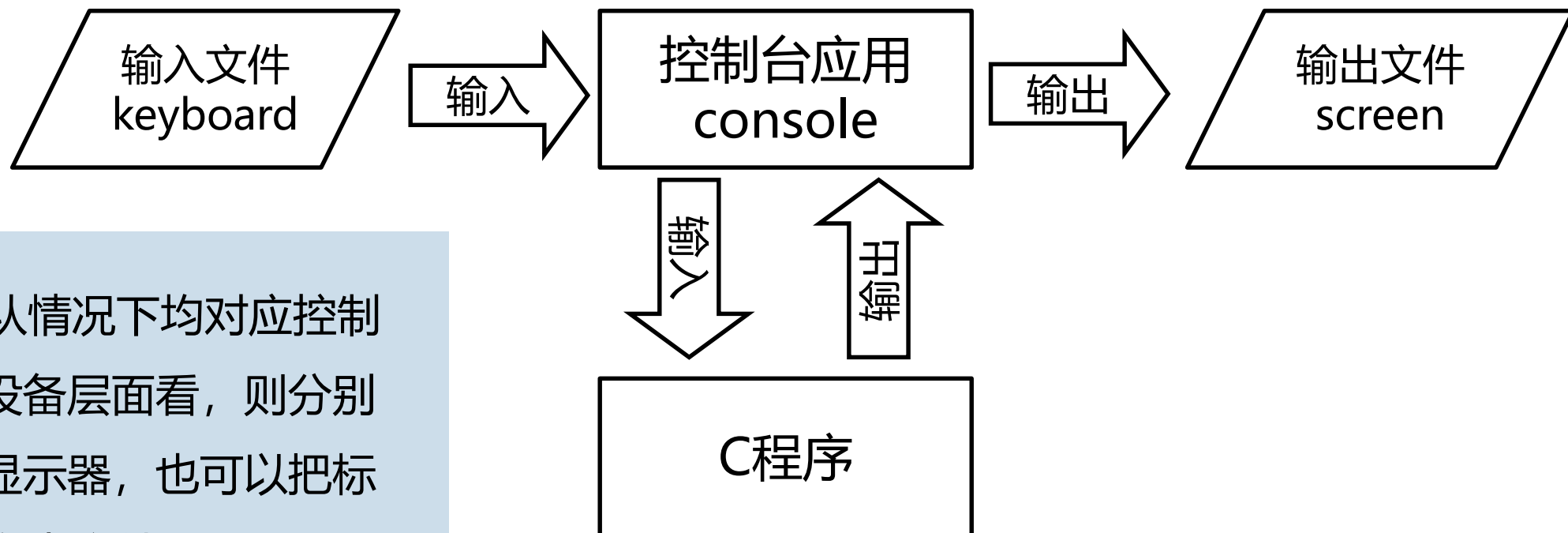
```
printf("    num: %d\n", n);
```

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

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

```
fclose(stdin);
```

标准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）。



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。