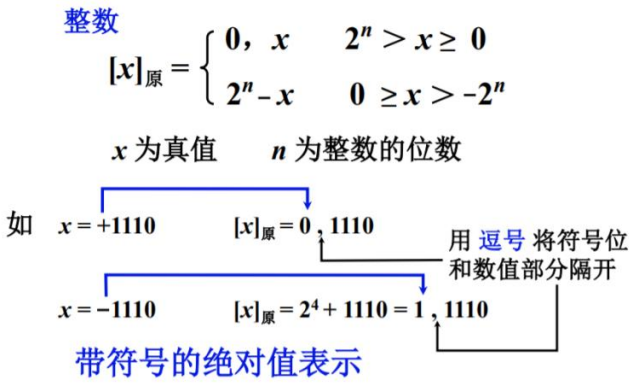
第二章速记清单：

Part1：整数

1. 有符号数的表示（原码、补码和移码）：

（1）整数原码的表示（补充知识）



（2）原码的特点（补充知识）

但是用原码作加法时，会出现如下问题：

要求	数1	数2	实际操作	结果符号
加法	正	正	加	正
加法	正	负	减	可正可负
加法	负	正	减	可正可负
加法	负	负	加	负

能否 只作加法？

找到一个与负数等价的正数 来代替这个负数

就可使 减 ⟶ 加

（3）引入补码的意义：可以将减法转换为加法，这样计算机系统内的加减计算可以统一到加法。

（4）求补码的快捷方式：

① 当这个数为正数时，补码和原码一样。

② 当这个数为负数时，补码可用原码除符号位外每位取反，末位加一得到，或者用扫描法。例如：原码 1,0010101➔补码：1,1101011

（5）用移码表示浮点数阶码的意义：由于移码的二进制表示的值是随着真值递增的，所以能够方便地判断浮点数阶码的大小（只需从高位开始按位比较每一位），从而使得浮点数的比较可以通过整数的比较来实现。

（6）真值、补码、移码对照表：

真值 x (n=5)	$[x]_{补}$	$[x]_{移}$	$[x]_{移}$对应的十进制整数
-100000	100000	000000	0
- 11111	100001	000001	1
- 11110	100010	000010	2
⋮	⋮	⋮	⋮
- 00001	111111	011111	31
± 00000	000000	100000	32
+ 00001	000001	100001	33
+ 00010	000010	100010	34
⋮	⋮	⋮	⋮
+ 11110	011110	111110	62
+ 11111	011111	111111	63

注意：补码的二进制表示的值是不随着真值的递增而递增的，但是移码是的，所以补码用于计算，移码用于比较。

2. 有符号数与无符号数的表示范围（以 32 位的数为例）：

无符号数：Umax =0，Umax = 2^{32}-1

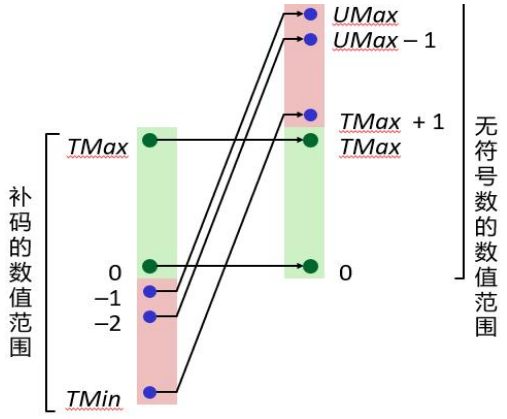
有符号数（补码表示）：Tmax=-2^{31}，Tmax= 2^{31}-1

3. 有符号数和无符号数转换的基本原则：

（1）位模式不变

（2）重新解读（按目标编码的规则解读）

（3）负的有符号转为无符号数会加 2^w，大于 Tmax 的无符号数转为有符号数会减 2^w



注意：当表达式含有无符号数和有符号数时，有符号数会隐式转换为无符号数。

4. 整数的加法、乘法、移位，以及溢出

（1）w 位无符号数加法溢出判断

运算结果超过 w 位则溢出，对结果取 mod2^w，即只保留低 w 位。

（2）w 位有符号数加法和溢出判断

有符号数加法和无符号数加法有完全相同的位级表现。

溢出判断：两个正数相加结果为负数则溢出，或者两个负数相加结果为正数则溢出，其他情况皆没有发生溢出。

$$TAdd(x,y)=\begin{cases} x+y-2^{w}, & TMax_{w}<x+y & \text{正溢出} \\ x+y, & TMin_{w}\le x+y\le TMax_{w} & \text{正常} \\ x+y+2^{w}, & x+y<TMin_{w} & \text{负溢出} \end{cases}$$

（3）w 位无符号数乘法

结果最多可能为 2w 位，对结果取 mod 2^w，即只保留低 w 位。

（4）w 位有符号数乘法

结果最多可能为 2w 位，对结果取 mod 2^w，即只保留低 w 位。

5. 关于整数除以 2 的幂（k）以及舍入的问题

（1）当被除数为正数时，是向零舍入（向下舍入）的

（2）当被除数为负数时，会向下舍入，为了达到向 0 舍入的要求，负的被除数会先加上一个偏置 2^k-1 再做除法。

Part2：浮点数

1. IEEE754 规格化数和非规格化数，无穷的表示以及 NaN

单精度浮点数值的分类

1. 规格化的

s	≠0 && ≠255	f

2. 非规格化的

s	0000 0000	f

3a. 无穷大

s	1111 1111	000 0000 0000 0000 0000 0000

3b. NaN(Not a Number)

s	1111 1111	≠0

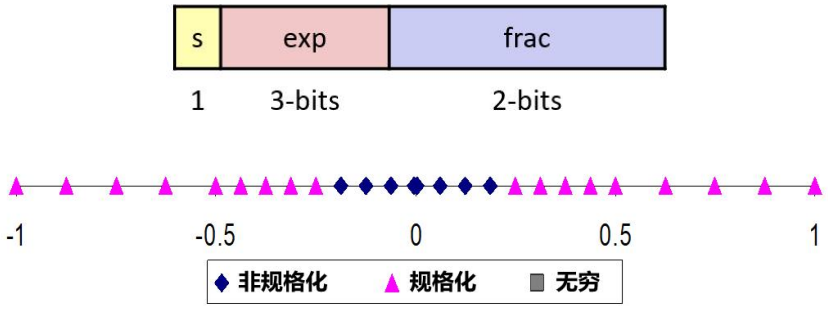
2. 关于 C 中出现和 NaN 或 inf 比较的测试：

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(){
    if(NAN){
        printf("yes1\n");
    }
    if(NAN>1.0||NAN<=1.0){
        printf("yes2\n");
    }
    if(INT_MAX<INFINITY){
        printf("yes3\n");
    }
    if(INT_MIN>-INFINITY){
        printf("yes4");
    }
    return 0;
}
```

注意：NAN 与任何数比较都是 false

```
[Running]
yes1
yes3
yes4
[Done] exi
```

3. 浮点数在数轴上的分布以及当阶码为零时，为什么规定 E=1-Bias



可以看到由于 frac 是 2 位，因此每个区间是被 4 个点等分的，且可以发现每个区间都是前面所有区间（到 0 为止）长度之和，因此越往后点之间的间隔是越大的。

同时可以发现非规格化数和第一个区间的规格化数相邻点之间的距离是相同的，这得益于当阶码为零时，规定 E=1-Bias。

4. 一定要熟练地进行实数与浮点数之间的转换！

5. 本章最后 PPT 的思考题答案（供参考）

（1）IEEE754 比 32 位传统浮点数（整数符号（阶符）1 位，整数数值部分（阶码）10 位，小数符号（数符）1 位，小数数值部分（尾数）20 位）的表示方法有什么优、缺点?

答：相比之下，IEEE754（1 位、8 位、23 位三个字段）浮点数表示数的范围更小，但 IEEE754 精度更高。根本原理在于，阶码位数决定范围，尾数位数影响精度。

（2）float 非无穷的最大值和最小值？最大值：$(2-2^{-23})*2^{127}$，最小值：$-(2-2^{-23})*2^{127}$

（3）float 数 1，65536，0.4，-1，0 的内存表示（注意：教材 29 页低地址从左开始）

1：0 01111111 00000000000000000000000 内存表示（小端）：00 00 80 3F

65536：0 10001111 00000000000000000000000 内存表示（小端）：00 00 80 47

0.4：0 01111101 10011001100110011001101 内存表示（小端）：CD CC CC 3E

-1：1 01111111 00000000000000000000000 内存表示（小端）：00 00 80 BF

0：1 00000000 00000000000000000000000 内存表示（小端）：00 00 00 80

或 0 00000000 00000000000000000000000 内存表示（小端）：00 00 00 00

（4）一个数的 Float 形式是唯一的吗（除了 0）？ 是的

（5）每一个 IEEE754 编码对应的数是唯一的吗？ 不是的，还有 0 和 NaN

（6）简述 Float 数据的浮点数密度分布： 上面第 3 点说过了

（7）c 语言中除以 0 一定报错溢出吗？

答：整数会报错，浮点数会得无穷大（测试程序在最后）

（8） int 与 float 都占 32 个二进制位，float 与 int 相比谁的数的个数多？各自是多少个？多多少？

答：int 表示的数多， int 可以表示 2^{32} 个数。

float 可以表示 $2^{32}-1-2^{24}$ 个数（减 1 是因为 0 有两种表示，需要减去一种，减 2^{24} 是因为 NaN 和 inf 一共有这么多种）。int 比 float 多 $1+2^{24}$ 个数。

单精度浮点数值的分类

1. 规格化的

s	≠0 && ≠255	f

2. 非规格化的

s	0000 0000	f

3a. 无穷大

s	1111 1111	000 0000 0000 0000 0000 0000

3b. NaN(Not a Number)

s	1111 1111	≠0

（9）怎么判断和定义浮点数的无穷大以及 NaN？

NaN：阶码的每个二进制位全为 1，并且尾数不为 0；

无穷：阶码的每个二进制位全为 1，并且尾数为 0；符号位为 0，是正无穷，符号位为 1 是负无穷。

编程实例：在 C 中，1.0/0 会得到正无穷，对负数开根号或取对数或 inf-inf 会得到 NaN。

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(){
    if(1.0/0==INFINITY){
        printf("yes\n");
    }
    if(isnan(sqrt(-1))){
        printf("yes1");
    }
    return 0;
}
```

```
[Running] cd "c:\Users\chenq\Desktop\pypc\" && gcc test.
test.c: In function 'main':
test.c:5:11: warning: division by zero [-Wdiv-by-zero]
    if(1.0/0==INFINITY){
          ^
yes
yes1
[Done] exited with code=0 in 0.457 seconds
```

Part3：int、float、double 之间的转换规则

C 语言中的 float 和 double 类型分别对应于 IEEE 754 单精度浮点数和双精度浮点数。long double 类型对应于扩展双精度浮点数，但 long double 的长度和格式随编译器和处理器类型的不同而有所不同。在 C 程序中等式的赋值和判断中会出现强制类型转换，以 char->int->long->double 和 float->double 最为常见，从前到后范围和精度都从小到大，转换过程没有损失。

- 从 int 转换为 float 时，虽然不会发生溢出，但 int 可以保留 32 位， float 保留 24 位，可能有数据舍入，若从 int 转换为 double 则不会出现。
- 从 int 或 float 转换为 double 时，因为 double 的有效位数更多，因此能保留精确值。
- 从 double 转换为 float 时，因为 float 表示范围更小，因此可能发生溢出。此外，由于有效位数变少，因此可能被舍入。
- 从 float 或 double 转换为 int 时，因为 int 没有小数部分，所以数据可能会向 0 方向被截断（仅保留整数部分），影响精度。另外，由于 int 的表示范围更小，因此可能发生溢出。

在不同数据类型之间转换时，往往隐藏着一些不易察觉的错误，编程时要非常小心。

Part4：类型转换经典案例

1. 已知：int x = …; float f = …; double d = …。假定 d 和 f 都不是 NaN，则以下式子一直成立的是：

x == (int)(float) x	**No: float 只有 24 位尾数**
x == (int)(double) x	**Yes: 53 位尾数**
f == (float)(double) f	**Yes: 增加精度**
d ==(double) (float) d	**No: 溢出或损失精度**
f == -(-f);	**Yes: 仅仅改变符号位**
2/3 == 2/3.0	**No: 2/3 == 0**
d < 0.0⇒ ((d*2) < 0.0)	**Yes!**
d > f ⇒ -d < -f	**Yes**
d *d >= 0.0	**Yes!**
x *x >= 0	**No! 例如 50000*50000**
(d+f)-d == f	**No: 不具备结合性，可能大数吃小数**

注意：浮点型计算都是按照实数来进行，不像整数计算有“正+正=负”或“负+负=正” 这两种溢出情况。**浮点计算溢出有两种情况**：

1）**向无穷大溢出**：当求得的值大于浮点数能表示的最大值，就变为+∞，负无穷同理；

2）**向 0 溢出**：当为正数时，当求得的值小于能表示的最小值时，舍入为 0，−0 同理。

Part5：浮点数加法、乘法经典案例：

1. 浮点数加法不满足结合律

(3.14+1e10)-1e10 = 0 （被称作“大数吃小数”）

3.14+(1e10-1e10) = 3.14

2. 浮点数乘法不满足结合律

(1e20*1e20)*1e-20 = inf

1e20*(1e20*1e-20) = 1e20

3. 浮点数不满足分配率

1e20*(1e20 - 1e20) = 0.0

1e20*1e20 - 1e20*1e20 = NaN

- 正无穷（inf）、负无穷（-inf）：1.0/0.0 = −1.0/−0.0 = +∞,

  1.0/−0.0 = −1.0/0.0 = −∞

- NaN 表示没有数值结果（实数或无穷），例如： sqrt(−1)， ∞−∞， ∞ × 0