信息与编码

kiterunner_t
TO THE HAPPY FEW.

现代计算机以二进制的形式存储和处理信息。13 世纪由意大利数学家 Leonardo Pisano，即 Fibonacci 带到西方。对于人来说，十个手指头的十进制是更好的选择，但对计算机来说，诸如低电压、高电压之类的二进制表示更方便。

位组合在一起，不同的位模式**解释**为不同的含义，我们就能够表示有限集合的元素，并在这些集合元素上定义一系列**运算规则**，这就是**编码**。**数字**和**字母**（符号）以不同的方式进行编码。通常有 3 种重要的数字表示，无符号整数、有符号整数、浮点数。由于表示整数和浮点数的有限性的方式不同——整数精确的编码了一个相对较小的数值范围，而浮点数只能近似的表示一个较大的数值范围，它们有不同的数学属性，如整数运算会溢出，但满足我们熟悉的整数运算定律（即使结果不是期望的，但至少会保持与运算性质的统一），浮点数溢出时，并不会导致负数，但不是可结合的。字符表示通过在一系列字符集上定义类似于整数的位集合来表示。

历史上，由于不同的编码范围和算术运算的属性，造成了很多安全漏洞，同时对于理解编译器产生的机器级代码也是很重要的。

现代计算机将存储器视为一个单一的大的数组，由硬件、编译器和运行时操作系统提供了被称为**虚拟存储器**（参见另一篇计算机存储）抽象概念。通常将虚拟存储器划分为 8 位的块，称为一个**字节**，每个字节都由一个唯一的数字来标识，即**地址（指针）**，访问存储器是按照字节（即地址）来进行的，而不是位。所有可能地址的集合称为**虚拟地址空间**。编译器和操作系统分配和管理虚拟存储空间，用以存放不同的程序对象（数据、指令）。机器级代码并不维护任何数据类型的信息，它所看见的只是一个大的字节数组，一个个程序字节块。**本文中的存储器都是指虚拟存储器。**

计算机的虚拟地址空间是通过指针（地址）来访问的，指针的大小由**字长**决定。因此字长决定了虚拟地址空间的大小。当前，常用的字长有 2 种，32 位和 64 位。64 位的地址空间并没有全部被使用。（参见另一篇计算机存储）

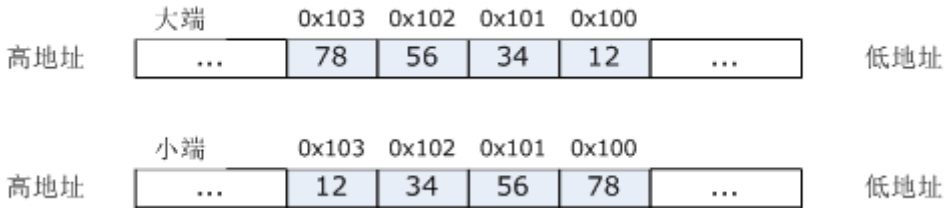
数值表示
二进制、八进制、十进制、十六进制，需要熟悉这几种数字表示方式之间的转换。

十进制数$x = 2^n$，且 n 为非负整数，令 n=i+4j，0≤i≤3，则 x 可以写成开头的十六进制数字为 1(i=0)、2(i=1)、4(i=2)、8(i=3)，后面跟着 j 个十六进制的 0。如 $x = 2048 = 2^{11}$，n=3+4·2，x 十六进制表示为 0x800。

十进制表示和十六进制表示之间的转换需要使用乘法或除法来处理一般情况。将一个十进制数字 x 转换为十六进制，可以反复的用 16 除 x，得到一个商 q 和余数 r，即 x=16*q+r。

对象的大小、地址、字节顺序

跨越多字节的对象，需要建立两个规则：对象的**地址**是什么，存储器中如何**排列**这些字节。在几乎所有机器上，多字节对象都被存储为连续的字节序列，其中较小字节的地址为对象的地址。对象排列时，则出现了大端和小端之争，大端将字节按照从最高有效字节到最低有效字节的方式进行排列，而小端则将最低有效字节放在前面。有些机器同时支持这两种方式。如十进制整数 0x12345678 在大端和小端机器上的表示如图（画存储器时，一般低地址在下，高地址在上，低地址在左，高地址在右），假设 x 位于地址 0x100 处



参考代码 byte_order.c

C 语言支持整数和浮点数的多种数据格式，其对象字节大小通常如表所示，根据长整数、指针大小的不同，有 ILP 和 LP 模型之分。由于 C 语言并没有精确规定每种数据类型的大小，而只是设定了下界，没有上界，因此程序员需要确保程序再不同的机器和编译器上的可移植性。

C 声明	C 标准中下界	32 位	64 位
char		1	1
unsigned char		1	1
short int		2	2
int		4	4
long int		4	8
long long int		8	8
void *		4	8
float		4	8
double		8	8
long double		12	12

C 语言中字符串被编码为一个以 null（值为 0）字符结尾的字符数组，每个字符以标准编码来表示，通常使用 ASCII。以 ASCII 码表示的字符串在任何系统上都将得到相同的结果，与字节顺序和字大小规则无关，比二进制数据有更强的平台独立性。ASCII 能表示的字符有限，现在多用 Unicode 标准编码。

在不同的机器上使用不同的且不兼容的指令和编码方式，因此二进制代码很少能在不同机器和操作系统组合之间移植。从机器的角度来看，程序仅仅只是字节序列，机器没有源代码的任何信息，除了可能用来帮助调试的辅助信息（**参考链接和加载**）。

代码也是数据，二进制兼容性 ABI，dwarf

位

布尔代数简介

二进制是计算机编码、存储和操作信息的核心，因此演化出了丰富的数学知识体系。它起源与 1850 年前后的 George Boole，后来 Claude Shannon 在 20 世纪 30 年代末建立了布尔代数和数字逻辑电路之间的联系。

布尔代数在二元集合{0, 1}上定义了非、与、或、异或四种运算，通过位向量扩展到各种对象。位向量是有固定长度 w、由 0 和 1 组成的串，可以表示为 $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$。长度为 w 的为向量上的运算形成一种被称为布尔环的数学形式。布尔环有一些有趣的属性，如其加法逆元为自身，即 0^0=1^1=0，则(a^b)^a=b，这可以用来不交换两个不同位置的元素。如代码 swap_bool_ring.c

位向量可以有效的来表示有限集合，位向量$\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$可以编码集合 $X \subseteq \{0, 1, \ldots, w-1\}$，其中$x_i = 1$当且仅当$i \in A$。实际应用中，常用位向量来对集合编码，如信号集 sigset_t。通常会用到**掩码**来取得某些位或屏蔽某些位。

C 语言中位级运算有 |，&，~，^，可以应用于任意整数数据类型上。

区分逻辑运算和位运算
逻辑运算功能与位运算不同，逻辑运算中，所有非零参数都表示真，参数 0 表示假。第二个区别是如果对第一个参数求值就能确定表达式结果，则不会对第二个参数求值，如 a && t/a 将不会造成被零除。

移位运算
左移，x<<K 表示位模式$[x_{w-1}, x_{w-2}, \ldots, x_0]$变为$[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$，即丢弃高位的 k 个位，低位补齐 k 位 0。右移分为逻辑右移和算术右移，逻辑右移高位填 0，而算术右移高位填符号位。算术右移对于无符号算术运算很有用。

移位运算是从左至右结合的，需要注意的是加减法的运算级别比移位的高，因此 1<<2+3<<4 等价于 1<<(2+3)<<4，这可能不是我们想要的。

C 语言标准并没有明确规定应该使用哪种右移，因此假设无符号都是算术右移都存在潜在的可移植性问题。但，实际上，几乎所有编译器/机器组合都是用算术右移。Java 中明确规定了使用算术右移。

C 语言标准中并没有明确规定移动 k 位，k≥w 时会发生什么，因此移位需要尽量保证移位量小于字长。大多数机器的决定是，实际移位量是通过 k mod w 来计算的。Java 则明确规定了采用取模的方法计算移位量。
bit_op.c

整数
有 2 种类型的整数表示方法：无符号和有符号整数。C 语言标准中定义了若干种整数数据类型，但没有明确规定每种类型的具体取值范围，只定义了一个下界。下表显示了大多数 C 实现中 32 位和 64 位的整数取值范围。

特点：取值范围不对称，负数的范围比正整数大 1。Java 中只支持有符号数。

C 数据类型	32 位最小值	32 位最大值
char	-128	127
unsigned char	0	255
short [int]		
unsigned short [int]		
int		
unsigned [int]		
long [int]		
unsigned long [int]		
long long [int]		
unsigned long long [int]		

无符号数

假设整数有 w 位，其位的表示为位向量 $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$。把 $\vec{x}$ 看成一个二进制表示的数，于是就得到了整数的无符号表示。下列函数定义了该整数的值，B2U 是 Binary to Unsigned 的缩写。

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

很显然，最大值为位向量[11…1]，最小值为 0，

$$UMax_w = \sum_{i=0}^{w-1} 2^i = 2^w - 1$$

函数$B2U_w$能够被定义为一个一一映射$B2U_w: \{0,1\}^w \to \{0, \dots, 2^w - 1\}$，也就是说在 0~$2^w - 1$之间的每个整数都有一个唯一的 w 位的二进制编码序列。

有符号数

有符号数使用补码来表示，将最高有效位解释为负权，因此位向量 $\vec{x}$ 用下面的函数表示，B2T 是 Binary to Two's-complement 的缩写。最高有效位 x_{w-1} 也称为符号位。

$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

考虑其取值的范围，最小值为位向量[10…0]，最大值的位向量为[01…1]

$$TMin_w = -2^{w-1}$$

$$TMax_w = \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$$

函数$B2T_w$能够被定义为一个一一映射 $B2T_w: \{0,1\}^w \to \{-2^{w-1}, \dots, 2^{w-1} - 1\}$。

关于补码，需要注意：1．补码的范围是不对称的$|TMin_w| = |TMax_w| + 1$，这种特性可能造成程序中细微的错误。2．无符号数的最大值刚好比补码的最大值的 2 倍大一：$UMax_w = TMax_w + 1$。下表总结了一些重要的数字，-1 和$UMax_w$ 的位级表示一样。

数	字长			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65,535	0xFFFFFFFF 4,294,967,295	0xFFFFFFFFFFFFFFFF 18,446,744,073,709,551,615
$TMin_w$	0x80 -128	0x8000 -32,768	0x80000000 -2,147,483,648	0x8000000000000000 -9,223,327,036,854,775,808
$TMax_w$	0x7F 127	0x7FFF 32,767	0x7FFFFFFF 2,147,483,647	0x7FFFFFFFFFFFFFFF 9,223,372,036,854,775,807
-1	0xFF	0xFFFF	0xFFFFFFFF	0Xffffffffffffffff
0	0x00	0x0000	0x00000000	0x0000000000000000

C 语言标准并没有规定用补码来表示有符号整数，但大多数编译器都是这样子做的。头文件<limits.h>中定义了一组常量来限定编译器的数据的取值范围，如 INT_MAX、UINT_MAX 等。

由于 C 语言没有明确规定某种数据类型的大小，可移植性程序需要明确编码数据类型。C99 中，头文件 stdint.h 中引入了 intN_t 和 uintN_t 之类的无歧义大小的数据类型（N 取值为 8、16、32、64），也定义了 INT8_MIN、INT8_MAX 等宏。

Java 则明确规定了每种数据类型的取值范围以及编码方式等，保证了在不同机器上的可移植性。

认识反汇编代码中的数是逆向工程中很重要的一个技巧。正数 x 的负数位级补码表示为 -x=(~x)+1，即取反，再加 1；负数的位级表示转换为十进制形式则**先减一，再取反**。如有符号数 0xFFFFE74=~(0xFFFFFE74-1)=-(1 1000 1100)=-0x18C=-396。

整数类型转换、扩展和截断

有符号、无符号数之间的**转换**

大多数 C 实现中，处理长度相同的有符号和无符号数之间相互转换的规则是：数值（对位的解释）可能改变，但位模式不变。定义$U2B_w$为$B2U_w{}^{-1}$，$T2B_w$为$B2T_w{}^{-1}$，有

$$U2T_w(x) = B2T_w\big(U2B_w(x)\big) = -x_{w-1}2^w + x = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases}$$

$$T2U_w(x) = B2U_w\big(T2B_w(x)\big) = x_{w-1}2^w + x = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$

$U2T_w$输入一个 $0\sim 2^w - 1$的数，得到一个 $-2^{w-1}\sim 2^{w-1} - 1$的值；$T2U_w$则相反。需要注意转换时候的映射区间。

C 语言中，常量数字默认是有符号的，如果要创建无符号数，需要在常量数字后面添加符号 u 或 U。符号转换可以使用显式强制类型转换，或者在赋值时发生隐式类型转换，如代码

```
int x;
int y;
unsigned ux;
unsigned uy;
```

```
x = (int) ux;
uy = (unsigned) y;
x = ux;
uy = y;
```

当有符号、无符号数字进行混合运算时，C 语言会隐式的将有符号强制转换成无符号的，然后再进行相应运算。这对于算术运算来说问题不大，但是对于关系运算符如>或<之类的，会导致非直观的结果，导致 bug 发生。

书写 INT_MIN 时也较晦涩，由于有符号数的不对称特点，最小值并不能直接书写。默认为 int 类型。

```
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX – 1)
#define INT_MIN ((int) 0x80000000U) // 这样子书写是不是可以？
```

数的扩展和截断

在不同字长的整数之间进行转换的时候，我们希望数值不变。从字长较短的整数转换为较长时，利用位的扩展方法可以精确保证数值；而从字长较长的整数转换为较短时，则发生截断，此时并不能保证数值不变。

将无符号数转换为更大的无符号整数时，在开头添加 0，进行零扩展运算；而无符号的数据，则扩展符号位，成为符号扩展运算。位模式变化如下（证明略）

$$[x_{w-1}, x_{w-2}, \dots, x_0] \rightarrow [0, .., 0, x_{w-1}, x_{w-2}, \dots, x_0]$$

$$[x_{w-1}, x_{w-2}, \dots, x_0] \rightarrow [x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]$$

见代码 byte_order.c/test_show_bit_expand

当数的扩展与有符号无符号数据类型转换混合运算的时候，其相对顺序会对影响程序的行为，需要特别小心。正确的方式应该是先进行数的扩展，再进行符号转换，以防数值发生变化。这也是 C 语言标准所要求的，因此(unsigned) sx 等价于(unsigned) (int) sx。

数字截断

将一个 w 位的数 $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$截断为一个 k 位数字时，丢弃高 w-k 位，得到位向量 $\vec{x}' = [x_{k-1}, x_{k-2}, \dots, x_0]$。于是截断就可能改变数的值（溢出），相当于进行模运算

$$x \bmod 2^k = B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k$$

$$= \sum_{i=0}^{k-1} x_i 2^i$$

$$= B2U_k([x_{k-1}, x_{k-2}, \dots, x_0])$$

因此，对于有无符号数进行截断有下列结果

$$B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k$$

有符号阶段则为

$$B2T_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = U2T_k\big(B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k\big)$$

有符号、无符号数据之间的隐式数据类型转换会导致很多意料之外的错误。特别需要小心在 C 标准库中，size_t 被定义为 unsigned int，不经意的类型转换可能导致 bug。

Java 认为 unsigned 导致的问题比其带来的好处更多，因此不提供该类型。但在系统编程中，利用 unsigned 当作位集合，常用来作为条件的标记、地址指针、更高精度的数学库等应用。

整数运算

运算需要小心的属性
两个正数相加，可能得到负数
x<y 和 x-y<0 并不相同

整数运算
无符号加法
无符号减法（加法逆元）
无符号乘法

补码加法
补码非
补码乘法

乘法、除法的优化

特性：溢出，正溢出，负溢出

加法

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \le x + y \le 2^{w+1} \end{cases}$$

$$x +_w^t y = U2T_w[T2U_w(x) +_w^u T2U_w(y)] = U2T_w[(x + y) \bmod 2^w]$$

无符号溢出的判定：$s = x +_w^u y$，若 s<x 则溢出。

有符号溢出有几种情况，令 z=x+y（整数和）。
$-2^w \le z < -2^{w-1}$，负溢出，则 $x +_w^t y = x + y + 2^w$，得到一个正数。
$-2^{w-1} \le z < 2^{w-1}$，正常。
$2^{w-1} \le z < 2^w$，正溢出，$x +_w^t y = x + y - 2^w$，得到一个负数。

判断加法是否溢出，参考代码 op_overflow.c

非（加法逆元）

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases}$$

$$-_w^t x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

计算非时，可以利用 C 语言中，-x 和~x+1 得到的结果一样。

乘法

无符号

$$x *_w^u y = (x \cdot y) \bmod 2^w$$

有符号

$$x *_w^t y = U2T_w[(x \cdot y) \bmod 2^w]$$

从这两个公式来看，无符号和有符号的乘法位级结果是一样。

乘以常数

大多数机器上，整数乘法指令相当慢，需要 10 个或更多的始终周期，其他如加法、减法、位级运算和移位等只需 1 个时钟周期。因此编译器会用移位和加法运算的组合来代替乘以常数因子的乘法。

常数可以写成 0 和 1 的位序列，如 14 为 1110，则有两种方式进行乘法替换

A：x<<3 + x<<2 + x<<1

B：x<<4 – x<<1

除法

整数（包括无符号和有符号的正数）除法总是舍入到零。x≥0 和 y>0，则除法结果是⌊x/y⌋。

补码除法，负数时，需要添加一个偏置 biasing。则除法结果有$\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$。

除法不能用除以 2 的幂来表示任意常数 K 的除法。（除法运算的特性决定了，没有除法分配律）

除以 2 的幂

在大多数机器上，整数除法要比整数乘法更慢——需要 30 个或者更多的时钟周期。除以 2 的幂也通过移位实现，无符号和补码分别通过逻辑和算术移位实现。

浮点数

这里介绍 IEEE 浮点数标准，最开始由 Intel 赞助 William Kahan 设计，IEEE 在其基础上修改而成标准。

浮点数表示

浮点数是用原码来表示的，也就说 w 位的位向量 $\vec{x}$ 有以下结果

$$B2S_w(\vec{x}) = (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

浮点数十进制形式可以表示为

$$d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$$

$$d = \sum_{i=-n}^{m} 10^i \cdot d_i$$

对于二进制形式，IEEE 标准使用下列公式来表示浮点数。

$$V = (-1)^s \cdot M \cdot 2^E$$

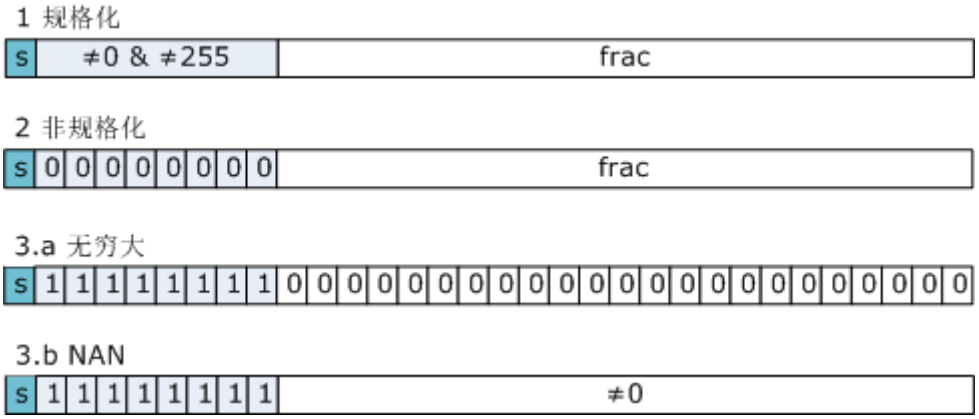
符号 s 决定正负，1 为负，0 为正。
尾数 M 是一个二进制小数，范围是 1~2-ε，或 0~1-ε。
阶码 E 是对浮点数加权，权重是2^E。

单精度 s 为 1 位，exp 为 8 位，frac 为 23 位。
双精度 s 为 1 位，exp 为 11 位，frac 为 52 位。

浮点数的表示（其位模式分别用符号 s, exp, frac 表示）

浮点数值的分类（以单精度为例，Bias 是一个等于$2^k - 1$的偏置值，其中 k 为阶码的位数，单精度为 127，双精度为 1023）。

1. 规格化数。exp 的位模式既不全为 0，也不全为 1。
   阶码 E=e-Bias，e 是 exp 所表示的无符号数，故取值范围为-126 ~ +127。
   尾数 M=1+f，f 是 frac 所表示的小数部分。
2. 非规格化数
   阶码 E=1-Bias
   尾数 M=f。
3. 无穷大



非规格化数使用 E=1-Bias，而不是-Bias 作为偏置，是为了让从非规格化数到规格化数的平滑过渡。可表示的浮点数不是均匀分布的，越靠近原点，数越稠密。

浮点数计算的不精确性，绝对误差在长时间运行时需要特别小心。

C 语言中浮点数的存储的地址是指向低位（小端），故符号位在高位。

舍入 rounding

浮点数只能近似的表示实数运算，对于任意实数 x，我们希望能找到一个最接近它的可表示的浮点数 x′，这就是舍入运算。IEEE 浮点格式定义了四种舍入方法，默认的**向偶数舍入**用来找到最接近的匹配，其他三种用于计算上界和下界。四种舍入方式分别是

1. 向偶数舍入 round-to-even，也成为向最接近的值舍入 round-to-nearest，舍入的方式是将数字向上或向下舍入，使得结果的最低有效数字是偶数。如 1.5 和 2.5 都舍入成 2。
2. 向上舍入，正数和负数都向上舍入，如 1.5 和-1.5 分别舍入成 2 和-1。
3. 向下舍入，正数和负数都向下舍入，如 1.5 和-1.5 分别摄入成 1 和-2。
4. 向零舍入，正数向下舍入，负数向上舍入。如 1.5 和-1.5 分别舍入成 1 和-1。

向偶数舍入避免了在舍入一组数值时，在这些舍入值中引入统计偏差，因为它以 50% 的概率分别向上、向下舍入一组数。

浮点运算

浮点运算 op，实际将产生 round(x op y)，这是对精确计算进行舍入的结果。实际运算时，浮点单元的设计者并不会进行 x op y 的精确运算，只要保证满足舍入的结果就行。当运算数是某个特殊值，如-0、-∞、NaN 时，定义了一些使之更加合理的规则，如 1/-0 将产生-∞，而 1/+0 将得到+∞。

运算定律必须考虑舍入的影响，x+y 是可交换的，但不是可结合的。加法的不可结合是浮点运算中较重要的特性，需要注意。编译器不能对其进行响应的优化，如

```
x = a + b + c;
y = b + c + d;
```

不能优化成（可以省略一个浮点加法）

```
t = b + c;
x = a + t;
y = t + d;
```

浮点加法都有逆元，但无穷和 NaN 不存在逆元，+∞+(-∞)=NaN，NaN+x=NaN。

浮点加法满足单调性，如果 a≥b，那么对于任何 a、b 以及 x 的值，除了 NaN，都有 x+a≥x+b。而对于无符号、有符号加法都不满足单调性。

乘法可交换，不可结合，在加法上不具备分配性（舍入的影响）。缺乏结合性和分配性对编译器、科学计算程序员来说是比较严重的问题。

C 语言中的浮点数

标准 C 并没规定一定要求使用 IEEE 浮点，但绝大多数机器都使用该标准。GNU C 库中定义了一下常量

```
#define _GNU_SOURCE 1
#include <math.h>
```

```
#define INFINITY
#define NAN
```

C99 还支持 long double，对 Intel 兼容机器来说，使用了 80 位的扩展精度浮点数。而其他许多机器等价于 double。

数据类型转换规则：
1. 从 int 转换成 float，数字不会溢出，但可能被舍入。
2. int 或 float 转换成 double，能精确保留数值。
3. double 转换成 float，可能溢出为+∞或-∞，也可能舍入。
4. 从 float 或 double 转换成 int，值向零舍入。存在溢出的可能。C 语言标准并没有固定这种结果。在 Intel 兼容的机器上，TMin（位模式为[10…0]）被定义为整数不确定值。当从浮点数转换成整数时，如果不能为该值找到一个合理的整数近似值，则会产生该值。

字符

语言、文化（如时间、货币、排序方式等）和代码集
所有特定于文化的信息汇集于单个环境中，成为 GLS(Global Language Support).

字符 Character
字符集 Character Set
字符（Character）是各种文字和符号的总称，包括各国家文字、标点符号、图形符号、数字等。

字符集（Character set）是多个字符的集合，字符集种类较多，每个字符集包含的字符个数不同，常见字符集名称：ASCII 字符集、GB2312 字符集（简体中文）、BIG5 字符集（繁体中文）、 GB18030 字符集（亚洲字符集合）、Unicode(常用 UTF-8) 字符集等。

标准机构
ISO/IEC
    ISO/IEC 646 ASCII
    ISO/IEC 8859 字符集，如 Latin-1(ISO/IEC 8859-1)
    ISO/IEC 10646
The Unicode Consortium
    1980 年代末，位于加州，允许商业机构组织和个人加入； ISO 也成立了 ISO-10646 工作小组；后来，两个组织意识到没有两个标准的必要，因此二者保持了最大程度的兼容性，Unicode 包含了更具体的实现细节等主题。

中文标准
国家标准代码
Big5

UTF-8

参考资料

[1] 深入理解计算机系统，2nd。第二章，信息的表示和处理。
[2] Programming Perl, 4th. Chapter 6, Unicode.