Data Types

王慧妍

why@nju.edu.cn

南京大学 软件学院 计算机软件研究所

回顾一下

- 我们见过的数据类型
  - `int`
  - `double, float`
  - `char`

  - 数组

C的类型要求

- C是一个有类型的语言
  - 变量使用前必须要定义，并且需要确定类型
- 后续发展
  - C++/JAVA：强调类型，更严格的类型检查系统
  - JavaScript，Python，PHP：不看重类型，甚至不需要事先定义
- 观点
  - 认为明确的类型有助于尽早发现程序中的简单错误
  - 过于强调类型迫使程序员面对底层、实现而非事务逻辑

C的类型系统

- 整型
  - char, short, int, long, long long, bool, ……
- 浮点型
  - float, double, long double
- 指针类型
- 自定义类型

数的范围

- 整数
  - char：一个字节（8bit)
  - short：两个字节
  - int：通常表示为一个字长，（如，常见四个字节）
  - long：通常表示为一个字长，（如，常见四个字节）
  - long long：8字节

- 一个字节（8bit）可以表达的数
  - 2^8个
  - 18 → 00010010

整型类型

- 数据在内存中以二进制形式存放
  - E.g., `signed/unsigned int`

- 原码表示法
  - 最高位为符号位
  - 容易理解，　但是：
    - 0 的表示不唯一，故不利于程序员编程
    - 加、减运算方式不统一
    - 需额外对符号位进行处理，故不利于硬件设计
    - 特别当 a<b时，实现 a-b比较困难

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	Binary
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

整型类型

- 模运算的用处
  - 在一个模运算系统中，一个数与它除以“模”后的余数等价。

时钟是一种模12系统

假定钟表时针指向10点，要将它拨向6点， 则有两种拨法：

① 倒拨4格：10-4 = 6

② 顺拨8格：10+8 = 18 ≡ 6 (mod 12)

模12系统中： 10- 4 ≡ 10+8 (mod 12)

- 4 ≡ 8 (mod 12)

则，称8是- 4对模12的补码 （即：-4的模12补码等于8）。

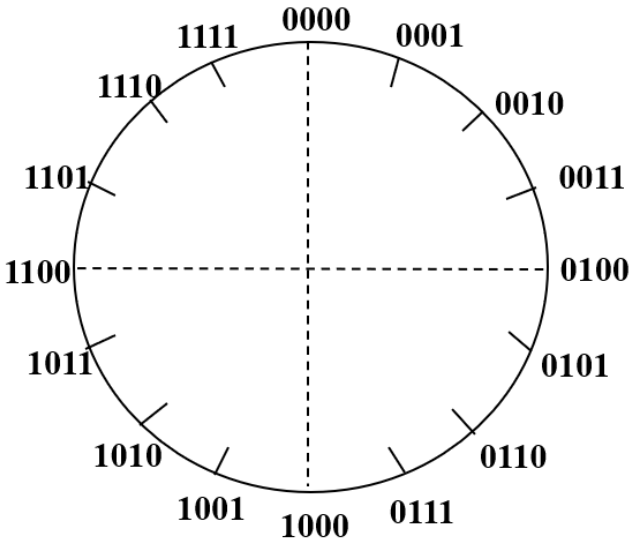
同样有 -3 ≡ 9 (mod 12)

-5 ≡ 7 (mod 12) 等

补码（modular运算）： + 和– 的统一

整型类型

- 采用补码表示法
  - “8位二进制数” 模运算系统（mod 2^8）
    - 0 → 00000000
    - 1 → 00000001
    - -1 → 11111111
    - 补码和原码可以加出一个溢出8位的0
    - 0111 1111 - 0100 0000 = ?



一个负数的补码等于对应正数补码的“各位取反、末位加1”

- 10000000 → -128
- 01111111 → 127

- “32位十进制数” 模运算系统（mod 2^{32}）

unsigned/signed int类型

- 无符号整数
  - 32位整数：0x0 ~ 0xFFFFFFFF（32个1）

- 带符号整数
  - 补码表示法（普遍采用）：各位取反末位加一
    - 用加法来实现减法
    - 00000000 → 0
    - 111111111 → -1
    - 10000000 → -128
    - 01111111 → 127
    - 比原码表示多表示一个最小的负数

整型类型

- Signed（有符号数）
  - `short int`
  - `int`
  - `long`
  - `long long`

- Unsigned（无符号数）
  - `bool (stdbool.h)`
  - `unsigned short int`
  - `unsigned int`
  - `unsigned long`
  - `unsigned long long`

类型	字节数	位数	取值范围	格式匹配符
char	1	8	$-2^7 \sim 2^7-1$	%c, %d
signed char	1	8	$-2^7 \sim 2^7-1$	%c, %d
unsigned char	1	8	$0 \sim 2^8-1$	%c, %d
signed short int	2	16	$-2^{15} \sim 2^{15}-1$	%hd
unsigned short int	2	16	$0 \sim 2^{16}-1$	%hu
signed int	4	32	$-2^{31} \sim 2^{31}-1$	%d
unsigned int	4	32	$0 \sim 2^{32}-1$	%u
signed long int	4	32	$-2^{31} \sim 2^{31}-1$	%ld
unsigned long int	4	32	$0 \sim 2^{32}-1$	%lu
signed long long int	8	64	$-2^{63} \sim 2^{63}-1$	%lld
unsigned long long int	8	64	$0 \sim 2^{64}-1$	%llu

short<=int<=long

类型所占机器位数与特定编译器平台相关

sizeof（静态运算符，编译时决定，不要在括号内做运算）

有符号数和无符号数

- C语言标准规定：
  - 若运算中同时有无符号和带符号整数，则按无符号整数运算

关系 表达式	类型	结果	说明
0 = = 0U	无	1	00…0B = 00…0B
-1 < 0	带	1	11…1B (-1) < 00…0B (0)
-1 < 0U	无		
2147483647 > -2147483647 - 1	带	1	011…1B (2^{31}-1) > 100…0B (-2^{31})
2147483647U > -2147483647 - 1	无		
2147483647 > (int) 2147483648U	带		
-1 > -2	带	1	11…1B (-1) > 11…10B (-2)
(unsigned) -1 > -2	无		

回忆一下：sizeof和strlen的返回值是什么类型？

计算机内部只有二进制

- 重点是外界你打算如何看待

```c
#include<stdio.h>

int main(){
    int x = -1;
    unsigned u = 2147483648;
    printf ( "x = %u = %d\n", x, x);
    printf ( "u = %u = %d\n", u, u);
    return 0;
}
```

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

```c
    int b = -1;
    char c = -1;
    printf("%u, %u\n", b, c);
```

```
4294967295, 4294967295
```

Integral Promotion（整型提升）

- Otherwise, the operand has integer type (because `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and unscoped enumeration were promoted at this point) and integral conversions are applied to produce the common type, as follows:
  - If both operands are signed or both are unsigned, the operand with lesser *conversion rank* is converted to the operand with the greater integer conversion rank
  - Otherwise, if the unsigned operand's conversion rank is greater or equal to the conversion rank of the signed operand, the signed operand is converted to the unsigned operand's type.
  - Otherwise, if the signed operand's type can represent all values of the unsigned operand, the unsigned operand is converted to the signed operand's type
  - Otherwise, both operands are converted to the unsigned counterpart of the signed operand's type.

The *conversion rank* above increases in order `bool`, `signed char`, `short`, `int`, `long`, `long long`. The rank of any unsigned type is equal to the rank of the corresponding signed type. The rank of `char` is equal to the rank of `signed char` and `unsigned char`. The ranks of `char8_t`, `char16_t`, `char32_t`, and `wchar_t` are equal to the ranks of their underlying types.

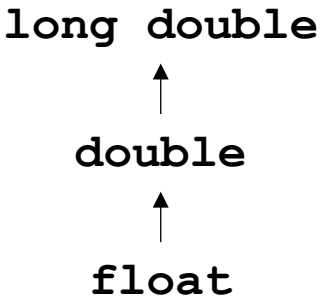
C语言中整型运算总是**至少**以**缺省整型**类型的**精度**来进行的。

类型转换

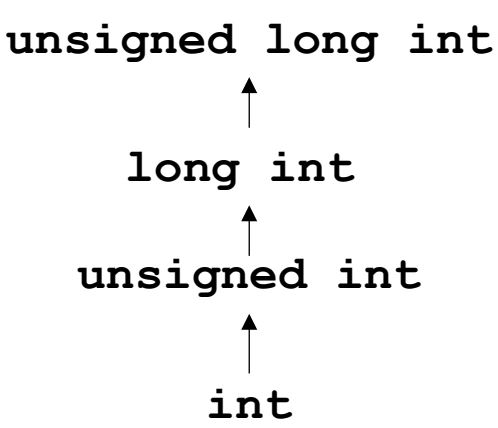
- 隐式类型转换
  - 算术表达式或逻辑表达式的操作数类型不同时（常规算术转换）
  - 赋值运算右侧表达式类型与左侧变量类型不匹配时
  - 函数调用时实参与形参不匹配时
  - return表达式类型与返回值不同时

隐式1：常规算术转换

- “狭小” -> “宽松”
  - 任一操作数类型是浮点数



  - 两个操作数类型都不是浮点数（整型提升后）




```
int i = -10;
unsigned int n = 10;
if(i < n) ????
```

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;//c is converted to int
i = i + s;//s is converted to int
u = u + i;//i is converted to unsigned int
l = l + u;//u is converted to long int
ul = ul + l;//l is converted to unsigned long int
f = f + ul;//ul is coverted to float
d = d + f;//f is converted to double
ld = ld + d;//d is coverted to long double
```

隐式2：赋值转换

- 赋值右侧转换为左侧的类型

```
char c;
int i;
float f;
double d;

i = c;
f = i;
d = f;
```

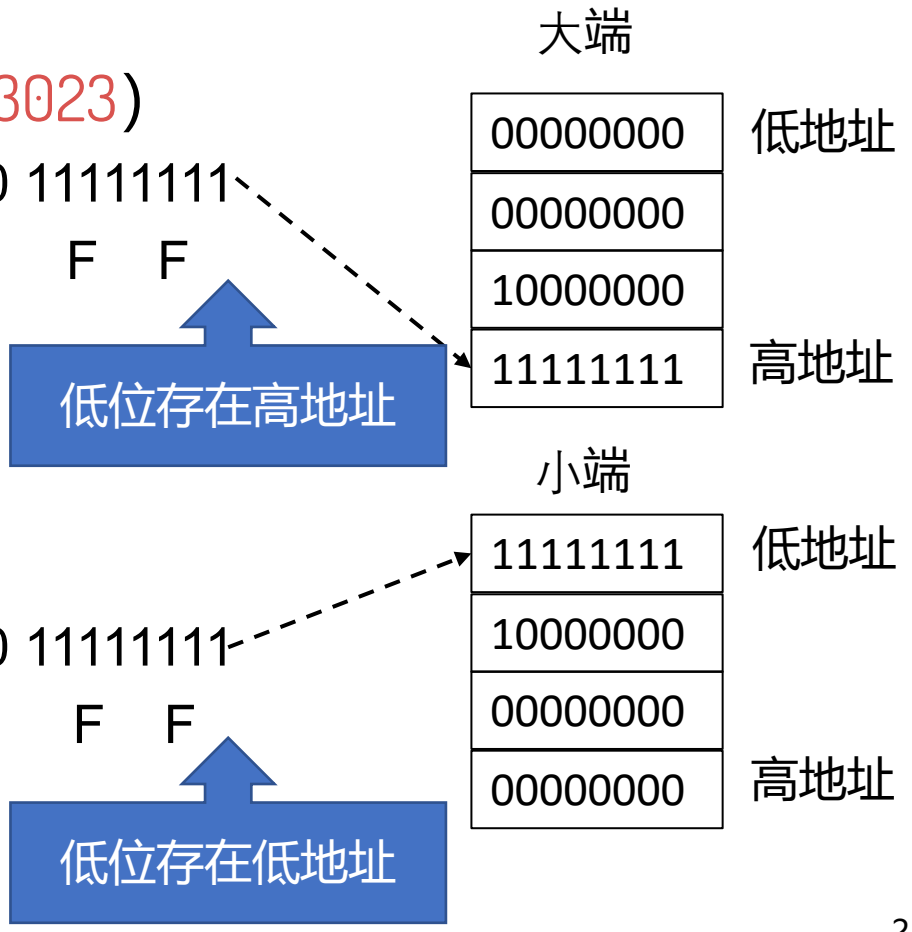
  - 浮点数赋值给整数会丢掉该数的小数部分
  - 把某一个类型数赋给更狭小的变量可能得到无意义结果
    - c = 10000;
    - i = 1.0e20;
    - f = 1.0e100;
  - f = 3.1415f;//建议加上f

整型类型

- 数据在内存中以二进制形式存放
  - E.g., `signed/unsigned int`

char c = 10000;

- 大端/小端存储方式（int a = 33023）
  - 00000000 00000000 10000000 11111111
    0 0 0 0 8 0 F F

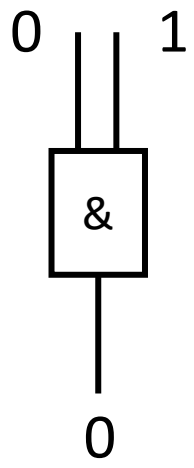


  - 00000000 00000000 10000000 11111111
    0 0 0 0 8 0 F F

一点bonus知识
位运算

为什么会有位运算

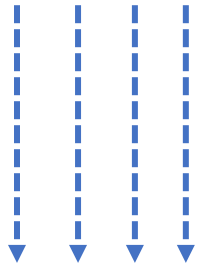
- 逻辑门和导线是构成计算机 (组合逻辑电路) 的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)



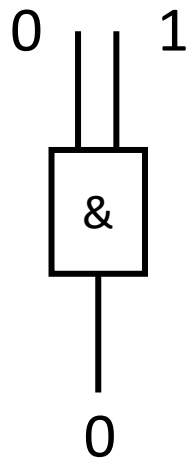
位运算与逻辑电路密不可分

a -> 0 1 0 1　　5

b -> 0 1 1 0　　10



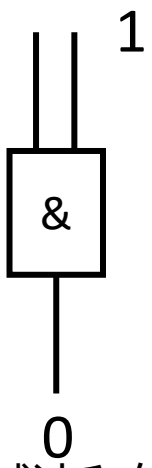
a & b -> 0 1 0 0　　8



加法呢?

为什么会有位运算

- 逻辑门和导线是构成计算机 (组合逻辑电路) 的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)



  - 例子：一代传奇处理器 8-bit MOS 6502
    - 3510 晶体管；56 条指令，算数指令仅有加减法和位运算

Instructions by Name

ADC .... add with carry
AND .... and (with accumulator)
ASL .... arithmetic shift left
BCC .... branch on carry clear
BCS .... branch on carry set
BEQ .... branch on equal (zero set)
BIT .... bit test
BMI .... branch on minus (negative set)
BNE .... branch on not equal (zero clear)
BPL .... branch on plus (negative clear)
BRK .... break / interrupt

BVC .... branch on overflow clear
BVS .... branch on overflow set
CLC .... clear carry
CLD .... clear decimal
CLI .... clear interrupt disable
CLV .... clear overflow
CMP .... compare (with accumulator)
CPX .... compare with X
CPY .... compare with Y
DEC .... decrement
DEX .... decrement X
DEY .... decrement Y

为什么会有位运算

- 逻辑门和导线是构成计算机 (组合逻辑电路) 的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)
  - 例子：一代传奇处理器 8-bit MOS 6502
    - 3510 晶体管；56 条指令，算数指令仅有加减法和位运算
  - 数学上自然的整数需要实现成固定长度的 01 字符串

整数：固定长度的 Bit String

- 142857 -> 0000 0000 0000 0010 0010 1110 0000 1001
  - 假设 32-bit 整数；约定 MSB 在左，LSB 在右

- 热身问题：字符串操作
  - 分别取出 4 个字节

x: 5 -> 0101

(x>>1)&1

~~010~~ &
001
0

怎么取出来？

x: 0000 0000 0000 0010 0010 1110 0000 1001

(x>>16)&0xFF

x&0xFF

练习

- 打印整数的二进制表达
  - printBinary.c

整数溢出与 Undefined Behavior

Undefined Behavior (UB)

Undefined behavior (UB) is the result of executing computer code whose behavior is not prescribed by the language specification to which the code adheres, for the current state of the program. This happens when the translator of the source code makes certain assumptions, but these assumptions are not satisfied during execution. -- Wikipedia

- C对UB的行为是不做任何约束的
  - 常见的 UB：非法内存访问 (空指针解引用、数组越界、写只读内存等)、被零除、有符号整数溢出、函数没有返回值……
    - 通常的后果比较轻微，比如 wrong answer, crash

为什么 C/C++ 会有 UB?

- 为了尽可能高效 (zero-overhead)
  - 不合法的事情的后果只好 undefined 了
  - Java, js, python, ... 选择所有操作都进行合法性检查

- 为了兼容多种硬件体系结构
  - 有些硬件 `/0` 会产生处理器异常
  - 有些硬件啥也不发生
  - 只好 undefined 了

选择整数类型

- 为什么整数要有多少种?
  - 为了准确表达内存，做底层程序需要
  - 现代计算机普遍字长32或者64位，更适合做int计算
  - 现代编译器普遍会进行内存对齐，所有更短的类型实际上在内存中也可能占据一个int大小（sizeof不一致）

- unsigned是否只是输出视角不同，不影响内部存储
  - unsigned设计的初衷，是为了进行纯二进制运算
  - 位运算：&, |, ~ , ^, >>, <<,

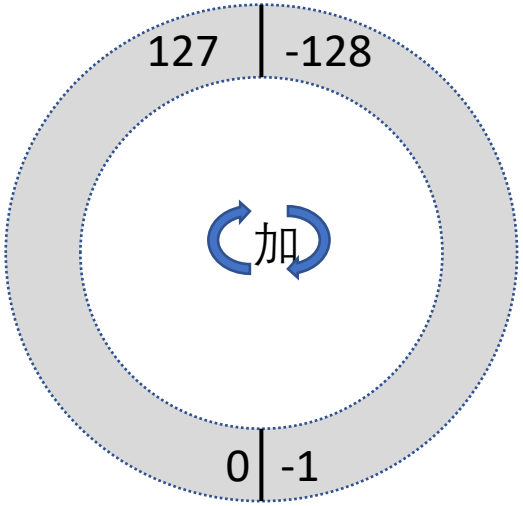
Undefined Behavior: 警惕整数溢出

表达式	值
`UINT_MAX+1`	0
`INT_MAX+1; LONG_MAX+1`	undefined
`char c = CHAR_MAX; c++;`	varies (???)
`1 << -1`	undefined
`1 << 0`	1
`1 << 31`	undefined
`1 << 32`	undefined
`1 / 0`	undefined
`INT_MAX % -1`	undefined

- W. Dietz, et al. Understanding integer overflow in C/C++. In *Proceedings of ICSE*, 2012.

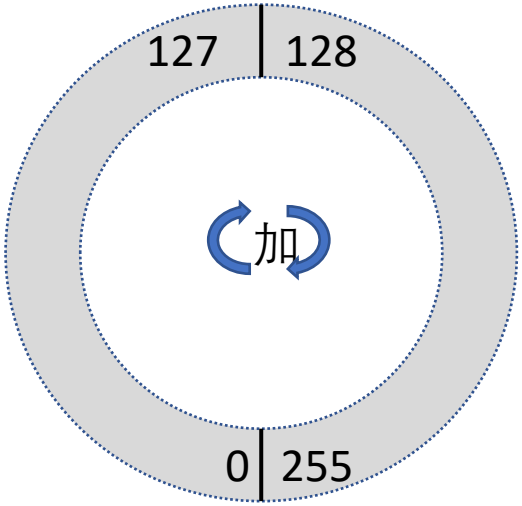
整数溢出

- 带符号整数
  - (signed) char



00000000 →0
01111111 → 127
10000000 → -128
111111111 → -1

- 无符号整数
  - unsigned char



00000000 →0
01111111 → 127
10000000 → 128
111111111 → 255

整数溢出和编译优化

```
int f() { return 1 << -1; }
```

- 根据手册，这是个UB，于是clang这样处置

```
0000000000000000 <f>:
   0:   c3      retq
```

- 编译器把这个计算直接删除了

W. Xi, et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of SOSP*, 2013.

  - `if(UB==0) yes; else if(UB!=0) no; //???`

Undefined Behavior: 位运算相关

表达式	值
`1 << -1`	undefined
`1 << 0`	1
`1 << 31`	undefined
`1 << 32`	undefined

```
1<<31
1u<<31
1LL<<31
```

- W. Dietz, et al. Understanding integer overflow in C/C++. In *Proceedings of ICSE*, 2012.

整型数安全编码标准（INT）

- 使用正确的整数类型
- 确保无符号整数运算不产生回绕
- 确保有符号整数运算不造成溢出
- 确保除法与余数运算不造成除0错误
- 确保整数转换不会造成数据丢失或者错误解释

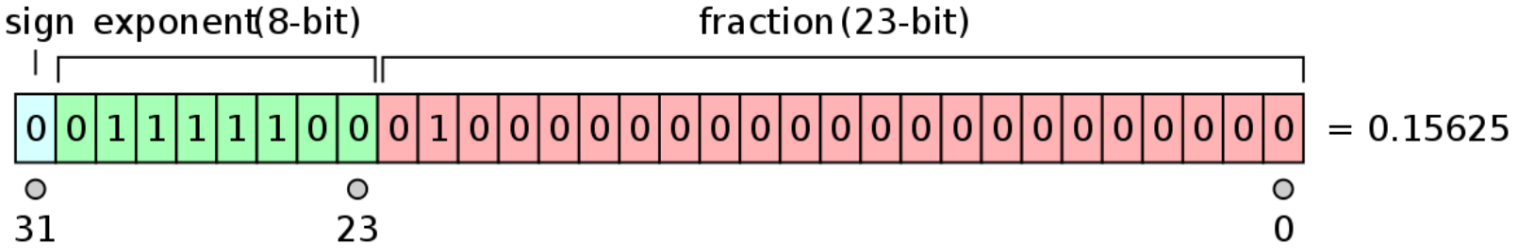
浮点数：IEEE 754

实数的计算机表示

- 实数非常非常多
  - 只能用32/64-bit 01串来表述一小部分实数
    - 确定一种映射方法，把一个01串映射到一个实数
      - 运算起来不太麻烦
      - 计算误差不太可怕

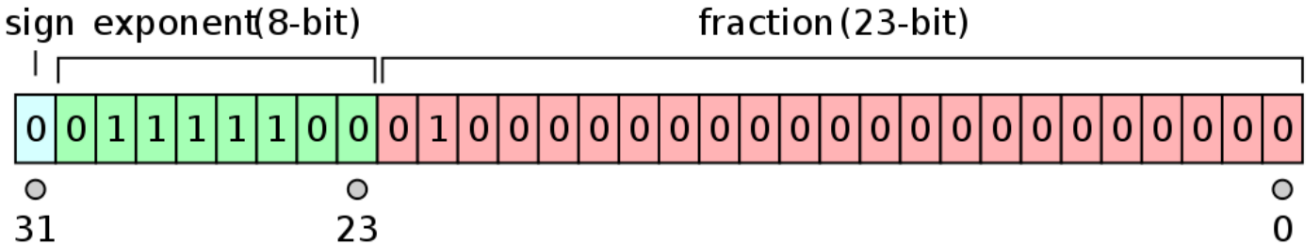
- 于是有了IEEE754
  - 1bit S, 23/53bits Fraction (尾数), 8/11bits Exponent (阶码，B=127)

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$



实数的计算机表示

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$



$(-1)^0 \times 1.01_2 \times 2^{124-127}$

$1.01_2 \times 2^{-3}$

0.00101_2

$0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$

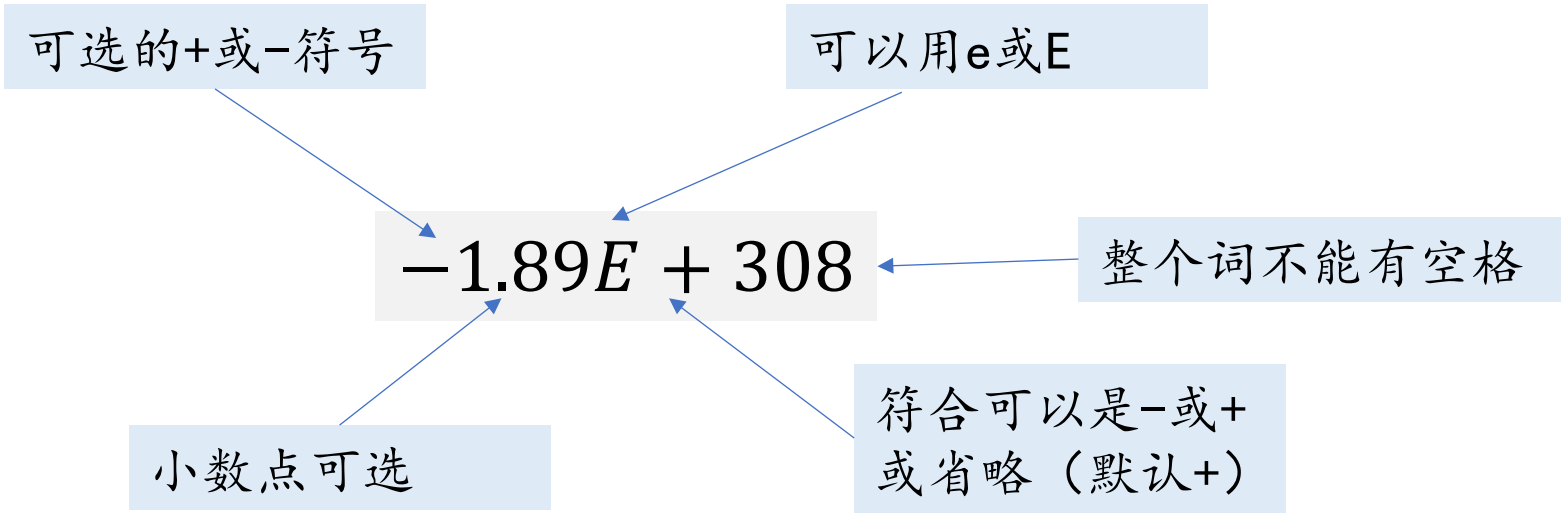
0.15625

类型	字节数	位数	规范	取值范围	输入符	输出符
float	4	32	S1|E8|F23	$\pm 1.2E-38 \sim \pm 3.4E+38$	%f	%f, %e
double	8	64	S1|E11|F52	$\pm 2.2E-308 \sim \pm 1.8E+308$	%lf	%f, %e
long double	10/12/16	80/96/128				

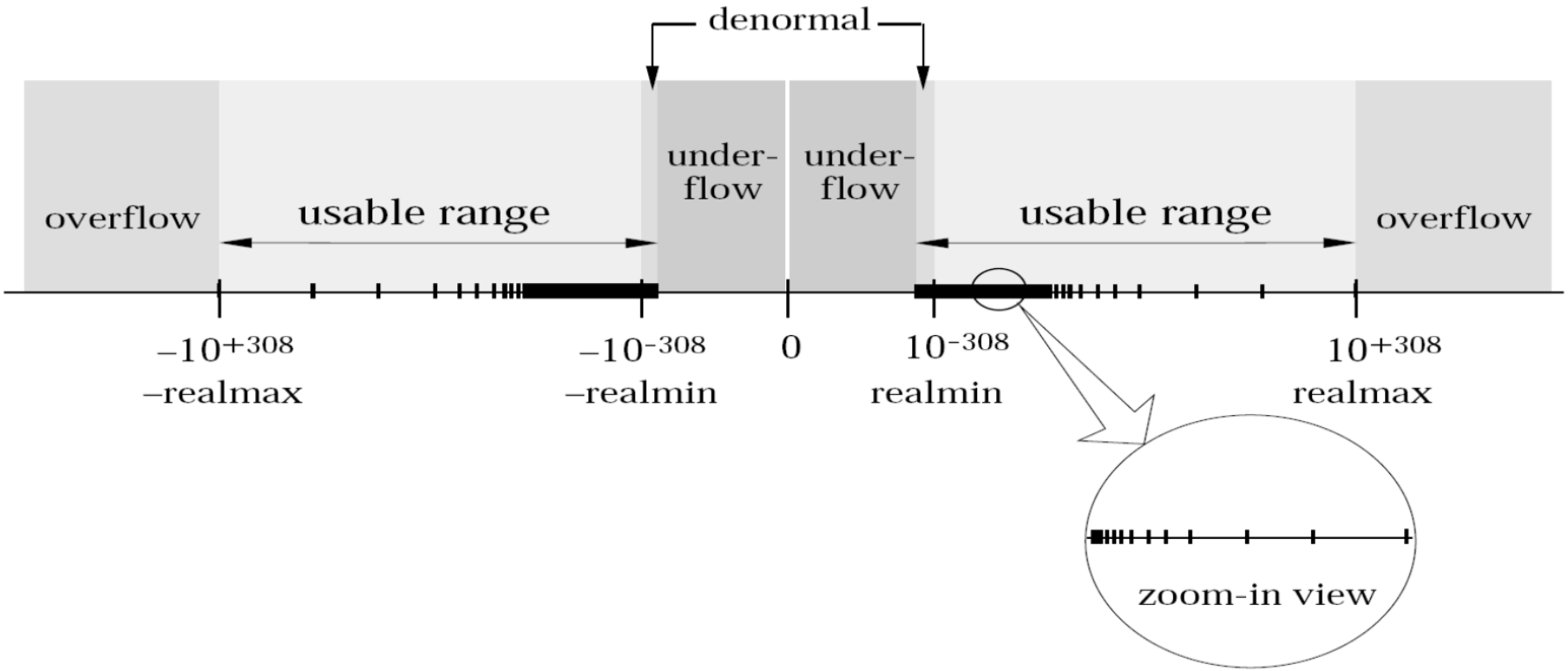
类型所占机器位数与特定编译器平台相关：sizeof

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$

科学计数法

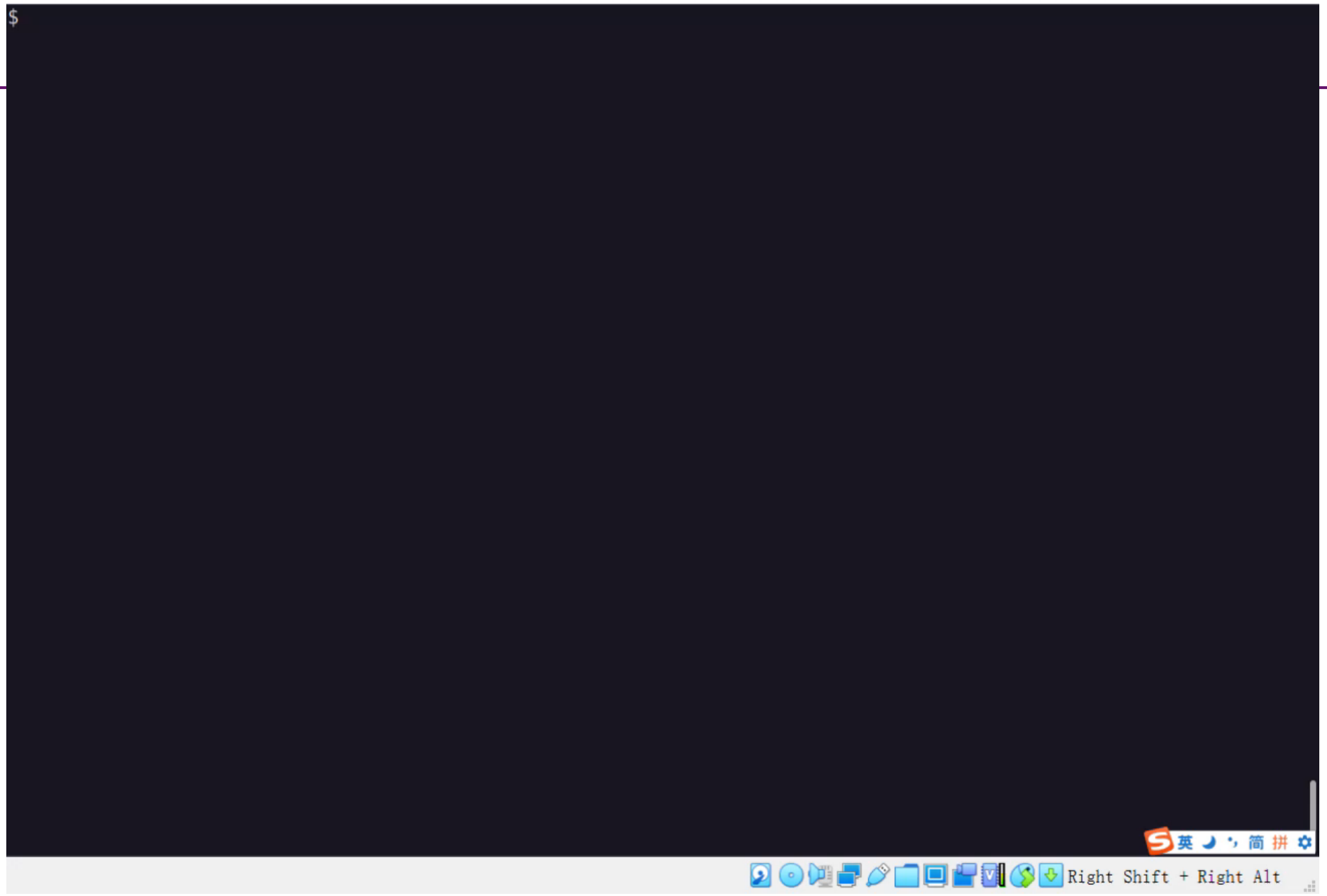


Floating Point Number Line



IEEE754: 你有可能不知道的事实

- 一个有关浮点数大小/密度的实验(float.c)
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化


管理 控制 视图 热键 设备 帮助
$
英 简 拼
Right Shift + Right Alt

IEEE754: 你有可能不知道的事实

- 一个有关浮点数大小/密度的实验(float.c)
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化

  - $1.00000000000000000000000_2 \times 2^{24}$
    - 16,777,216
  - $1.00000000000000000000001_2 \times 2^{24}$
    - 16,777,218

- 比较
  - a == b 需求谨慎判断（要假设自带ε）
  - 浮点数：$(a+b)+c \neq a+(b+c)$

IEEE754: 你有可能不知道的事实

- 一个有关浮点数大小/密度的实验(float.c)
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化

- 例子：计算$1+\frac{1}{2}+\frac{1}{3}...+\frac{1}{n}$

```
#define SUM(T, st, ed, d) ({ \
    T s = 0; \
    for (int i = st; i != ed + d; i += d) \
        s += (T)1 / i; \
    s; \
})
```

管理 控制 视图 热键 设备 帮助

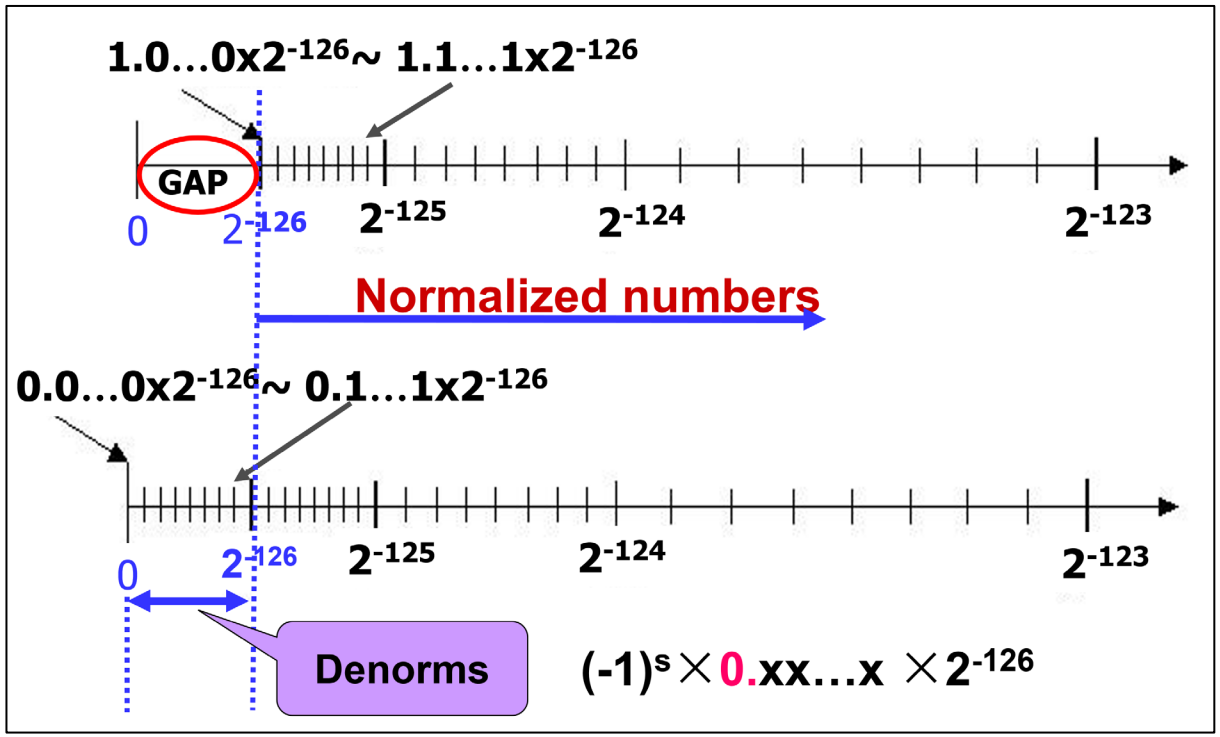
```c
#define SUM(T, st, ed, d) ({ \
  T s = 0; \
  for (int i = st; i != ed + d; i += d) \
    s += (T)1 / i; \
  s; \
})

#define n 1000000

int main(){
  printf("%.16f\n", SUM(float, 1, n, 1));
  printf("%.16f\n", SUM(float, n, 1, -1));
  printf("%.16f\n", SUM(double, 1, n, 1));
  printf("%.16f\n", SUM(double, n, 1, -1));
}

```

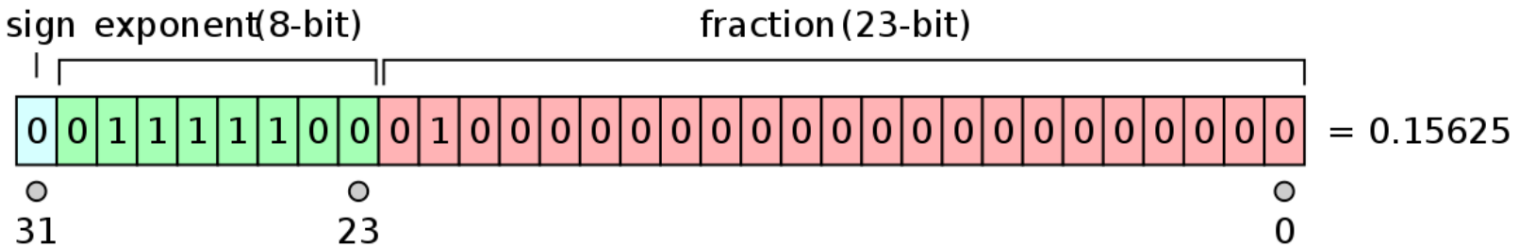
~/Documents/ICS2021/teach/sum.c[1] [c] unix utf-8 Ln 1, Col 1/16

Right Shift + Right Alt

IEEE754: 你可能不知道的事实 (cont'd)

- 非规格化数 (Exponenent == 0且Fraction != 0)
  - $x = (-1)^S \times (0.F) \times 2^{-126}$

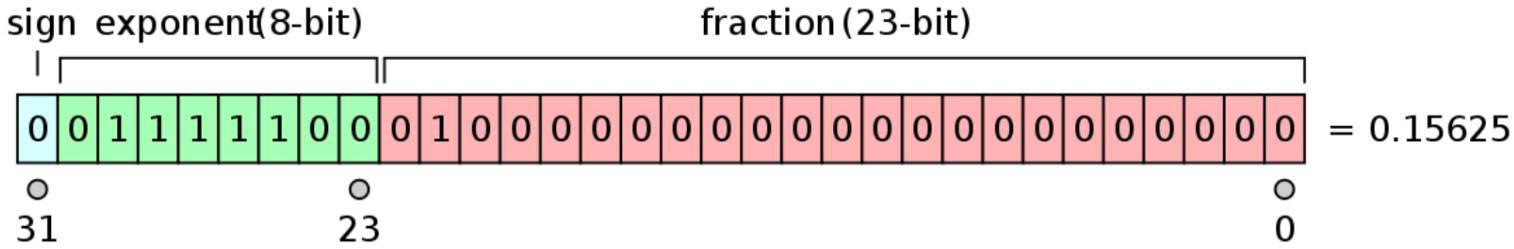




规格化和非规格化浮点数

- 零 (Exponenent == 0且Fraction == 0)
  - +0.0,-0.0的Sbit是不一样的，但+0.0 == -0.0

- Inf/NaN (Not a Number) (Exponenent == 255)
  - Inf：尾数为零，浮点数溢出很常见，不应该作为ub，+/-Inf
  - NaN(0.0/0.0)：尾数非零，能够满足x!=x表达式的值



浮点运算的精度

- `float`：大约6-7位有效数字
- `double`：大约15-16位有效数字

```
// Absolute tolerance comparison of x and y
if (Abs(x - y) <= EPSILON) …
```

```
// Relative tolerance comparison of x and y
if (Abs(x - y) <= EPSILON * Max(Abs(x), Abs(y)) …
```

```
if (Abs(x - y) <= Max(absTol, relTol * Max(Abs(x), Abs(y)))) …
```

Floating-point tolerances revisited – realtimecollisiondetection.net – the blog

浮点数的选择

- double精度优于float
- 现代CPU也可以对double直接做运算
- 涉及复杂小数运算，适当归一化计算

浮点数安全编码标准（FLP）

- 不要使用浮点数变量作为循环计数器
- 避免或者检测数学函数中的定义域与值域错误
- 确保浮点数转换在新类型的范围中

浮点数的精度掌握是极难的

- math.h

自动类型转换

- 当运算符两边出现不一致的类型时，会自动转换为较大的类型
  - 表达的数的范围更大
  - char → short → int → long → long long
  - int → float → double

- 对于printf
  - 任何小于int的类型会被转换为int，float会被转换为double
  - scanf不行，需要指明%hd，%u，%lf

强制类型转换

- 需要把一个量强制转换成另一个类型（通常是较狭小的类型）
  - （类型）变量值
    - `(int)10.3`
    - `(short)32`
  - 安全性可能不能保证
    - `(short)32768`
    - `(char)32768`
  - 不改变原来变量的值和类型
  - 强制类型转换优先级高于四则运算

关于怨声载道的平衡三进制

- b-ternary-1.c
- b-ternary-2.c

- 既然学了函数，我们就可以拆解问题到子问题
  - 问题规模越小，代码越不容易出错
  - 代码更加清楚，各部分定义好交互规约即可

End.