Data Types

王慧妍
why@nju.edu.cn

南京大学 计算机科学与技术系 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`

- 通常采用补码表示法
  - 0 → 00000000
  - 1 → 00000001
  - -1 → 11111111

  - 补码和原码可以加出一个溢出的0

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

- 带符号整数
  - 原码表示法：最高位为符号位
  - 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

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）

计算机内部只有二进制

- 重点是外界你打算如何看待

```
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.

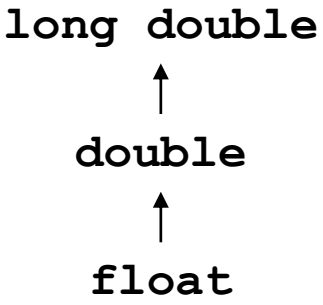
C语言中整型运算总是**至少**以**缺省整型**类型的**精度**来进行的。

类型转换

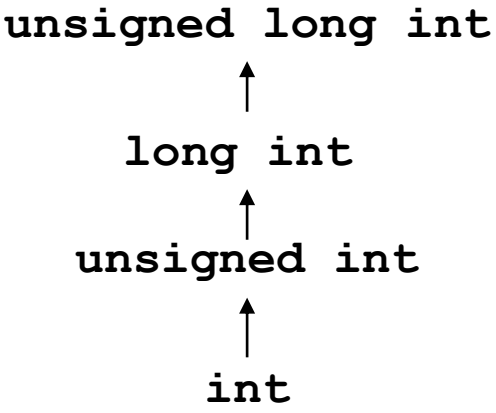
- 隐式类型转换
  - 算术表达式或逻辑表达式的操作数类型不同时（常规算术转换）
  - 赋值运算右侧表达式类型与左侧变量类型不匹配时
  - 函数调用时实参与形参不匹配时
  - return表达式类型与返回值不同时

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

printf 回顾

```
int printf(const char *__restrict__ _Format, ...)
```

- <stdio.h>
- 参数：格式串，可变长的参数列表
- 格式串包括普通字符和以%开始的转换说明

转换说明	传入数据类型	输出
%d (%i)	int	decimal ([-]dddd)
%f	double	decimal ([-]ddd.ddd)
%e (%E)	double	decimal ([-]d.ddde[+-]dd)
%g (%G)	double	%f or %e
%c	int	character
%s	pointer to a char array	string
%%		%

scanf 回顾

```
int scanf(const char *__restrict__ _Format, ...)
```

- <stdio.h>
- 参数：格式字符串，可变长的参数列表
- 以%开始的转换说明

空白字符：包括空格符，水平和垂直制表符/换页符和换行符

转换说明	传入参数	输出
%d	跳过空白字符，匹配int	pointer to int
%le, %lf, %lg	跳过空白字符，匹配double	pointer to double
%e, %f, %g	跳过空白字符，匹配float	pointer to float
%c	a character	pointer to a char
%s	a sequence of non-white-spaces	pointer to a char array
%[abc]	scanlist	pointer to a char array
%[^abc]	not in scanlist	pointer to a char array

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，把电脑炸了都行
  - 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：&, |, ^, >>, <<,

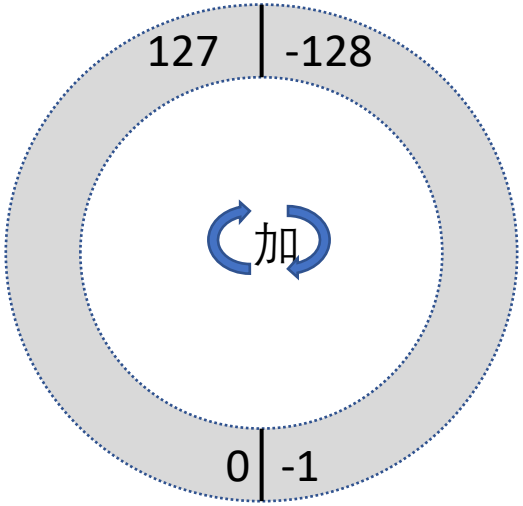
Undefined Behavior: 警惕整数溢出

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.

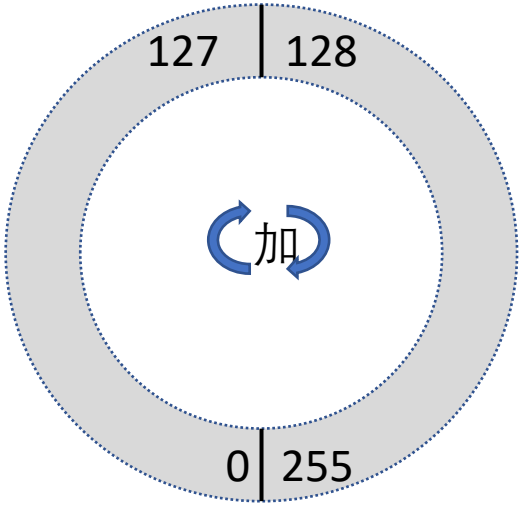
整数溢出

- 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; //???`

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

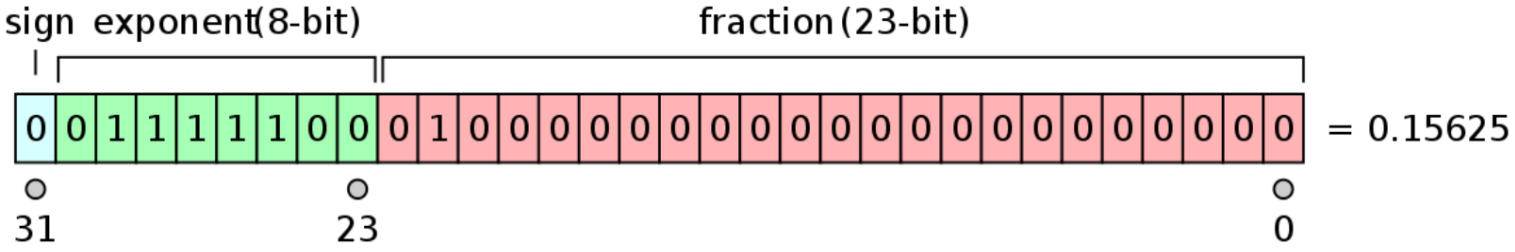
浮点数：IEEE 754

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

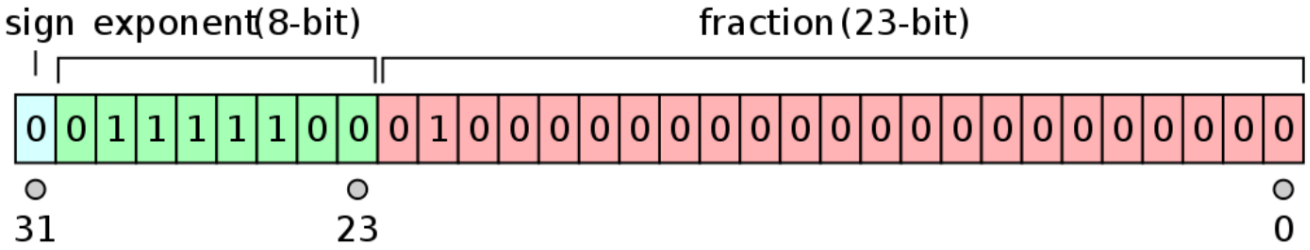
- 于是有了IEEE754
  - 1bit S, 23/53bits Fraction (尾数), 8/11bits Exponent (阶码),

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



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

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

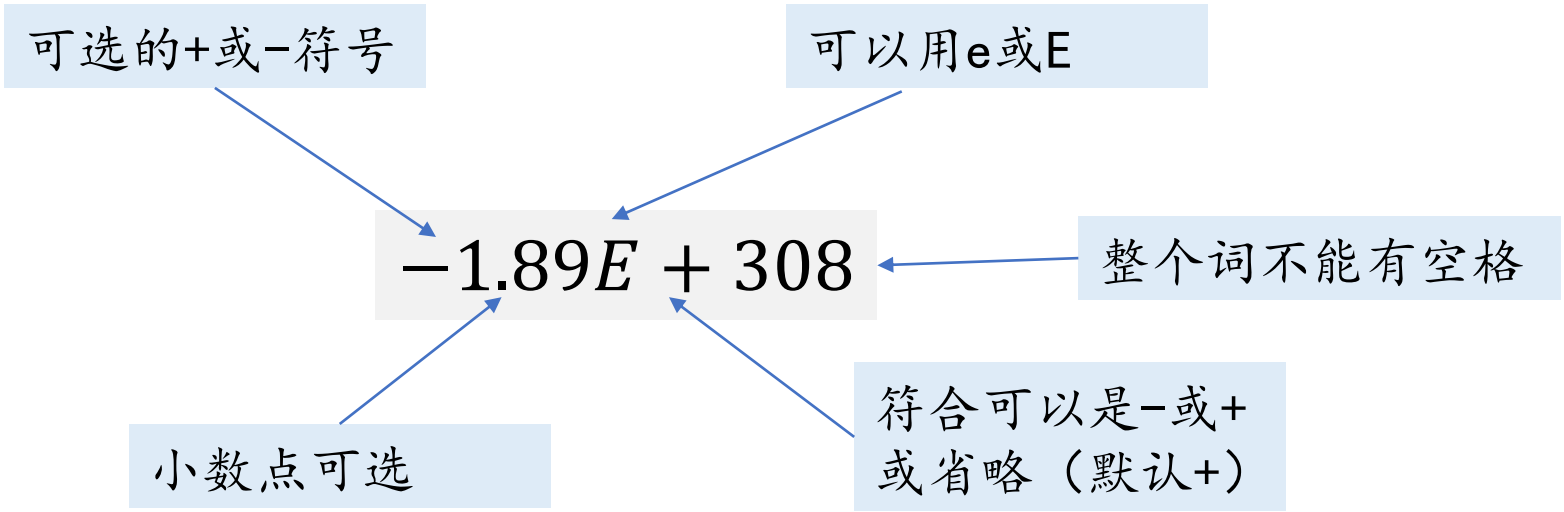
0.15625

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				

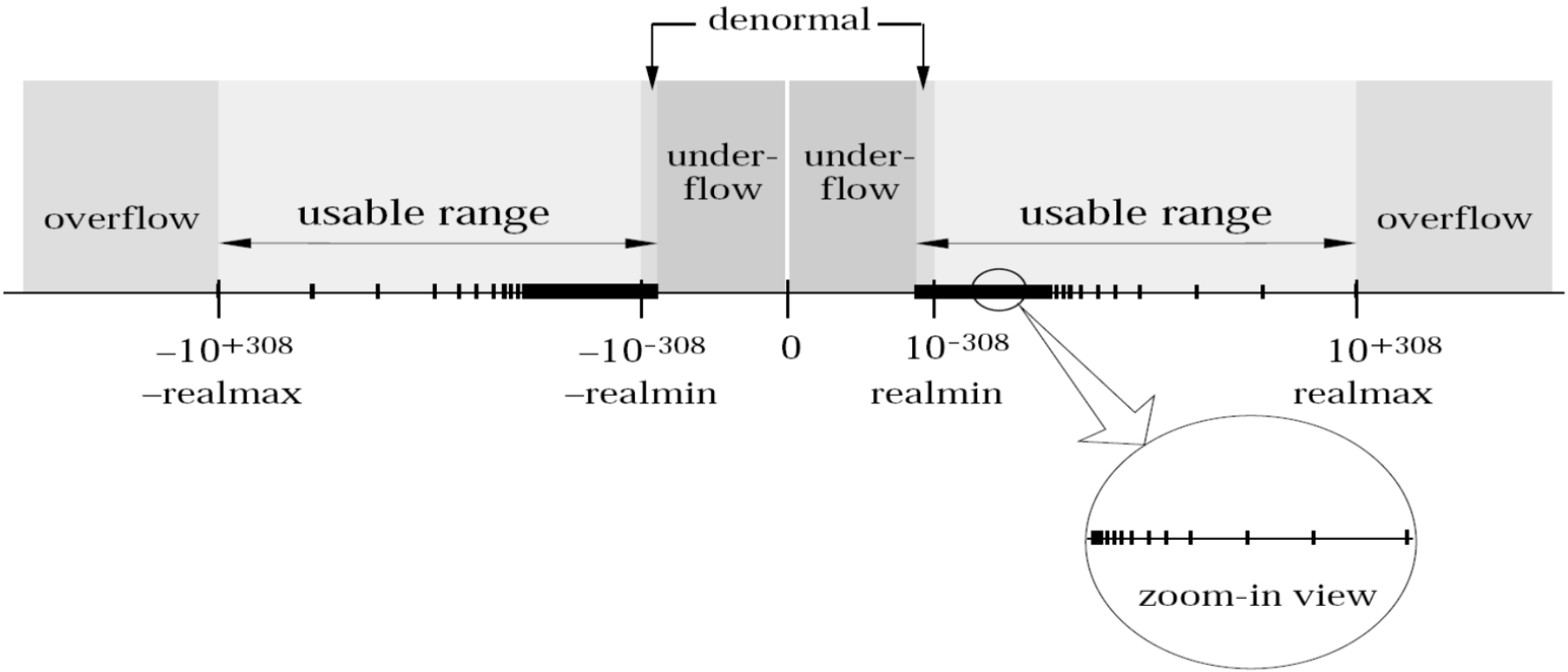
类型所占机器位数与特定编译器平台相关：sizeof

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

科学计数法

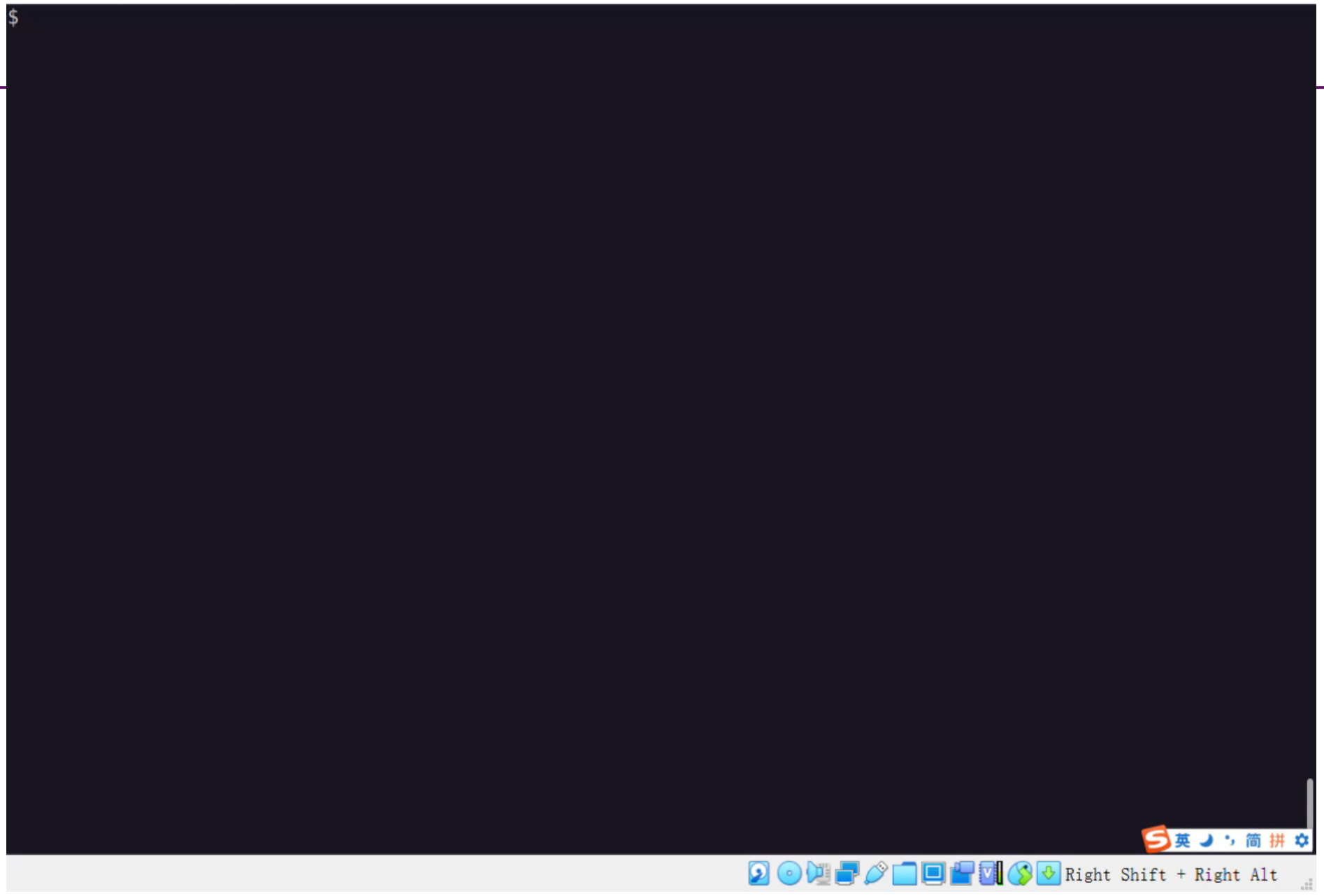


Floating Point Number Line



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

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)

- 比较
  - a == b 需求谨慎判断（要假设自带ε）
  - 浮点数：$(a+b)+c \neq a+(b+c)$
- 非规格化数（Exponenent == 0）
  - $x = (-1)^S \times (0.F) \times 2^{-126}$
- 零
  - +0.0,-0.0的Sbit是不一样的，但+0.0 == -0.0
- Inf/NaN (Not a Number)
  - Inf: 浮点数溢出很常见，不应该作为undefined behavior
  - NaN(0.0/0.0): 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

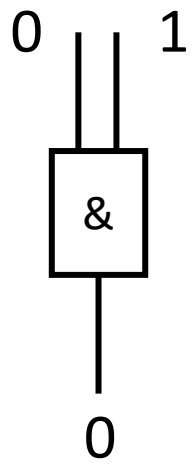
强制类型转换

- 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

一点bonus知识
位运算

为什么会有位运算

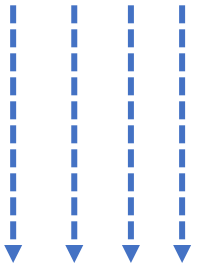
- 逻辑门和导线是构成计算机 (组合逻辑电路) 的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)



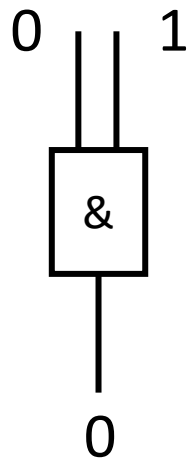
位运算与逻辑电路密不可分

a -> 0 1 0 1　　5

b -> 0 1 1 0　　10



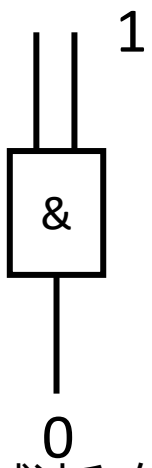
a & b -> 0 1 0 0　　8



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

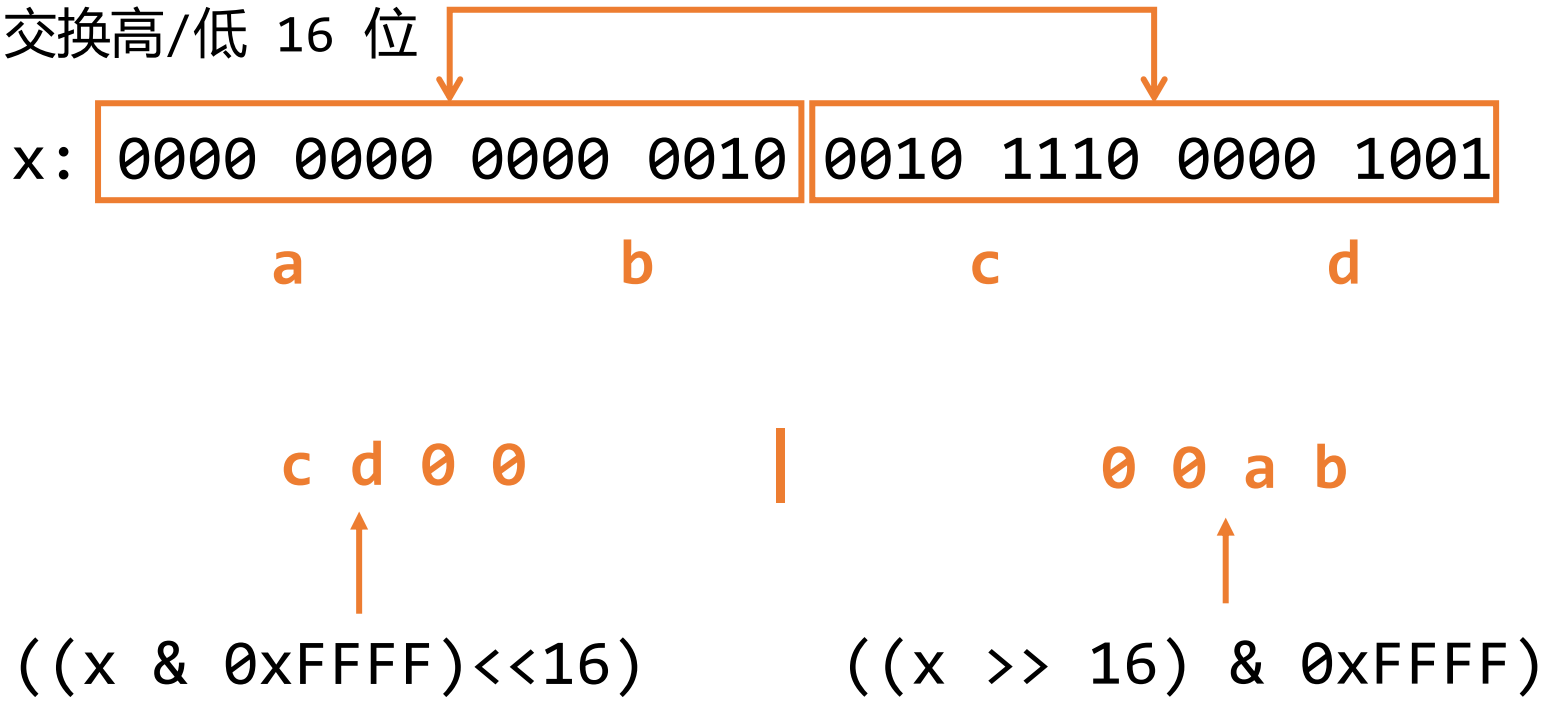
(x>>16)&0xFF

x&0xFF

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
  - 交换高/低 16 位



单指令多数据

`&, |, ~, ...` 对于整数里的每一个 bit 来说是独立（并行）的

- 如果我们操作的对象刚好每一个 bit 是独立的
  - 我们在一条指令里就实现了多个操作
  - SIMD (Single Instruction, Multiple Data)

- 例子：Bit Set
  - 32-bit 整数 $x \to S \subseteq \{0, 1, 2, 3, \ldots, 31\}$
  - 位运算是对所有 bit 同时完成的
    - C++ 中有 bitset，性能非常可观

`5 -> 0101`　　S: {0,2}

Bit Set: 基本操作

- 测试 $x \in S$
  - `(S >> x) & 1`

```
5 -> 0101 -> S: {0,2}
S
```

- 求 $S' = S \cup \{x\}$
  - `S | (1 << x)`

```
0101 | (0001 << 1)
0101 | 0010 -> 0111
```

- 更多习题
  - 求 $|S|$
  - 求 $S_1 \cup S_2$， $S_1 \cap S_2$
  - 求 $S_1 \setminus S_2$
  - 遍历 S中的所有元素（foreach）

```
S1: {0,2}, S2: {1,2}
   0101    0110
   |: 0111
   &: 0100
S1 \ S2: 0001
```

0	0	0
1	1	0
0	1	0
1	0	1

`S1 & (~S2)`

Undefined Behavior: 位运算相关

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

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

Bit Set: 求 $|S|$ (S二进制表示有多少个1)

- 测试 $x \in S$
  - `(S >> x) & 1`

`5 -> 0101 -> S: {0,2}`

```
int bitset_size(uint32_t S) {
   int n = 0;
   for (int i = 0; i < 32; i++) {
      n += bitset_contains(S, i);
   }
   return n;
}
```

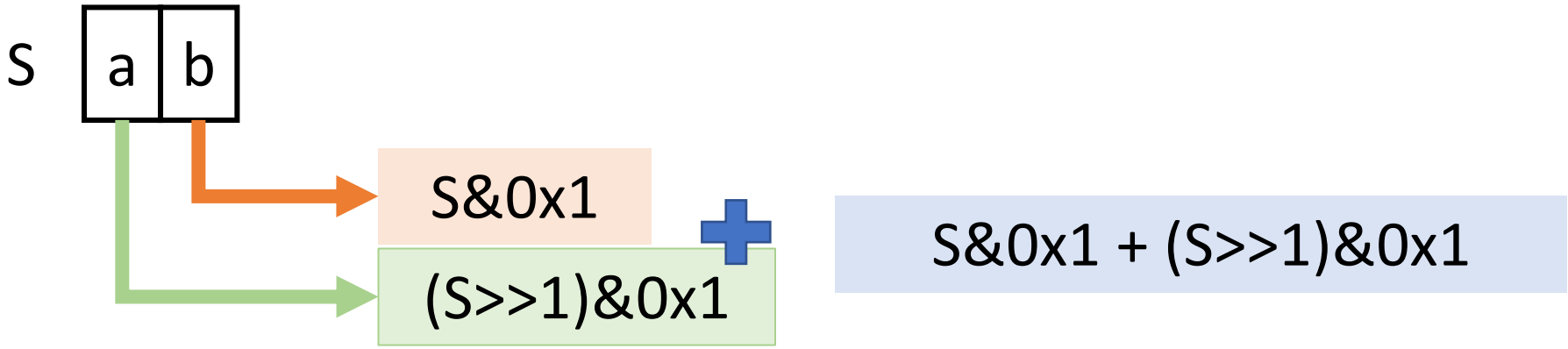
Bit Set: 求 $|S|$ (S二进制表示有多少个1)

```
int bitset_size(uint32_t S) {
   int n;
   for (int i = 0; i < 32; i++) {
      n += bitset_contains(S, i);
   }
   return n;
}
```

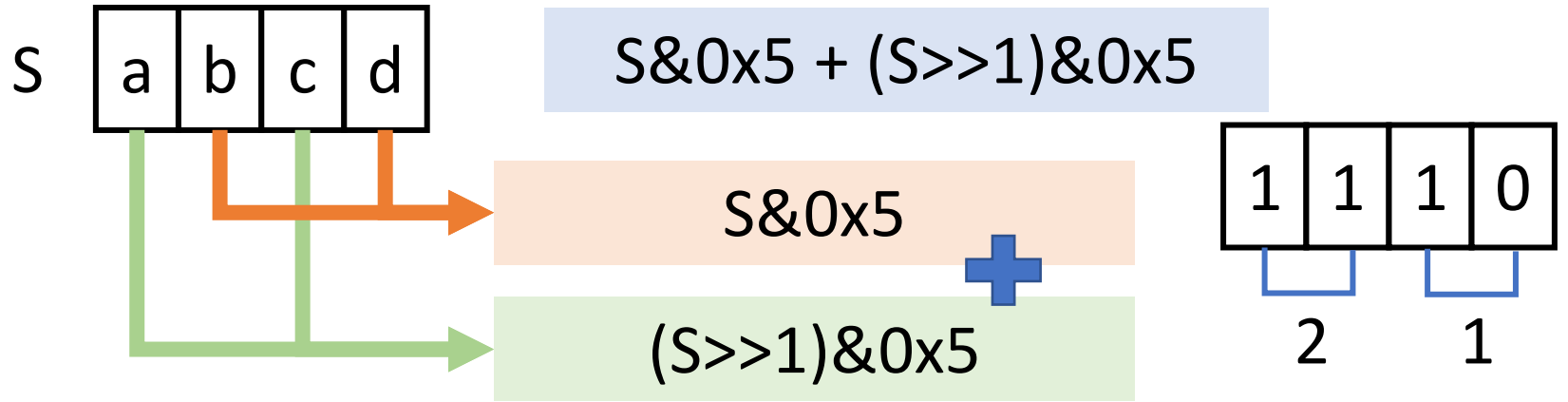
```
int bitset_size1(uint32_t S) { // SIMD
   S = (S & 0x55555555) + ((S >> 1) & 0x55555555);
   S = (S & 0x33333333) + ((S >> 2) & 0x33333333);
   S = (S & 0x0F0F0F0F) + ((S >> 4) & 0x0F0F0F0F);
   S = (S & 0x00FF00FF) + ((S >> 8) & 0x00FF00FF);
   S = (S & 0x0000FFFF) + ((S >> 16) & 0x0000FFFF);
   return S;
}
```

Bit Set: 求 $|S|$ (S二进制表示有多少个1)

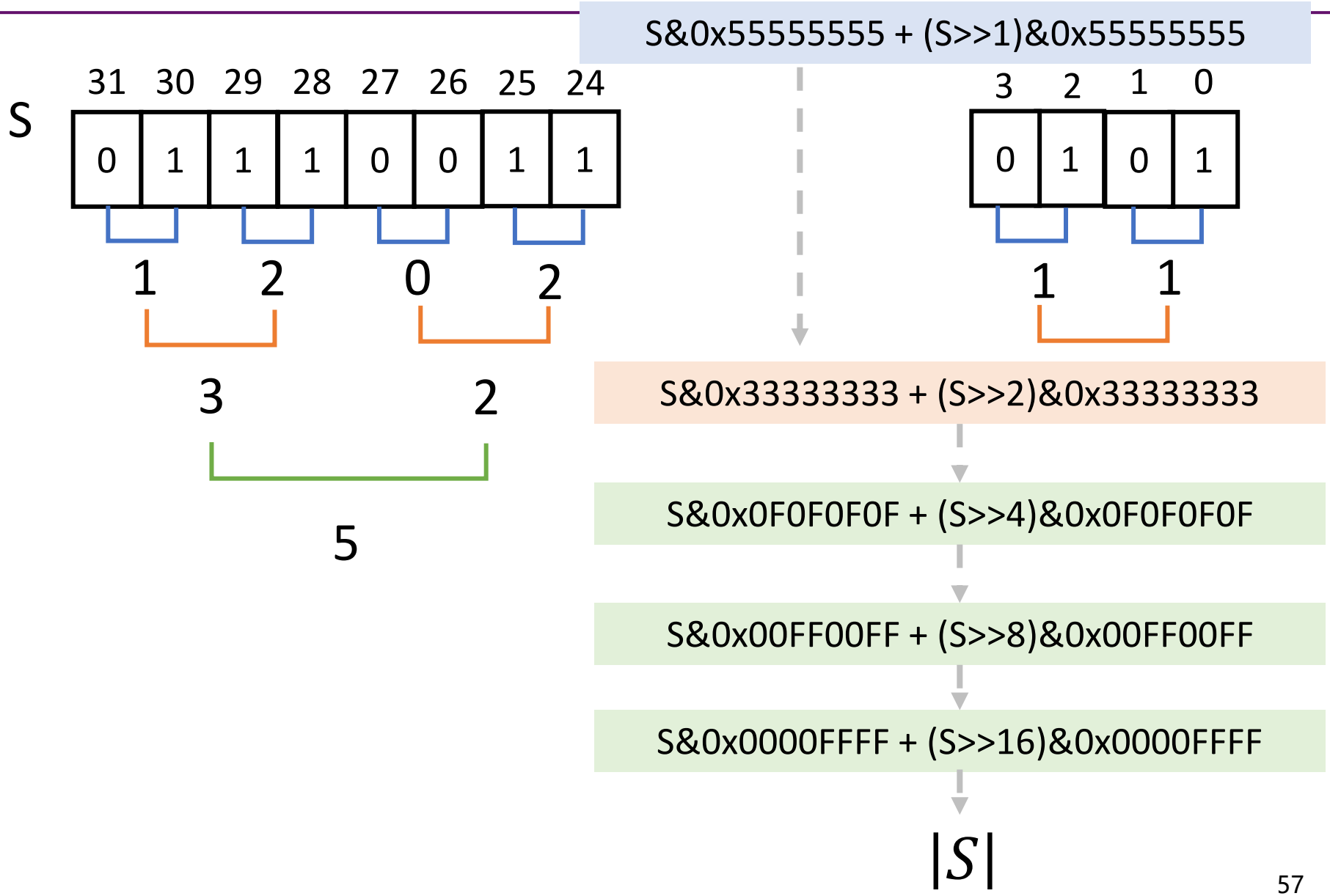
- S: 0b10 → $S = \{1\}$



- S: 0b1110 → $S = \{1,3\}$



Bit Set: 求 $|S|$ (S二进制表示有多少个1)



Bit Set: 求 $|S|$ (S二进制表示有多少个1)

```
int bitset_size(uint32_t S) {
   int n;
   for (int i = 0; i < 32; i++) {
      n += bitset_contains(S, i);
   }
   return n;
}
```

```
int bitset_size1(uint32_t S) { // SIMD
   S = (S & 0x55555555) + ((S >> 1) & 0x55555555);
   S = (S & 0x33333333) + ((S >> 2) & 0x33333333);
   S = (S & 0x0F0F0F0F) + ((S >> 4) & 0x0F0F0F0F);
   S = (S & 0x00FF00FF) + ((S >> 8) & 0x00FF00FF);
   S = (S & 0x0000FFFF) + ((S >> 16) & 0x0000FFFF);
   return S;
}
```

遍历 S中的元素

```
x=5 -> 0101
```

0101：x^(1<<0)

0100：x^(1<<2)

- Lowbit：找到最右边的1
  - x^lowbit(x)

  - **x & -x**
    - **这是怎么来的?**

Bit Set: 返回 $S \neq \emptyset$ 中的某个元素

- 有二进制数x = 0b+++++100，我们希望得到最后那个100
  - 想法：使用基本操作构造一些结果，能把+++++的部分给抵消掉

表达式	结果
x	0b+++++100
x-1	0b+++++011
~x	0b-----011
~x+1	0b-----100

- 一些有趣的式子：
  - x & (x-1) → 0b+++++000; x ^ (x-1) → 0b00000111;
  - x & (~x+1) → 0b00000100 (lowbit)
    - **x & -x**, (~x & (x-1))+ 1 都可以实现lowbit
    - 只遍历存在的元素可以加速求$|S|$

Lowbit: x&-x

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

- 带符号整数
  - 原码表示法：最高位为符号位
  - 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

```
x  = 0b+++++100
-x = 0b-----011+0b1 = 0b-----100
```

x&-x：0b00000100 → lowbit

Bit Set：求$\lfloor \log_2(x) \rfloor$

- 等同于31−$\mathrm{clz}(x)$

```
int clz(uint32_t x) {
    int n = 0;
    if (x <= 0x0000ffff) n += 16, x <<= 16;
    if (x <= 0x00ffffff) n += 8, x <<= 8;
    if (x <= 0x0fffffff) n += 4, x <<= 4;
    if (x <= 0x3fffffff) n += 2, x <<= 2;
    if (x <= 0x7fffffff) n ++;
    return n;
}
```

x：0x00000001	16
x：0x00010000	24
x：0x01000000	28
x：0x03000000	30
$S = \{0\}$	31

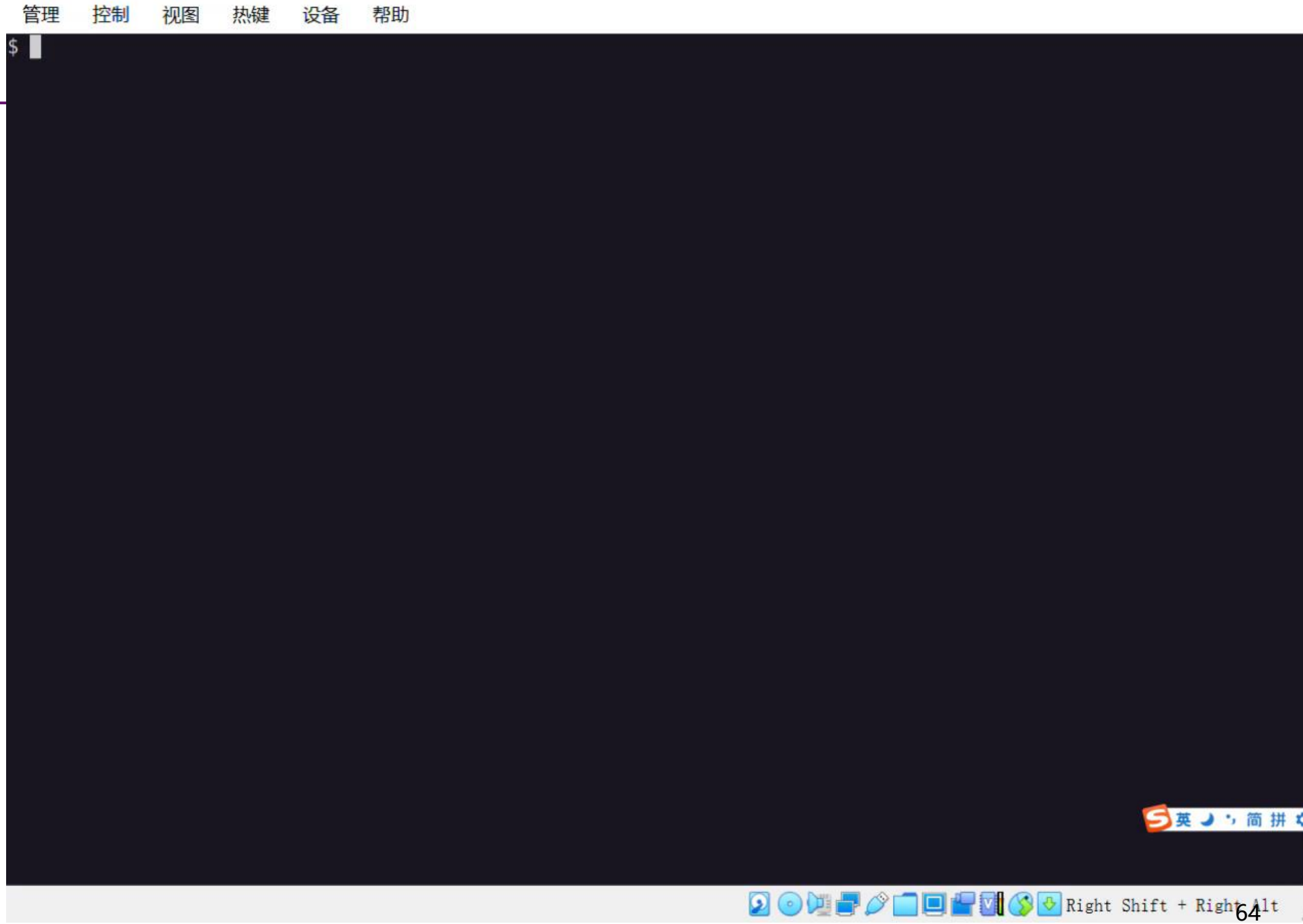
- （奇怪的代码）假设x是lowbit的结果？

```
#define LOG2(x) \
    ("-01J2GK-3@HNL;-=47A-IFO?M:<6-E>95D8CB"[(x) % 37] - '0')
```

Bit Set：求$\lfloor \log_2(x) \rfloor$ (cont'd)

- 用一点点元编程（meta-programming）；试一试log2.c

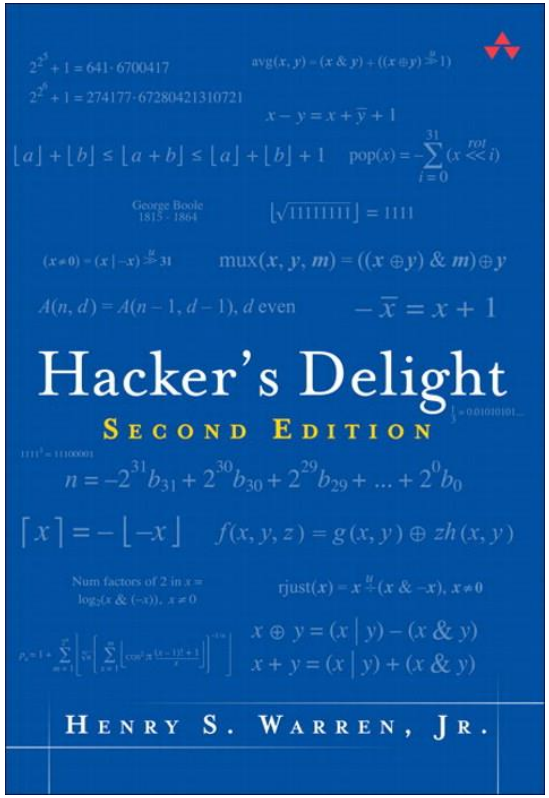
```
import json

n, base = 64, '0'
for m in range(n, 10000):
    if len({ (2**i) % m for i in range(n) }) == n:
        M = { j: chr(ord(base) + i)
              for j in range(0, m)
                  for i in range(0, n)
                      if (2**i) % m == j }
        break

magic = json.dumps(''.join(
    [ M.get(j, '-') for j in range(0, m) ]
    )).strip('"')

print(f'#define LOG2(x) ("{magic}"[(x) % {m}] - \'{base}\')')
```


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

一本有趣的参考书

Henry S. Warren, Jr. *Hacker's Delight* (2ed), Addison-Wesley, 2012.

- 让你理解写的更快的代码并不是“瞎猜”
  - 主要内容是各种数学（带来的代码优化）
  - 官方网站：hackersdelight.org
  - 见识一下真正的“奇技淫巧”



tictactoe

- tictactoe.c
- chess.c

三只小猪 (tictactoe.c)

100ms | 256000KiB | 130/500 = 26.0%

End.

IEEE754：异常复杂

- 除了$x = (-1)^S \times (1.F) \times 2^{E-B}$，还要考虑
  - 非规格化数, +0.0/-0.0, Inf, NaN
    - 一度引起了硬件厂商的众怒 (碰到非规格数干脆软件模拟吧)
    - 很多“对浮点数精度要求不高”硬件厂商选择不兼容 IEEE 754 (比如各种 GPU 制造商)
    - Nvidia 从 Fermi 才开始完整支持 IEEE754 (2010)

An interview with the old man of floating-point.
Reminiscences elicited from William Kahan by Charles Severance.

例子：计算$\frac{-b-\sqrt{b^2-4ac}}{2a}$

- 如果考虑比较极端的数值条件?
  - 消去误差： $-b$ v.s. $\sqrt{b^2-4ac}$ (catastrophic cancellation)
    - 还记得 x + 1.0 == x的例子吗
  - 溢出： b^2

- 一个更好的一元二次方程求根公式
  - $\frac{(-b)^2-\left(\sqrt{b^2-4ac}\right)^2}{(-b)+\sqrt{b^2-4ac}}/2a = \frac{4ac}{(-b)+\sqrt{b^2-4ac}}/2a \ (b<0)$
  - $\left(-b-\sqrt{b^2-4ac}\right)\cdot\frac{1}{2a} \ (0 \le b \le 10^{127})$
  - $-\frac{b}{a}+\frac{c}{b} \ (b > 10^{127})$
    - P. Panchekha, et al. Automatically improving accuracy for floating point expressions. In *Proc. of PLDI*, 2015.

凭什么?

It looked pretty complicated. On the other hand, we had a rationale for everything. -- William Kahan, 1989 ACM Turing Award Winner for his *fundamental contributions to numerical analysis*.

浮点数可以直接当成整数比较大小

+0.0/-0.0和Inf保证$(1/x)/x$不会发生sign shift

……

- IEEE754天才的设计保证了数值计算的稳定
  - 如果想了解IEEE754，请阅读D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.