浮点数

Floating

本章内容

Topic

- 编码

  Encoding

- 运算

  Operations

思考
Thoughts

- **Example 1**: Is $x^2 >= 0$?
  - ✓ Float's : Yes!
  - X Int's: No!

    40000 * 40000 = 1600000000

    50000 * 50000 = ?

- **Example 2**: Is (x + y) + z = x + (y + z)?
  - ✓ Unsigned & Signed Int's: Yes!
  - X Float's:

    (1e20 + (-1e20)) + 3.14 → 3.14

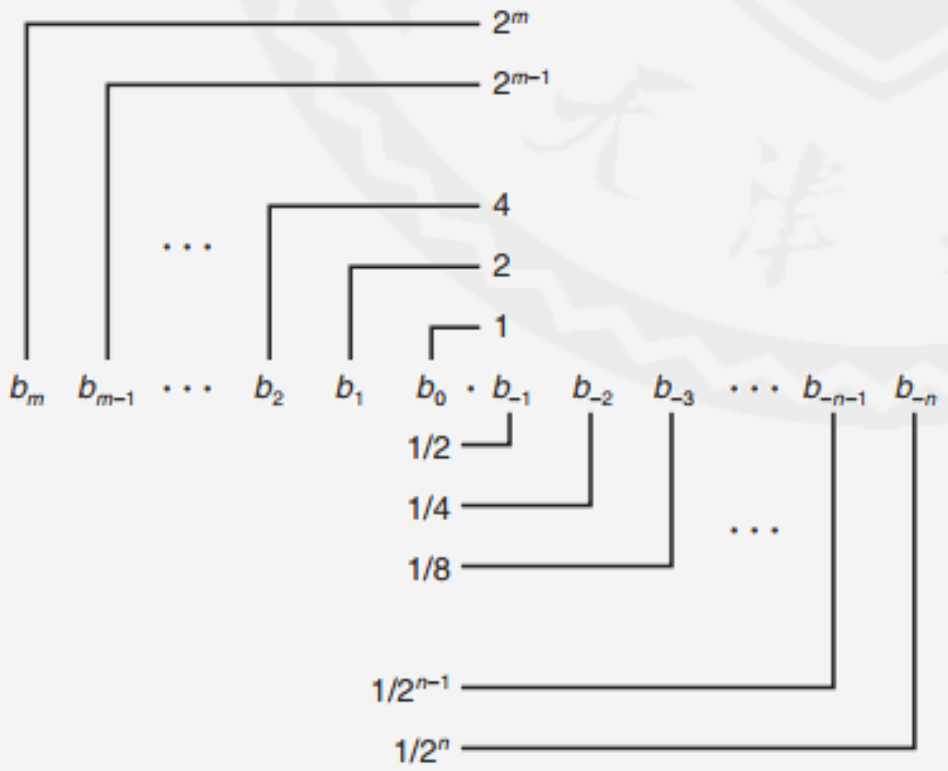
    1e20 + (-1e20 + 3.14) → ??

小数的二进制表示
Fractional Binary Representation

- 小数点左侧每一位的权重为2^i，右侧为 $1/2^i$.（位置计数法）

  Digits to the left of the binary point have weights of the form 2^i, while those to the right have weights of the form $1/2^i$.

$$b = \sum_{i=-n}^{m} 2^i \times b_i$$



举例：小数的二进制表示
Fractional Binary Representation：Examples

值 Value	二进制 Binary
$5\frac{3}{4}$	101.11_2
$2\frac{7}{8}$	010.111_2
$1\frac{7}{16}$	001.0111_2

- 观察

  Observations

  - 除以2可通过逻辑右移实现

    Divide by 2 by shifting right(unsigned)
  - 乘以2可通过左移实现

    Multiply by 2 by shifting left
  - 数字形如 $0.11111\cdots\cdots_2$ 是一个十分接近但小于1的数字

    Numbers of form $0.11111\cdots\cdots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \cdots\cdots 1/2^i \rightarrow 1.0$
    - 使用 $1.0 - \varepsilon$ 表示

      Use notation $1.0 - \varepsilon$

可表示数字的局限性
Representable Numbers of Limitation

值 Values	二进制 Binary
1/3	$0.0101010101[01]\cdots_2$
1/5	$0.001100110011[0011]\cdots_2$
1/10	$0.0001100110011[0011]\cdots_2$

- 只能精确地表示 $x/2^k$ 这种形式的数字

  Can only exactly represent numbers of the form $x/2^k$

  - 其他的数字在表示时都会出现循环节

    Other rational numbers have repeating bit representations

- 小数点只能设置在w比特范围内（受位宽的约束）

  Just one setting of decimal point within the *w* bits

  - 限制了表示数字的范围（极大数，极小数）

    Limited range of numbers (very small values? very large?)

编码有理数
Encoding Rational Numbers

- 采用 $v = x \times 2^y$ 的方式
  Form $v = x \times 2^y$
- 在 $|v| \gg 0$ 和 $|v| \ll 1$ 时，这种表示方法十分有效
  Very useful when $|v| \gg 0$ or $|v| \ll 1$
- 这是对数学上有理数的近似表示
  An Approximation to real arithmetic
- 从程序员的角度上看
  From programmer's perspective
  - 无趣、神秘且难以理解
    Uninteresting、arcane and incomprehensive

发展历史及编码特点
Encoding Rational Numbers

- 在1980年以前，存在着许多特殊的编码格式
  Until 1980s, there are many idiosyncratic formats
- IEEE 754 规范

  IEEE Standard 754
  - 1985年建立了统一浮点数运算统一标准

    Established in 1985 as uniform standard for floating point arithmetic
  - 由W. Kahan为Intel处理器设计

    Designed by W. Kahan for Intel processors
  - 主流处理器都支持

    Supported by all major CPUs

- 编码在设计上充分的考虑的数字表示的问题

  Driven by numerical concerns
  - 优雅的处理了取整, 上溢出和下溢出的问题

    Nice standards for rounding, overflow, underflow
  - 编码能够尽最大限度的发挥硬件的运算性能

    Try hard to make fast in hardware
  - 在标准定义时，数值分析人员相比硬件设计人员占据更多的主导地位

    Numerical analysts predominated over hardware designers in defining standard

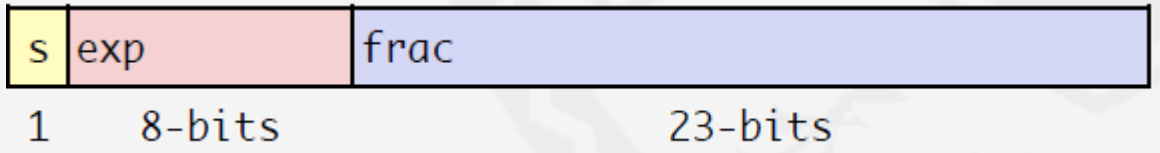
浮点数的表示
Floating Point Representation

- **数字的表示形式**
  Numerical Form $(-1)^s \times \mathrm{M} \times 2^E$
  - 符号位 s：决定了数字的正负
    Sign bit *s* determines whether number is negative or positive
  - 尾数 M：一个小数，取值为[1.0,2.0) 或 [0.0, 1.0)
    Significand *M* normally a fractional value in range [1.0,2.0) or [0.0, 1.0)
  - 阶码 E：2的E次幂
    Exponent *E* weights values by power of two
- **编码方式：**
  Encoding：
  - 最高位是符号位 s
    MSB s is sign bit s
  - exp 编码后得到 E （exp不等于e）
    exp field encodes E (but is not equal to E)
  - frac 编码后得到 M （frac不等于M）
    frac field encodes M (but is not equal to M)

几种精度的浮点数
Precision options

- 单精度：32位

  Single precision: 32 bits

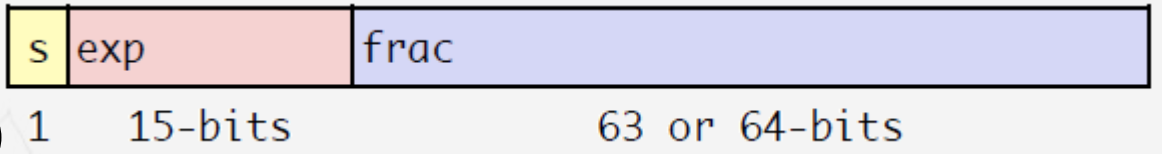


- 双精度：64位

  Double precision: 64 bits

- 扩展精度：80位（仅Intel支持）

  Extended precision: 80 bits (Intel only)



IEEE 754 标准
IEEE 754 Standard

- 规格化数

  Normalized Values

- 非规格化数

  Denormalized Values

- 特殊值

  Special Values

编码

Encoding

规格化数
Normalized Values

$(-1)^s \times \mathrm{M} \times 2^E$

- 当 exp ≠ 000…0 且 exp ≠ 111…1 时
  When: exp ≠ 000…0 and exp ≠ 111…1
- 阶码为指数减去一个常数（偏置）：
  Exponent coded as a biased value :

  E = Exp – Bias

  - Exp：exp域无符号数编码值
    Exp: unsigned value exp
  - bias = $2^{k-1} - 1$, k是exp的位宽
    bias = $2^{k-1} - 1$, where k is number of exponent bits
    - 单精度 127 (Exp: 1…254, E: -126…127)
      Single precision:
    - 双精度 1023 (Exp: 1…2046, E: -1022…1023)
      Double precision:

- 尾数编码里面包含一个隐式前置的1
  Significand coded with implied leading 1

  $M = 1.xxx\cdots x_2$

  - xxx…x: 为frac域的各位的编码
    xxx…x: bits of frac
  - 最小值：frac = 000…0 (M = 1.0)
    Minimum when frac = 000…0 (M = 1.0)
  - 最大值：frac = 111…1 (M = 2.0 - ε)
    Maximum when frac = 111…1 (M = 2.0 - ε)
  - 隐式前置的整数1始终存在，因此在frac中不需要占用额外的位来表示
    Get extra leading bit for “free”

举例：规格化数编码
Normalized Encoding Example

- 值 Value: float f = 15213.0

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \times 2^{13}$

- 尾数 Significand:

  frac = **1101101101101**0000000000$_2$

  M = 1.**1101101101101**$_2$

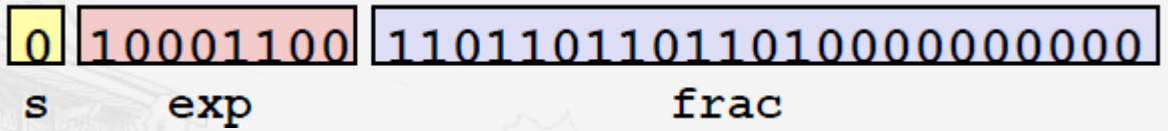
- 阶码 Exponent:

  E = 13

  Bias = 127

  Exp = 140 = 10001100_2

- 结果 Result



非规格化数
Denormalized Values

- 当 exp = 000…0 时
  When exp = 000…0
- 阶码
  Exponent value: E = - Bias + 1
  (insteads of E = 0 - Bias)
- 尾数编码内包含一个隐式前置的0
  Significand coded with implied leading 0
  $M = 0.xxx\cdots x_2$
  - xxx…x: 为frac域的各位的编码
    xxx…x: bits of frac

- 当 exp = 000…0, 且 frac = 000…0 时
  Case: exp = 000…0, frac = 000…0
  - 表示 0
    Represents zero value
  - 注意区别：+0 和 -0 （为什么？）
    Note distinct values: +0 and -0 (why?)
- 当 exp = 000…0, 且 frac ≠ 000…0 时
  Case: exp = 000…0, frac ≠ 000…0
  - 非常接近于0.0的数字
    Numbers closest to 0.0
  - 等间距的
    Equispaced

特殊值
Special Values

- 当 exp = 111…1 时
  When exp = 111…1
- 当 exp = 111…1 且 frac = 000…0 时
  Case: exp = 111…1, frac = 000…0
  - 表示无穷 ∞
    Represent value ∞ (infinity)
  - 意味着运算出现了溢出
    Operation that overflows
  - 正向溢出或负向溢出
    Both positive and negative
  - E.g., 1.0/0.0 = -1.0/-0.0 = + ∞, 1.0 / -0.0 = - ∞

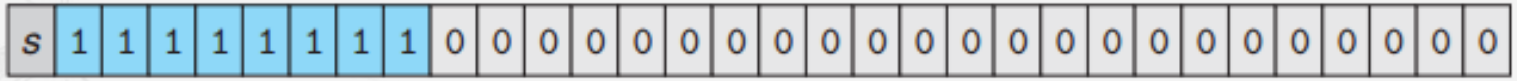
- 当 exp = 111…1 且 frac ≠ 000…0 时
  Case: exp = 111…1, frac ≠ 000…0
  - 不是一个数字
    Not-a-Number
    NaN
  - 表示数值无法确定
    Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}$, ∞ - ∞, ∞ × 0

单精度浮点数的类型
Categories of single-precision floating-point values

1. Normalized

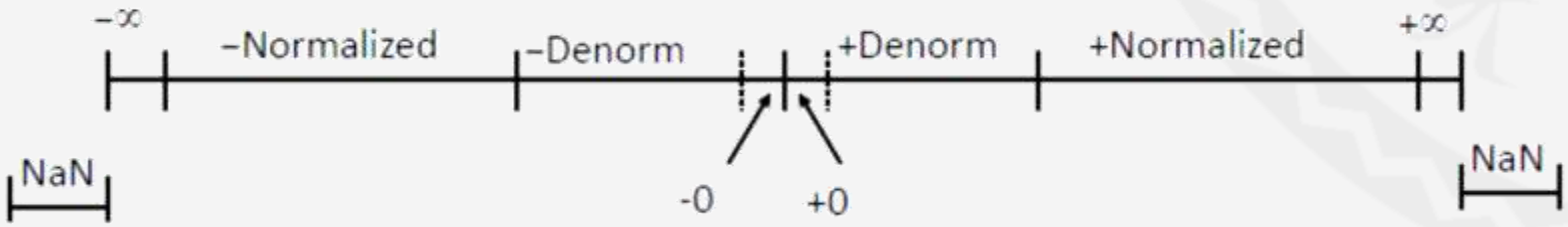
2. Denormalized

3a. Infinity



3b. NaN

浮点数编码的可视化表示
Visualization: Floating Point Encodings

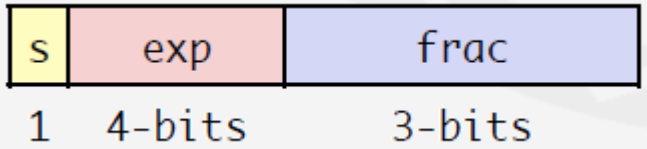


编码

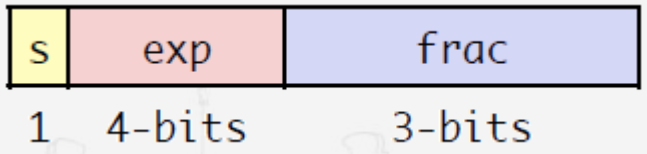
Encoding

举例：一个微型的浮点数编码系统
Tiny Floating Point Example

- 8位浮点数编码
  8-bit Floating Point Representation
  - 最高位为符号位
    The sign bit is in the most significant bit
  - 接下来是4位exp，偏置Bias为7
    The next four bits are the *exponent*, with a **Bias** of 7
  - 最后3位是frac
    The last three bits are the *frac*
- 与IEEE规范具有相同的形式
  Same general form as IEEE Format
  - 规格化数、非规格化数
    normalized, denormalized
  - 0、NaN和无穷的编码
    represenation of 0, NaN, infinity



动态的数字间隔（正数部分）
Dynamic Range (Positive Only)



	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

需要关注的数字
Interesting Numbers

Description	exp	frac	Single Precision		Double Precision	
			Value	Decimal	Value	Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times 2^{-126}$	1.4×10^{-45}	$2^{-52}\times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00\cdots01$	$0\cdots00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01\cdots11$	$0\cdots00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times 2^{127}$	3.4×10^{38}	$(2-\epsilon)\times 2^{1023}$	1.8×10^{308}

IEEE 754编码的特殊属性
Special Properties of the IEEE Encoding

- 浮点数 0 和 整数 0 编码相同
  FP Zero Same as Integer Zero
  - 所有位都为0
    All bits = 0
- 几乎可以用无符号整数比较的方法实现浮点数的比较运算
  Can (Almost) Use Unsigned Integer Comparison
  - 首先比较符号位
    Must first compare sign bits
- 考虑 -0 = 0 的问题
  Must consider -0 = 0
- NaN的问题
  Nans problematic
  - 比其他值都大
    Will be greater than any other values
- 其他的和无符号数比较方法类似
  Otherwise OK
  - 非规格化数与规格化数
    Denormalized vs. Normalized
  - 规格化数与无穷
    Normalized vs. Infinity

本章内容

Topic

- 编码

  Encoding

- 运算

  Operations

几种舍入模式
Round Modes

- 向下舍入
  Round down
  - 舍入结果接近但不会大于实际结果
    rounded result is close to but no greater than true result.
- 向上舍入
  Round up
  - 舍入结果接近但不会小于实际结果
    rounded result is close to but no less than true result.
- 向 0 舍入
  Round to Zero
  - 舍入结果向 0 的方向靠近
    rounded result is close to 0.
  - 如果为正数，舍入结果不大于实际结果
    if positive, round result is no greater than true result
  - 如果为负数，舍入结果不小于实际结果
    otherwise, no less than true result.

向偶数舍入
Round-to-Even

- 浮点数运算默认的舍入模式
  Default Rounding Mode
  - 其他的舍入模式都会统计偏差
    All others are statistically biased
    - 一组正数的总和将始终被高估或低估
      Sum of set of positive numbers will consistently be over- or under- estimated

比较
Comparison

Mode	1.40	1.60	1.50	2.50	-1.50
Round-to-Even	1	2	2	2	-2
Round-toward-zero	1	1	1	2	-1
Round-down	1	1	1	2	-2
Round-up	2	2	2	3	-1

向偶数舍入
Round-to-Even

- 适用于舍入至小数点后任何位置
  Applying to Other Decimal Places
  - 当数字正好处在四舍五入的中间时
    When exactly halfway between two possible values
    - 向最低位为偶数的方向舍入
      Round so that least significant digit is even

值 Value	向偶数舍入 Round-to-Even	说明 Descprition
1.2349999	1.23	Less than half way 比中间值小，四舍
1.2350001	1.24	Greater than half way 比中间值大，五入
1.2350000	1.24	Half way, round up 中间，向上舍入 （偶数方向）
1.2450000	1.24	Half way, round down 中间，向下舍入 （偶数方向）

二进制数的向偶数舍入方法
Rounding Binary Number

- 偶数方向意味着舍入后最后一位为 0
  “Even” when least significant bit is 0
- 中间意味着待舍入的部分为 $100\cdots_2$
  Half way when bits to right of rounding position = $100\cdots_2$

Value	Binary	Rounded	Action	Round Decimal
2 3/32	10.00011	10.00	Down	2
2 3/16	10.00111	10.01	Up	2 1/4
2 7/8	10.111	11.00	Up	3
2 5/8	10.101	10.10	Down	2 1/2

浮点数运算：基本思想
Floating Point Operations: Basic Idea

$$x +^f y = Round(x + y)$$

$$x \times^f y = Round(x \times y)$$

- 首先计算出精确的值
  First compute exact result
- 将结果调整至目标的精度
  Make it fit into desired precision
  - 如果阶码值过大，可能会导致溢出
    Possibly overflow if exponent to large
  - 可能会进行舍入以满足尾数的位宽
    Possibly round to fit into frac

浮点数乘法
Floating Point Multiplication

$(-1)^{s1}$ M1 2^{E1} × $(-1)^{s2}$ M2 2^{E2}

- 精确的结果 (Exact Result)：$(-1)^{s}$ M 2^{E}
  - 符号位 (Sign) s: s1^s2
  - 尾数 (Significand) M: M1 ×M2
  - 阶码 (Exponent) E: E1 + E2
- 修正 （Fixing）
  - 如果M ⩾ 2，右移M，并增大E的值
    If M ⩾ 2, shift M right, increment E
  - 如果E超出范围，溢出
    If E out of range, overflow
  - 对M进行舍入以满足frac的位宽精度要求
    Round M to fit frac precision
- 尾数相乘的实现细节十分繁琐
  Biggest chore is multiplying significands

浮点数加法
Floating Point Addition

$(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2}　　（假设E1 > E2 , Assume E1 > E2）

- 精确的结果 (Exact Result)：$(-1)^{s1}$ M 2^{E}
  - 符号位 s 和尾数 M：有符号数对齐后相加的结果
    Sign s, Significand M： Result of signed align & add
  - 阶码 (Exponent) E:　　E2
- 修正 （Fixing）
  - 如果M ⩾ 2，右移M，并增大E的值
    If M ⩾ 2, shift M right, increment E
  - 如果M < 1，将M左移k位，然后E减去k
    If M < 1, shift M left k positions, decrement E by k
  - 如果E超出范围，溢出
    If E out of range, overflow
  - 对M进行舍入以满足frac的位宽精度要求
    Round M to fit frac precision

C语言中的浮点数
Point in C

- C语言标准中支持两种精度的浮点数
  C Guarantees Two Levels
  - float　　单精度，single precision
  - double　　双精度，double precision
- 类型转换
  Conversions/Casting
  - 在 int、float和double间进行类型转换会改变编码（与有/无符号整数转换不同）
    Casting between int, float and double changes bit representation
- double/float → int
  - 截断尾数部分
    Truncates fractional part
  - 向0舍入
    Like rounding toward zero
  - 标准中没有定义越界和NaN的情况：通常是设置为T_{Min}和T_{Max}
    Not defined when out of range or NaN: Generally sets to T_{Min} or T_{Max}
- int → double
  - 精确转换，由于int的位宽小于53
    Exact conversion, as long as int has ⩽ 53 bit word size
- int → float
  - 将会浮点数舍入模式进行舍入
    Will round according to rounding mode

C语言中的浮点数
Point in C

```
int x = …;

float f = …;

double d = …;

// 假设 d 和 f 不是NaN和无穷
// Assume neither d nor f is NaN or infinity
```

x == (int)(float) x	No: 24 bit significand
x == (int)(double) x	Yes: 53 bit significand
f == (float)(double) f	Yes: increases precision
d == (float) d	No: loses precision
f == -(-f);	Yes: Just change sign bit
2/3 == 2/3.0	No: 2/3 == 0
d < 0.0 ⇒ ((d*2) < 0.0)	Yes!
d *d >= 0.0	Yes!
(d+f)-d == f	No: Not associative

运算

Operations

思考
Thoughts

- Why ?

0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1+ 0.1 ≠ 1

0.2 + 0.2 + 0.2 + 0.2 + 0.2 = 1

```
double d1, d2;
……
if (d1-d2==0)
{
    ……
}
```

- 准度和精度
  Accuracy & Precision