

# Floating-Point numbers

## Форма представления действительных чисел

- Числа с плавающей точкой: Floating-Point
- Числа с фиксированной точкой: Fixed-Point

Тип	размер в байтах			Приближенный диапазон и точность представления	
	x86-32	x86-64	ARM64		
float	4	4	4	$10^{\mp 38}$	7 дес. цифр
double	8	8	8	$10^{\mp 307}$	16 дес. цифр
long double	12	16	8	$10^{\mp 4931}$	19 дес. цифр

# Представление чисел с плавающей точкой

Структура числа:  $\pm(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \dots + d_p\beta^{-p}) \times \beta^E$

- $\beta$  – основание системы счисления:  $\beta = 2$  (или  $\beta = 10$ )
- Мантисса:  $\pm d_0.d_1d_2\dots d_p$ , где  $0 \leq d_i \leq \beta$ ,  $p$  – точность представления
- Порядок (или экспонента) числа:  $E$

Например для числа 0.1:

$$\beta = 10, p = 3 \quad 1.00 \times 10^{-1}$$

$$\beta = 2, p = 24 \quad 1.100110011001100110011001 \times 2^{-4}$$

## Нормализованные числа

- Для увеличения точности (числа значащих цифр) мантиссу хранят в диапазоне  $[1, \beta)$
- Для  $\beta = 2$  всегда  $d_0 = 1$ , поэтому хранится только дробная часть:  $d_1d_2\dots d_p$

# Особенности работы с FP числами

## Сложение и вычитание

- Поглощение значащих цифр малого числа

точное вычисление:  $123456.7 + 101.7654 = 123558.4654$

$$\begin{array}{rcll} & & \beta = 10, p = 7(\sim \text{float}) & \\ \hline 123456.7 & = & 1.234567 & \times 10^5 \\ 101.7654 & = & 0.001017654 & \times 10^5 \\ & & \text{-----} & \\ & & 1.235585 & \times 10^5 \quad (\text{округление}) \\ \hline \end{array}$$

- Катастрофическая потеря точности при вычитании

точное вычисление:  $123457.1467 - 123456.659 = 0.4877$

$$\begin{array}{rcll} & & \beta = 10, p = 7(\sim \text{float}) & \\ \hline 123457.1467 & = & 1.234571467 & \times 10^5 \\ 123456.659 & = & 1.234566659 & \times 10^5 \\ & & \text{-----} & \\ & & 0.000005 & \times 10^5 \quad (\text{округление}) \\ \hline \end{array}$$

## Умножение и деление

В этих операциях потери точности нет, только ошибка округления

точное вычисление:  $4734.612 \times 541724.2 = 2564853898.0104$

---

$\beta = 10, p = 7 (\sim \text{float})$

---

$$4734.612 = 4.734612 \times 10^3$$

$$541724.2 = 5.417242 \times 10^5$$

-----

$$25.64854 \times 10^8 \quad (\text{округление})$$

# Floating point $\neq$ Real

Нарушается ассоциативность:  $(a + b) + c \neq a + (b + c)$

$$123456.7 + 0.08 + 0.03 = 123456.81$$

$$123456.7 = 1.234567 \times 10^5$$

$$0.08 = 0.000000 \times 10^5$$

$$\begin{array}{r} \text{-----} \\ 1.234567 \times 10^5 \end{array}$$

$$1.234567 \times 10^5$$

$$0.03 = 0.000000 \times 10^5$$

$$\begin{array}{r} \text{-----} \\ 1.234567 \times 10^5 \end{array}$$

$$0.08 = 8.000000 \times 10^{-2}$$

$$0.03 = 3.000000 \times 10^{-2}$$

$$\begin{array}{r} \text{-----} \\ 1.100000 \times 10^{-1} \end{array}$$

$$123456.7 = 1.234567 \times 10^5$$

$$0.11 = 0.000001 \times 10^5$$

$$\begin{array}{r} \text{-----} \\ 1.234568 \times 10^5 \end{array}$$

Нарушается распределительный закон:  $(a + b) \times c \neq a \times c + b \times c$

$$(123456.7 + 0.08) \times 2 \text{ и } 246913.4 + 0.16$$

# Стандарт IEEE 754

## Бинарные базовые типы IEEE 754

	Знак	Экспонента*	Дробная часть мантииссы
binary16	1-bit	5-bits	10-bits
binary32	1-bit	8-bits	23-bits
binary64	1-bit	11-bits	52-bits
binary128	1-bit	15-bits	112-bits
binary256	1-bit	19-bits	236-bits

\* В экспоненте используется представление целых чисел *excess-K* с  $K = 2^{(n-1)} - 1$ , где  $n$  – число бит в поле экспоненты

# Специальные числа в IEEE 754

- 1 **Ноль (0)** – нулевые значения и в поле экспоненты и в поле дробной части  
☞ существует как  $+0$ , так  $-0$
- 2 **Денормализованные числа** – нули в поле экспоненты и в этом случае считают  $d_0 = 0$ :  
$$(-1)^s \times 0.d_1 d_2 \dots d_{52} \times 2^{-1022}$$
- 3 **Бесконечность ( $\text{Inf}, \infty$ )** – единицы в поле экспоненты и нули в дробной части, существует  $+\infty$  и  $-\infty$
- 4 **NaN (not a number)** – единицы в поле экспоненты и ненулевая мантисса, знак NaN не имеет значения  
☞ сравнение с NaN «неупорядоченно»

## Операции со специальными числами в IEEE 754

операция			результат
Num	/	$\pm\infty$	0
Num	/	0	$\pm\infty$
$\pm 0$	/	$\pm 0$	NaN
$\pm\infty$	/	$\pm\infty$	NaN
$\pm\infty$	$\times$	$\pm\infty$	$\pm\infty$
$\pm\infty$	$\times$	$\pm 0$	NaN
$\infty$	$+$	$\infty$	$\infty$
$\infty$	$-$	$\infty$	NaN
$-0$	$==$	$+0$	True
NaN	comp	Any	False
NaN	$!=$	Any	True

# Rounding modes in IEEE 754

## Концепция режима округления

☞ Способ округления результата вычисления до конечного (или, возможно, бесконечного) числа с плавающей точкой

- Округление к нулю (truncation)
- Округление к  $+\infty$  (ceiling)
- Округление к  $-\infty$  (floor)
- Округление к ближайшему: возможен конфликт если значение попадает точно посередине между двумя последовательными числами с плавающей точкой

## Fused Multiply-Add (FMA) in IEEE 754-2008

- Совмещенная операция умножения со сложением  $a + b \times c$ : одно округление вместо двух и ускоряет и повышает точность вычислений



# Соответствие FP в языках и стандарта IEEE 754

## Базовые форматы представлений IEEE 754

Имя	Мантисса	~точность <sub>10</sub>	E-min	E-max	~E-max <sub>10</sub>
binary32	23	7.22	-126	+127	38.23
binary64	52	15.95	-1022	+1023	307.95
binary128	112	34.02	-16382	+16383	4931.77
<i>Intel 80x87 "co-processor"</i>					
80-bit	63	19.27	-16382	+16383	4931.77

## X86-32, X86-64 и ARM64

C и C++: `float`  $\Rightarrow$  binary32

`double`  $\Rightarrow$  binary64

`long double`  $\Rightarrow$  80-bit для X86

$\Rightarrow$  binary64 для ARM64

Python3: `float` numbers  $\Rightarrow$  binary64

# Предельные значения для FP в C и C++

Заголовочный файл <float.h>: DBL\_, FLT\_, LDBL\_

```
printf("DBL_MIN          = %e\n", DBL_MIN);          // 2.225074e-308
printf("DBL_MAX          = %e\n", DBL_MAX);          // 1.797693e+308
printf("DBL_MIN_10_EXP   = %d\n", DBL_MIN_10_EXP);   // -307
printf("DBL_MAX_10_EXP   = %d\n", DBL_MAX_10_EXP);   // 308
printf("DBL_EPSILON      = %e\n", DBL_EPSILON);      // 2.220446e-16
```

C++ функции в классе шаблонов std::numeric\_limits

```
#include <limits>
using namespace std;
cout << numeric_limits<double>::min() << endl;      // 2.22507e-308
cout << numeric_limits<double>::max() << endl;      // 1.79769e+308
cout << numeric_limits<double>::lowest() << endl;   // -1.79769e+308
cout << numeric_limits<double>::denorm_min() << endl; // 4.94066e-324
cout << numeric_limits<double>::epsilon() << endl;  // 2.22045e-16
```

# Специальные числа в C и C++

Заголовочные файлы `<math.h>` для C и `<cmath>` для C++ (C99, C++11)

- Macro Constants:

`NAN, INFINITY`: NaN и  $\infty$  для `float`

`HUGE_VAL, HUGE_VALF, HUGE_VALL`:  $\infty$  для `double`, `float` и `long double`

- Функции возвращающие 'quiet NaN' с заданным стрингом

`double nan(const char* arg)`

`float nanf(const char* arg)`, `long double nanl(const char* arg)`

C++ функции в классе шаблонов `std::numeric_limits`

```
cout << numeric_limits<double>::infinity() << endl; // inf
cout << numeric_limits<double>::quiet_NaN() << endl; // nan
```

## Операции со специальными числами в C и C++

```
double one = +1, zero = 0;  
double p_inf = one/zero, m_inf = one/-zero, nan = zero/zero;
```

```
printf("one/zero=%+f,one/-zero=%+f,zero/zero=%+f\n",  
       one/zero,one/-zero,zero/zero);
```

```
Result> one/zero=+inf, one/-zero=-inf, zero/zero=-nan
```

```
printf("one/+inf=%+f,one/-inf=%+f,inf*zero=%+f\n",  
       one/p_inf,one/m_inf,p_inf*zero);
```

```
Result> one/+inf=+0.000000, one/-inf=-0.000000, inf*zero=-nan
```

```
printf("-inf+inf=%+f,-inf*+inf=%+f,+inf/+inf=%+f\n",  
       m_inf+p_inf,m_inf*p_inf,p_inf/p_inf);
```

```
Result> -inf+inf=-nan, -inf*+inf=-inf, +inf/+inf=-nan
```

```
printf("(%f==%f)=%d\n",zero,-zero,zero==zero);
```

```
Result> (0.000000==0.000000)=1
```

```
printf("(NaN>one)=%d, (NaN<=one)=%d, (NaN==one)=%d (NaN!=one)=%d\n",  
       nan>one,nan<=one,nan==one,nan!=one);
```

```
Result> (NaN>one)=0, (NaN<=one)=0, (NaN==one)=0 (NaN!=one)=1
```

## Функции для классификации FP чисел

(C99, C++11)

`fp` – анализируемое число

`int isnan(fp)` – возвращает ненулевое значение, если `fp = NAN`

`int isinf(fp)` – возвращает  $\pm 1$  если `fp =  $\pm$ INFINITY`

`int isfinite(fp)` – ненулевое значение, если `fp  $\neq$  {NAN; INFINITY}`

`int isnormal(fp)` – ненулевое значение, если `fp` нормализованное число

`int fpclassify(fp)` – классификатор, в зависимости от `fp` возвращает:  
`FP_INFINITY`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` или `FP_ZERO`

## Функция для проверки знакового бита `int signbit(fp)`

возвращает ненулевое значение если `fp` отрицательно

```
printf("signbit(+0.)= %d\n", signbit(+0.)); // 0
```

```
printf("signbit(-0.)= %d\n", signbit(-0.)); // 1
```

# Резюме

- Любые операции для чисел с плавающей точкой ведут к появлению ошибки округления, и в сложных вычислениях ошибки могут накапливаться
- Числа с плавающей точкой ограничены как в области нуля, так и в области больших чисел
- Вычисления не всегда возвращают числа, имеются специальные «значения» NaN, Inf
- Сравнение чисел с плавающей точкой допускает четыре взаимоисключающих отношения: меньше, равно, больше и неупорядоченно (unordered)

# Машинная точность $\epsilon$

Def:  $\epsilon$  — наименьшее положительное число такое, что  $1 + \epsilon \neq 1$

По смыслу,  $\epsilon$  — максимальная относительная ошибка представления ненулевого вещественного числа  $\left| \frac{Fp(x) - x}{x} \right| < \epsilon$

## Простая программа вычисления $\epsilon$

```
double eps() {  
    double one = 1, eps = one;  
    do {  
        eps /= 2;  
    } while( (one + eps) > one );  
    return eps*2;  
}
```

eps(double)	= 2.22045e-16	log <sub>2</sub> (eps) = -52
eps(float)	= 1.19209e-07	log <sub>2</sub> (eps) = -23
eps(long double)	= 1.0842e-19	log <sub>2</sub> (eps) = -63 for X86


# Сравнение чисел с плавающей точкой

Простое сравнение: `if( result == expectedResult ) {...}`

- Поведение нестабильно, зависит от архитектуры и компилятора
- Маловероятно, что сравнение «истинно»

```
double a = 2.34e-2; // floating-point in «scientific notation»
float  b = 2.34e-2F; // F for 'float' constant
if ( a==b ) {
    printf("a=%f is equal to b=%f\n",a,b);
} else {
    printf("a=%f is NOT equals b=%f\n",a,b);
}
```

Result> a=0.023400 is NOT equals b=0.023400

 Следует избегать сравнения чисел с плавающей точкой с помощью оператора `==`



## Тестовая программа на сравнение fp-чисел

```
double eps_m = eps(); // machine epsilon
double x = 0., y = 0.;
for(int i = 0; i < 10; i++) {
    x += 0.1;
    if( i%2 ) {
        y += 0.2;
        printf(" %19.17f %19.17f --> %3d %3d %3d\n",
            x,y, (x==y), IsEqualABS(x,y,eps_m), IsEqualREL(x,y,eps_m));
    }
}
```

## Output

0.2000000000000000001	0.2000000000000000001	-->	1	1	1
0.4000000000000000002	0.4000000000000000002	-->	1	1	1
0.599999999999999998	0.6000000000000000009	-->	0	1	1
0.799999999999999993	0.8000000000000000004	-->	0	1	1
0.999999999999999989	1.0000000000000000000	-->	0	1	1

## Абсолютная разница: сравнение с `epsilon`

```
int IsEqualABS(double x, double y, double epsilon) {  
    return fabs(x-y) < epsilon;  
}
```

## Достоинства и недостатки (pro et contra)

- **pros** Если диапазон значений  $x$  и  $y$  известен и ограничен, то эта проверка очень проста и эффективна
- **cons** Не работает если  $\varepsilon$  меньше, чем возможная разность  $|x - y|$ : например, если `epsilon < 0.01`, то для `float x = 12345.67` нет  $y$  «близких» к  $x$

## Относительная разница:

$$Rel(x, y) = \left| \frac{x - y}{f(x, y)} \right| < \varepsilon, \text{ где } f(x, y) = \begin{cases} \text{min}(|x|, |y|) \text{ или } \text{max}(|x|, |y|) \\ \text{или } (|x| + |y|)/2 \text{ или } \dots \end{cases}$$

## Функции сравнения для $f(x, y) = \min(|x|, |y|)$

```
int IsEqualREL(double x, double y, double epsilon) {  
    double ax = fabs(x);  
    double ay = fabs(y);  
    return fabs(x-y) < epsilon*((ax<ay) ? ax : ay);  
}
```

- **pros** Более общий способ сравнения чисел, работающий вне зависимости от абсолютных значений  $x, y$
- **cons** Плохо подходит для чисел близких к нулю, например: для  $x = -1e-9, y = +1e-9$  получим  $\frac{|x-y|}{\min(|x|, |y|)} = 2$  и совсем не работает для  $x = y = 0$

# Алгоритм суммирования Кахана (Kahan)

## Алгоритм для улучшения точности суммирования

```
double KahanSum(double V[], int Nv ) {  
    double sum = 0;  
    double c = 0; // compensation for lost low-order bits  
    for ( size_t i = 0; i < Nv; ++i ) {  
        double y = V[i] - c;  
        double t = sum + y;  
        c = (t-sum) - y; // algebraically, c should always be zero  
        sum = t;  
    }  
    return sum;  
}
```



Алгоритм выполняет суммирование с двумя накопителями:

**sum** содержит сумму, а **c** накапливает части не включенные в сумму

## Проверка алгоритма суммирования Кахана

```
double eps = DBL_EPSILON; // 2.22045e-16
double V[] = {1., eps/2, eps/2};
double sum = V[0] + V[1] + V[2];
printf("sum= %.17f\n",sum);    // 1.00000000000000000
double sumK = KahanSum(V,3);
printf("sumK= %.17f\n",sumK); // 1.000000000000000022
```

- Алгоритм Kahan'a не идеален и плохо работает если элемент больше суммы
- Имеются другие алгоритмы: улучшенный алгоритм Kahan–Babuška (Neumaier), 2Sum (Knut), Fast2Sum (Dekker) ...
- В python3.12 функция `math.fsum()` использует улучшенный алгоритм Кахана

# Математическая библиотека `<math.h>`

## Соглашения

- функции «с обычными именами» работают с `double`
- для `float` и `long double` используются функции с окончаниями `f` и `l`:  
`sin` → `sinf`, `sinl`
- углы задаются в радианах
- используются все соглашения стандарта IEEE 754

## В C++ рекомендуется использовать `<cmath>`

- 📖 Заголовочные файлы C++ включают перегрузку функций:  
`abs()` в C++ «универсальная функция» и для `int` ... и для `double` ...

## Константы (double)

M_E	Число $e$	M_PI	Число $\pi$	M_2_SQRTPI	$2/\sqrt{\pi}$
M_LOG2E	$\log_2(e)$	M_PI_2	$\pi/2$	M_SQRT2	$\sqrt{2}$
M_LOG10E	$\log_{10}(e)$	M_PI_4	$\pi/4$	M_SQRT1_2	$1/\sqrt{2}$
M_LN2	$\ln(2)$	M_1_PI	$1/\pi$		
M_LN10	$\ln(10)$	M_2_PI	$2/\pi$		

👉 В `gcc` имеется расширение этих констант для `long double`, к имени надо добавить «l»: `M_PI` → `M_PIl`

👉 Стандарт C99 не требует наличия этих констант в `math.h`

- В `gcc, clang` они доступны по умолчанию или надо определить:  
`#define _GNU_SOURCE`
- В `Microsoft Visual C++` что бы их использовать надо определить:  
`#define _USE_MATH_DEFINES`

👉 В **C Numerics library** можно найти больше информации о функциях математической библиотеки

## Функции округления

**ceil(x)** – округление до ближайшего большего целого числа

**floor(x)** – округление до ближайшего меньшего целого числа

**round(x)** – округление до ближайшего целого в сторону от нуля

**trunc(x)** – округление до ближайшего целого в сторону к нулю

**nearbyint(x)** – округление в соответствии с **fesetround()**

```
double x = 3.5;
printf("          %.1f  %.1f\n",x,-x);
printf("ceil      %.1f  %.1f\n",ceil(x),ceil(-x));
printf("floor     %.1f  %.1f\n",floor(x),floor(-x));
printf("round     %.1f  %.1f\n",round(x),round(-x));
printf("trunc     %.1f  %.1f\n",trunc(x),trunc(-x));
```

	3.5	-3.5
ceil	4.0	-3.0
floor	3.0	-4.0
round	4.0	-4.0
trunc	3.0	-3.0



# Полезные библиотеки и программы

## Библиотеки для вычислений с произвольной точностью

- **GMP – The GNU Multiple Precision Arithmetic Library**
  - Целые числа, рациональные числа и числа с плавающей точкой
  - Написана на C, а наиболее «тонкие» места на ассемблере
  - Размер целых чисел практически неограничен:  $2^{37}$ -bit для x86-64
- **MPFR – Multiple-Precision Floating-point with correct Rounding**

## Библиотека численных методов GSL — GNU Scientific Library

- Библиотека численных методов, написана на «чистом» C
- Большое число функций, алгоритмов ...
- Большое количество языков программирования в которых эта библиотека может быть использована

Дополнительные слайды

## Макросы в `#include <fenv.h>`

(C99, C++11)

- Задание окружения (environment) для вычислений с плавающей точкой
- Генерировать исключения в случае:  
`divide-by-zero, overflow, underflow, inexact, invalid`
- Моды округления: `fegetround()` и `fesetround()`

Mode		Test values			
Макрос	Пояснение	+11.5	+12.5	-11.5	-12.5
<code>FE_TONEAREST</code>	к ближайшему	+12.0	+12.0	-12.0	-12.0
<code>FE_TOWARDZERO</code>	к нулю	+11.0	+12.0	-11.0	-12.0
<code>FE_UPWARD</code>	к $+\infty$	+12.0	+13.0	-11.0	-12.0
<code>FE_DOWNWARD</code>	к $-\infty$	+11.0	+12.0	-12.0	-13.0

# Функции в `<math.h>`

- В «[cpreference](#)» можно найти больше информации.

## Тригонометрические функции

`sin(x)`, `cos(x)`, `tan(x)` – синус, косинус, тангенс

`asin(x)`, `acos(x)`, `atan(x)` – арксинус, арккосинус, арктангенс

`atan2(y,x)`  $\equiv$  *arctan*( $y/x$ ) и по знакам  $y$  и  $x$  определяет квадрант:  
возвращаемое значение лежит в диапазоне  $[-\pi, \pi]$

## Показательные и логарифмические функции

`pow(x,y)` – возведения  $x$  в степень  $y$ : ( $x^y \equiv e^{y \cdot \ln(x)}$ )

`sqrt(x)`, `cbrt(x)` – корни  $\sqrt{x}$  и  $\sqrt[3]{x}$

`exp(x)` – экспонента  $e^x$

`sinh(x)`, `cosh(x)`, `tanh(x)` – гиперболические функции

`log(x)`, `log10(x)`, `log2(x)` – логарифмы:  $\ln(x)$ ,  $\lg(x)$ ,  $\log_2(x)$

## Другие функции

`fabs(x)` – абсолютная величина:  $|x|$

`fmax(a,b)` `fmin(a,b)` – возвращает большее (меньшее) из  $a, b$

`erf(x)` `erfc(x)` – функции ошибок

`tgamma(x)` `lgamma(x)` – гамма-функция и натуральный логарифм от гамма-функции

внимание: `gamma(x) == lgamma(x)` – устаревшее имя

# Комплексные числа <complex.h> (C99)

👉 В C11 разрешено отсутствие <complex.h>: `__STDC_NO_COMPLEX__`

- `double complex;`
- `float complex;`
- `long double complex;`

**Макросы** для комплексного числа:  $I \equiv \_Complex\_I \equiv 0+1*i$

- макрос `I` предпочтителен, но может вызвать проблемы если уже есть переменная `I`
- можно отказаться от `I` и использовать «длинное имя» `_Complex_I`

```
#include <complex.h>
#undef I
```

👉 заметьте, что  $I*I$  – комплексное число  $(-1+0*i)$

## Функции (c - double complex)

`creal(c)`, `cimag(c)`, `cabs(c)`, `carg(c)` – базовые функции:  
действительная и мнимая части, абсолютная величина, аргумент  
`cexp(c)`, `clog(c)`, `csqrt(c)`, `cpow()` – показательные и  
логарифмические функции  
`csin(c)`, `ccos(c)`, `ctan(c)`, `casin(c)`, `cacos(c)`, `catan(c)` –  
тригонометрические функции

## Пример

```
#include <complex.h> // note: <math.h> will be included
#include <stdio.h>
int main() {
    double complex ca = 1 + I;
    double complex cb = cexp(ca);
    // there is no a specific format specifier for complex_t
    printf("%f + %f*i\n",creal(cb),cimag(cb)); // 1.468694 + 2.287355*i
}
```

## IEEE 754 floating-point test

<http://www.netlib.org/paranoia/>

Программа проверяющая на соответствие стандарту IEEE 754

## Fixed point maths library: libfixmath

<https://code.google.com/p/libfixmath/>

Библиотека для работы с числами с фиксированной точкой