Floating-Point numbers

Форма представления действительных чисел

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

Тип	размер в байтах x86-32	x86-64	ARM64	Приближенный диапазон и точность представления	
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: $\pm(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \cdots + d_p\beta^{-p}) \times \beta^E$

- β – основание системы счисления: $\beta = 2$ (или $\beta = 10$)
- Мантисса: $\pm d_0 . d_1 d_2 \ldots d_p$, где $0 \le d_i \le \beta$, p – точность представления
- Порядок (или экспонента) числа: E

Например для числа 0.1:

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

$\beta = 2, p = 24$ $\quad 1.10011001100110011001101 \times 2^{-4}$

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: $d_1 d_2 \ldots d_p$

Особенности работы с FP числами

Сложение и вычитание

- **Поглощение значащих цифр малого числа**
  точное вычисление: $123456.7 + 101.7654 = 123558.4654$

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

```
123456.7        = 1.234567     x 10^5
101.7654        = 0.001017654  x 10^5
                  ------------------
                  1.235585     x 10^5   (округление)
```

- **Катастрофическая потеря точности при вычитании**
  точное вычисление: $123457.1467 - 123456.659 = 0.4877$

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

```
123457.1467     = 1.234571467 x 10^5
123456.659      = 1.234566659 x 10^5
                  ----------------
                  0.000005 x 10^5   (округление)
```

Умножение и деление

В этих операциях потери точности нет, только ошибка округления

точное вычисление: $4734.612 \times 541724.2 = 2564853898.0104$

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

```
4734.612        = 4.734612 x 10^3
541724.2        = 5.417242 x 10^5
                  ---------------
                  25.648544 x 10^8  (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 x 10^5
0.08     = 0.000000 x 10^5
           ----------------
           1.234567 x 10^5

           1.234567 x 10^5
0.03     = 0.000000 x 10^5
           ----------------
           1.234567 x 10^5
```

```
0.08     = 8.000000 x 10^-2
0.03     = 3.000000 x 10^-2
           -----------------
           1.100000 x 10^-1

123456.7 = 1.234567 x 10^5
0.11     = 0.000001 x 10^5
           ----------------
           1.234568 x 10^5
```

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

$(123456.7 + 0.08) * 2$ и $246913.4 + 0.16$

Стандарт IEEE 754

Бинарные базовые типы IEEE 754

	Знак	Экспонента*	Дробная часть мантиссы
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

* Для целых чисел в показателе экспоненты используется представление *excess-K*, где $K = 2^{(n-1)} - 1$ (n – число бит в поле экспоненты), но значения со всеми нулями и всеми единицами зарезервированы для специальных чисел

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 $d_0 = 0$:
   $(-1)^s \times 0.d_1 d_2 \ldots d_{52} \times 2^{-1022}$
3. **Бесконечность (`Inf`,∞)** – 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	$\pm\infty$
± 0	/	± 0	NaN
$\pm\infty$	/	$\pm\infty$	NaN
$\pm\infty$	$\times$	$\pm\infty$	$\pm\infty$
$\pm\infty$	$\times$	± 0	NaN
∞	$+$	∞	∞
∞	$-$	∞	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, а если значение попадает точно посередине то конфликт разрешается округлением к нулевому наименьшему биту
  ☞ используется по умолчанию для бинарного представления

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$_{10}$	E-min	E-max	~E-max$_{10}$
binary32	23	7.22	−126	+127	38.23
binary64	52	15.95	−1022	+1023	307.95
binary128	112	34.02	−16382	+16383	4931.77
Intel 80x87 "co-processor"					
80-bit	63	19.27	−16382	+16383	4931.77

X86-32, X86-64 и ARM64

C и C++: `float` ⇒ binary32
`double` ⇒ binary64
`long double` ⇒ 80-bit для X86
⇒ binary64 для ARM64

Python3: `float` numbers ⇒ 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 C99 и `<cmath>` для C++11

- Macro Constants:
  `NAN`,`INFINITY`: NaN и ∞ для `float`
  `HUGE_VAL`,`HUGE_VALF`,`HUGE_VALL`: ∞ для `double`, `float` и `long double`
- Функции возвращающие ‘quiet NaN’:
  `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 в C99, C++11

`fp` – анализируемое число

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

`int isinf(fp)` – возвращает ± 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
```

Задание FP в шестнадцатеричном виде в C++17

Вид: 0x[integer-hex].[fractional-hex]p[exponent-of-two-decimal]

```
#define PRT(x) std::printf("%s = %f\n",#x,x); // macro for printing
PRT(0x1.0p10); // 0x1.0p10 = 1024.000000 (2^10)
PRT(0x8p-3);   // 0x8p-3 = 1.000000 (8*2^-3)
PRT(0x0.1p0);  // 0x0.1p0 = 0.062500 (1/16)
PRT(0x0.11p0); // 0x0.11p0 = 0.066406 (1/16+1/16^2)

// printing floating point numbers in hex format
std::cout << std::hexfloat << 3.141593 <<'\n'; // 0x1.921fb82c2bd7fp+1
PRT(0x1.921fb82c2bd7fp1); // 0x1.921fb82c2bd7fp1 = 3.141593
```

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 ϵ

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

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

Простая программа вычисления ϵ

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

```
eps(double)      = 2.22045e-16  log_2(eps) = -52
eps(float)       = 1.19209e-07  log_2(eps) = -23
eps(long double) = 1.0842e-19   log_2(eps) = -63 for X86, 80-bit
```

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 = DBL_EPSILON; // machine epsilon for double
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

```
 0.20000000000000001 0.20000000000000001 -->   1   1   1
 0.40000000000000002 0.40000000000000002 -->   1   1   1
 0.59999999999999998 0.60000000000000009 -->   0   1   1
 0.79999999999999993 0.80000000000000004 -->   0   1   1
 0.99999999999999989 1.00000000000000000 -->   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 ε меньше, чем возможная разность $|x - y|$: например, если `epsilon < 0.01`, то для `float x = 12345.67` нет `y` «близких» к `x`

Относительная разница:

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

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 $x = -1e\text{-}9, y = +1e\text{-}9$ 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);   //  sum= 1.00000000000000000
double sumK = KahanSum(V,3);
printf("sumK= %.17f\n",sumK); // sumK= 1.00000000000000022
```

- Алгоритм Kahan'а не идеален и плохо работает если элемент больше суммы
- Осторожно с оптимизацией, компиляция с `-Ofast` даёт: sumK= 1.00000000000000000
- Имеются другие алгоритмы: улучшенный алгоритм Kahan–Babuška (Neumaier), 2Sum (Knut), Fast2Sum (Dekker) ...
- В python3.12 функция `math.fsum()` использует улучшенный алгоритм Кахана

Математическая библиотека C и C++

Соглашения для `<math.h>` C99

- 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

☞ документация о функциях численной библиотеки C

В C++ рекомендуется использовать `<cmath>`

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

☞ В C++17 включены дополнительные математические специальные функции

Константы (double) в math.h

M_E	Число e	M_PI	Число π	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	$2/\pi$		

☞ 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++20 в `<numbers>` содержится набор «тех же» констант

☞ это шаблоны переменных в пространстве имен `std::numbers`

```
inline constexpr double e;
inline constexpr double log2e
...
```

Комплексные числа в C99: <complex.h>

☞ В C11 разрешено отсутствие <complex.h>: __STDC_NO_COMPLEX__

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

Макросы для комплексного числа: I ≡ _Complex_I ≡ 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)

Комплексные 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

```c
#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
}
```

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» C99
- Большое число функций, алгоритмов ...
- Большое количество языков программирования в которых эта библиотека может быть использована

Дополнительные слайды

Настройка окружения (environment) для FP

Макросы в <fenv.h> в C99, C++11

- Переход в режим генерирования исключений в случае: `divide-by-zero`, `overflow`, `underflow`, `inexact`, `invalid`
- Задание или получение режима округления функциями `fesetround()` и `fegetround()`

Argument/Return value	
`FE_TONEAREST`	к ближайшему
`FE_TOWARDZERO`	к нулю
`FE_UPWARD`	к $+\infty$
`FE_DOWNWARD`	к $-\infty$

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()` из `<fenv.h>`

```
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
```

Not all CPU operations are created equal