

Подсчет денег и времени на компьютере.
Ни цента мимо



Дмитрий Баровик

Дмитрий Баровик



Более 10 лет в IT



БГУ ФПМИ,
Кандидат физико-математических наук



ОАО «Центр банковских технологий»

Руководжу проектами по разработке банковского ПО

Предпочитаемые технологии:

MS Visual C++,
MS SQL Server

История вопроса про центы

Встроенные числовые типы языка C++ (MSDN):

Type Name	Bytes	Other Names	Range of Values
int	4	signed	-2,147,483,648 to 2,147,483,647
short	2	short int, signed short int	-32,768 to 32,767
long	4	long int, signed long int	-2,147,483,648 to 2,147,483,647
long long	8	__int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4		3.4E +/- 38 (7 digits)
double	8		1.7E +/- 308 (15 digits)
long double	8		Same as double

Для расчетов центов и копеек в «базовой комплектации» имеется только два типа:

float и double.

$$2 + 2 = 4 ?$$

```
#include <iostream>
using namespace std;
```

```
int main() {
```

```
    float a, b, r;
```

```
    a = 123456789;
```

```
    b = 123456788;
```

```
    cout << a - b; // result - ?
```

```
    return 0;
```

```
}
```

⚙️ stdout

Result: 8

8 (Ответ должен быть 1)

int вместо float дает правильный ответ

```
#include <iostream>

using namespace std;

int main() {
    int a, b, r;
    a = 123456789;
    b = 123456788;
    cout << a - b;
    return 0;
}
```

 stdout

1

Зависит ли результат от языка программирования

C#

```
using System;

public class Test
{
    public static void Main()
    {
        float a, b, r;
        a = 123456789;
        b = 123456788;
        r = a - b;
        Console.WriteLine("Result: {0}", r);
    }
}
```

⚙️ stdout


Result: 8

Transact SQL

```
/* В T-SQL real
   это 4 байта,
   а float - 8 байт */
DECLARE @a real
DECLARE @b real

SET @a = 123456789
SET @b = 123456788

SELECT @a - @b
```

 Results

8

Может виртуальная Java машина умнее?

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Ideone
{
    public static void main (String[] args)
        throws java.lang.Exception
    {
        float a, b, r;
        a = 123456789;
        b = 123456788;
        r = a - b;
        System.out.format("Result: %f", r);
    }
}
```

⚙️ stdout

Result: 8.000000

Опасная редукция

$a=123\ 456\ 789;$

$b=123\ 456\ 788;$

Относительная погрешность вычисления, по сравнению с точным ответом: 800%
(8 вместо 1)

Операция разности привела к **опасной редукции**.

Так называется катастрофическое понижение точности вычислений в операции, где абсолютное значение результата (в нашем случае 1) много меньше любого из входных значений аргументов (в нашем случае 123 456 789).

Причина этого эффекта – будет пояснена ниже.

Пример с double. Размер не имеет значения

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    double a, b, r;
```

```
    a = 87654321098765432;
```

```
    b = 87654321098765431;
```

```
    cout << a - b; // result - ?
```

```
    return 0;
```

```
}
```

⚙️ stdout

16

Предупреждены ли мы об ошибках?

- Ошибки не зависят от языка программирования
- Ошибки не зависят от процессора или виртуальной машины
- При компиляции нет предупреждений
- При исполнении так же нет исключений или предупреждений



Программа может быть протестирована и успешно работать на одних числах.

А на других «больших» числах может **молча выдать неправильный** результат вычислений!

Хранение чисел с плавающей точкой

Чтобы пояснить в чем причина описанной выше ошибки, познакомимся со стандартом:

Institute of Electrical and Electronics Engineers:

ANSI/IEEE Std 754-1985, Standard for Binary Floating-Point Arithmetic.

Продублирован в международной электротехнической комиссии IEC:

IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems.

В 2008 г. стандарт был выпущен новый стандарт **IEEE 754-2008**, который включил в себя предыдущую версию. Принципиальных изменений не произошло.

ANSI/IEEE Std 754-1985

Используется большинством микропроцессоров, а также логическими и программными устройствами.

Стандарт описывает:

- как представлять числа с плавающей точкой в двоичном виде
- как производить операции над числами
- как представлять нулевые числа
- как представлять специальную величину бесконечность
- как представлять специальную величину "Не число"
- описывает правила нескольких режимов **округления**



Способ хранения чисел

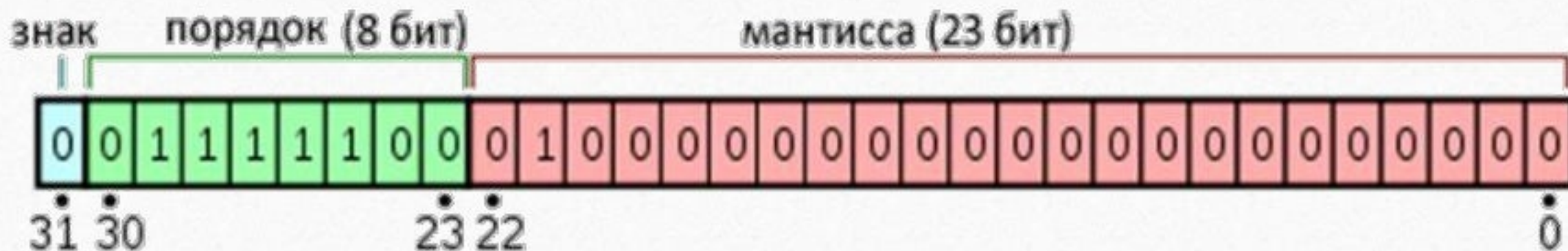
У нас на входе имеется число F , которое необходимо хранить как тип Float (4 байта).
Магическим образом находятся **два целых числа** E и M , такие:

$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

S отвечает за знак (0 – положительное число, 1 – отрицательное).

E и M называют смещенной экспонентой и остатком мантиссы.

Число хранится в таком виде:



Пример хранения нормализованного числа

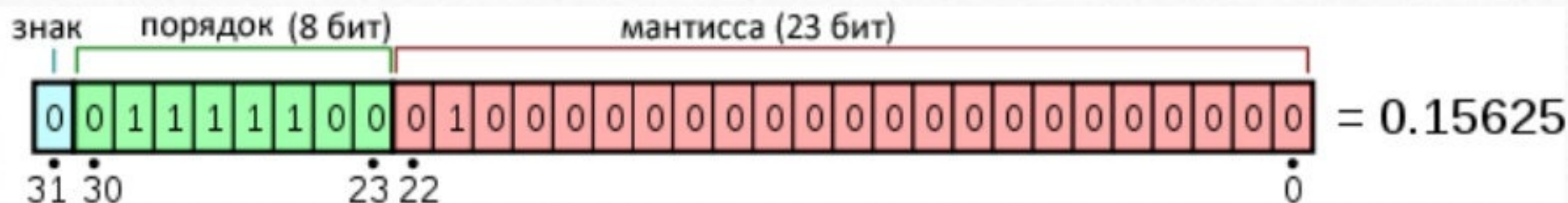
$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

$F = 0.15625$ будет храниться так:

$S = 0$;

$E = 124$;

$M = 2\,097\,152$;

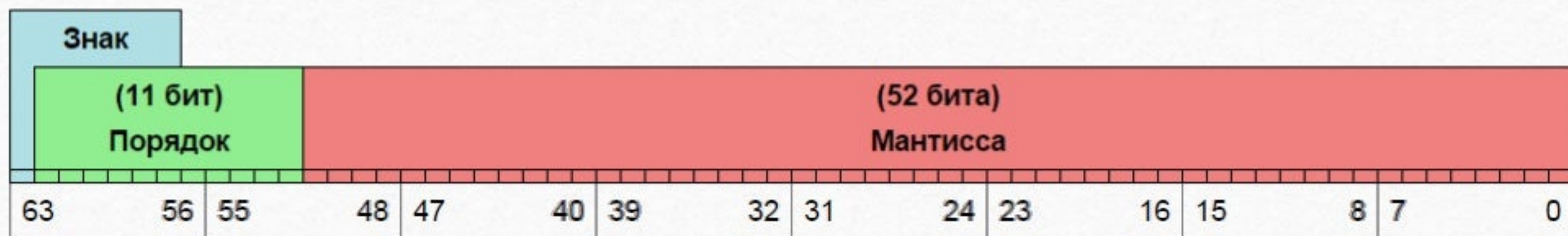


Числа double и «крупнее»

Принцип тот же, просто на мантиссу отводится не 23, а 52 бита.

А на экспоненту 11 бит вместо 8.

$$\pm \text{знак} \cdot 2^E - 1023 \cdot (1 + M/2^{52})$$



Чудеса с целыми числами

$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

Все **целые числа** в определенном диапазоне можно **точно представить** в таком виде (скорее всего 😊).

Но начиная с некоторого целого числа это уже невозможно. И хранится будет другое (ближайшее) число.



Причина опасной редукции

В примере выше:

123456789: $M=7\ 043\ 490$ $E = 153$ \rightarrow **123456784**

123456788: $M=7\ 043\ 491$ $E = 153$ \rightarrow **123456792**

Другие целые числа просто невозможно представить в виде F.

Поэтому вместо числа:

123 456 789 используется ближайшее из доступных 123456792 (на 3 большее)

123 456 788 используется ближайшее из доступных 123456784 (на 5 меньшее)

Их разность и дает ошибку ровно в 8 единиц.

А в чем собственно проблема?

$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

Какая нам разница, как хранится число? В чем собственно вопрос?

Целые числа «неплохо» хранятся в таком виде как показано выше.

А вот для дробных чисел F точно удовлетворяющих этой формуле практически **не существует!** Невозможно найти такие целые E и M !

Примеры хранения центов и копеек

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float a = 1.01f;    float b = 0.01f;    float one = 1.0f;
6
7      float result1 = a-b;
8      if (result1 == one)
9          cout << "1.01-0.01=1 is correct" << endl;
10
11     float result2 = a-one;
12     if (result2 == b)
13         cout << "1.01-1=0.01 is correct";
14     else
15         cout << "1.01-1 equals " << result2;
16
17     return 0;
18 }
```

 Input  Output

Успешно time: 0 memory: 3412 signal:0

1.01-0.01=1 is correct
1.01-1 equals 0.00999999

Примеры хранения центов и копеек

0.01: $M=2348810$ $E=120 \rightarrow 0.00999999977648258209228515625$

1.01: $M=83886$ $E=127 \rightarrow 1.00999999904632568359375$

$1.01 - 0.01 = 1$ \leftarrow Каким-то чудом ответ равен 1.

$1.01 - 1 \neq 0.01$ \leftarrow Здесь ответ получается 0.00999999905

Если $a - b = c$, то не обязательно $a - c = b$

В целом все знакомые нам из математики простые правила типа **ассоциативности**, не работают в арифметике с плавающей точкой на компьютере.

Неассоциативность сложения

```
declare @f1 real
```

```
declare @f2 real
```

```
declare @f3 real
```

```
set @f1 = +113.0
```

```
set @f2 = -111.0
```

```
set @f3 = +7.51
```

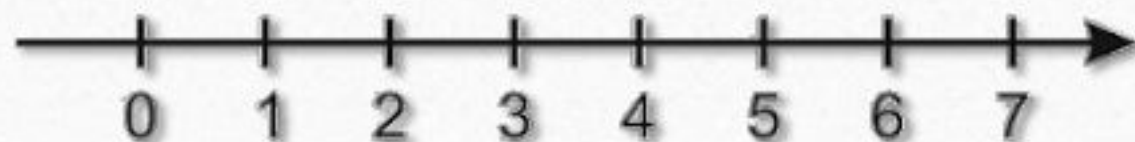
```
select (@f1 + @f2) + @f3 -- Результат: 9,51
```

```
select @f1 + (@f2 + @f3) -- Результат: 9,510002
```

компилятор,
оптимизатор

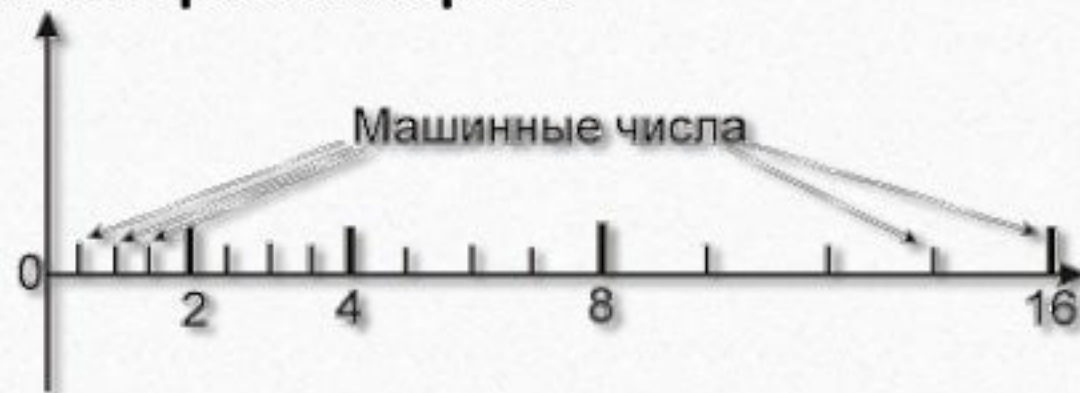
Промежуточные выводы по теории

Целые типы (`int`, `long`) располагаются **равномерно** на цифровой прямой.



А дробные числа в формате IEEE-754 расположены **неравномерно**.

Шаг чисел удваивается с увеличением экспоненты двоичного числа на единицу. То есть, **чем дальше от нуля, тем шире шаг чисел** в формате IEEE754 по числовой оси.



Причем необходимых нам чисел с шагом в центы (копейки) нет в принципе! И в расчетах используются **другие (приближенные) числа**.

«Дикие ошибки» или неприведение типов

```
#include <iostream>
using namespace std;
```

```
int main() {
    float   a;
    double  b;
    double  c;

    a = 123456789.123457;
    b = 123456789.123457;
    c = a - b;

    cout << c;

    return 0;
}
```

Результат: 2.87654

Как было пояснено выше – возможные значения чисел с плавающей точкой – это некий пунктир на числовой прямой. Причем точки для Single, Double и т.д. могут не совпадать.

Существует ряд ошибок, которые вызваны именно тем, что исходные **числа**, представленные в формате **single** и **double** **обычно не равны друг другу**.

Относительная погрешность результата
равна: ∞ (бесконечности).

Такую ошибку называют «**грязным нулем**».

Неявное приведение типов

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    float a = 1.0;
```

```
    float b = 3.0;
```

```
    float c = a / b;
```

```
    float d = (c - 1.0/3.0) * 1000000000;
```

```
    cout << d;
```

```
    return 0;  stdout
```

```
}
```

```
9.93411
```

!!! Переменные и промежуточные результаты компьютерных вычислений должны быть приведены к **одному типу** данных.

Требование приведения данных к одному типу изложено в стандарте на язык Си ISO/IEC 9899:1999.

Рекомендация: чтобы избавиться от ошибок приведения типов данных надо просто использовать тип данных `double` и **забыть о существовании типа `single (float)`**.

Циклические дыры в embedded-системах

Производство
конфеток
 $10 \text{ мг} = 0.0001 \text{ кг}$

Производственная линия:

```
int main(int argc, char *argv[])
{
    cout << "Штатный режим упаковочной машины:"
          << endl;
    cout << 100 /* Вес бункера */
          << -0.0001 /* Вес таблетки */ << endl;

    cout << "После переполнения бункера:"
          << endl;
    cout << 200 - 0.0001 << endl;

    return 0;
}
```

Штатный режим упаковочной машины:

99.9999

После переполнения бункера:

200



Проблема нуля и сравнения

$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

При любых E и M полученное число по модулю всегда строго положительное.

Т.о. представить 0 в таком виде невозможно.

Поэтому были введены два специальных значения **+0** и **-0**.

Так же были введены **+** и **-** **бесконечности**.

А также число NaN – «Not a number». Например, после деления на 0.

Проблема нуля и сравнения

$$F = (-1)^S 2^{(E-127)} (1 + M/2^{23})$$

Не пытайтесь сравнивать выражение с числом нуль.

if (число == 0). Практически всегда это будет иметь значение ложь.

И вообще не сравнивайте числа с плавающей точкой через = и !=,
сравнивайте на $\text{abs}(\text{число1} - \text{число2}) < 0.01 / 2$

if ... else

Проблема нуля в 3D-графике

Система трассировки лучей POV-Ray для создания фотореалистичных изображений

```
#include "colors.inc"
```

```
difference {  
    box      { -1,      1      pigment { Red } }  
    cylinder { -z,      z, 0.5 pigment { Green } }  
    //cylinder { -1.1*z, 1.1*z, 0.5 pigment { Green } }  
}
```

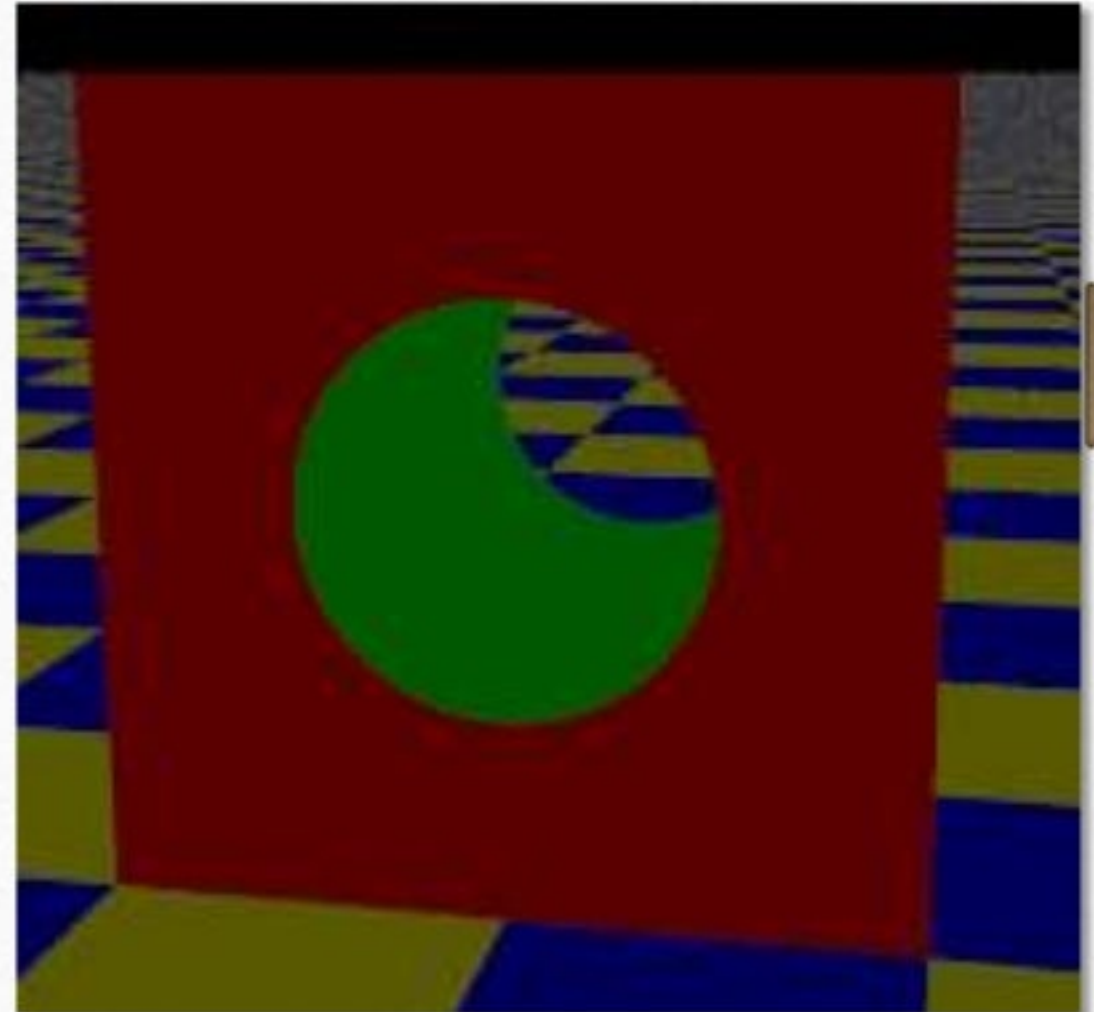
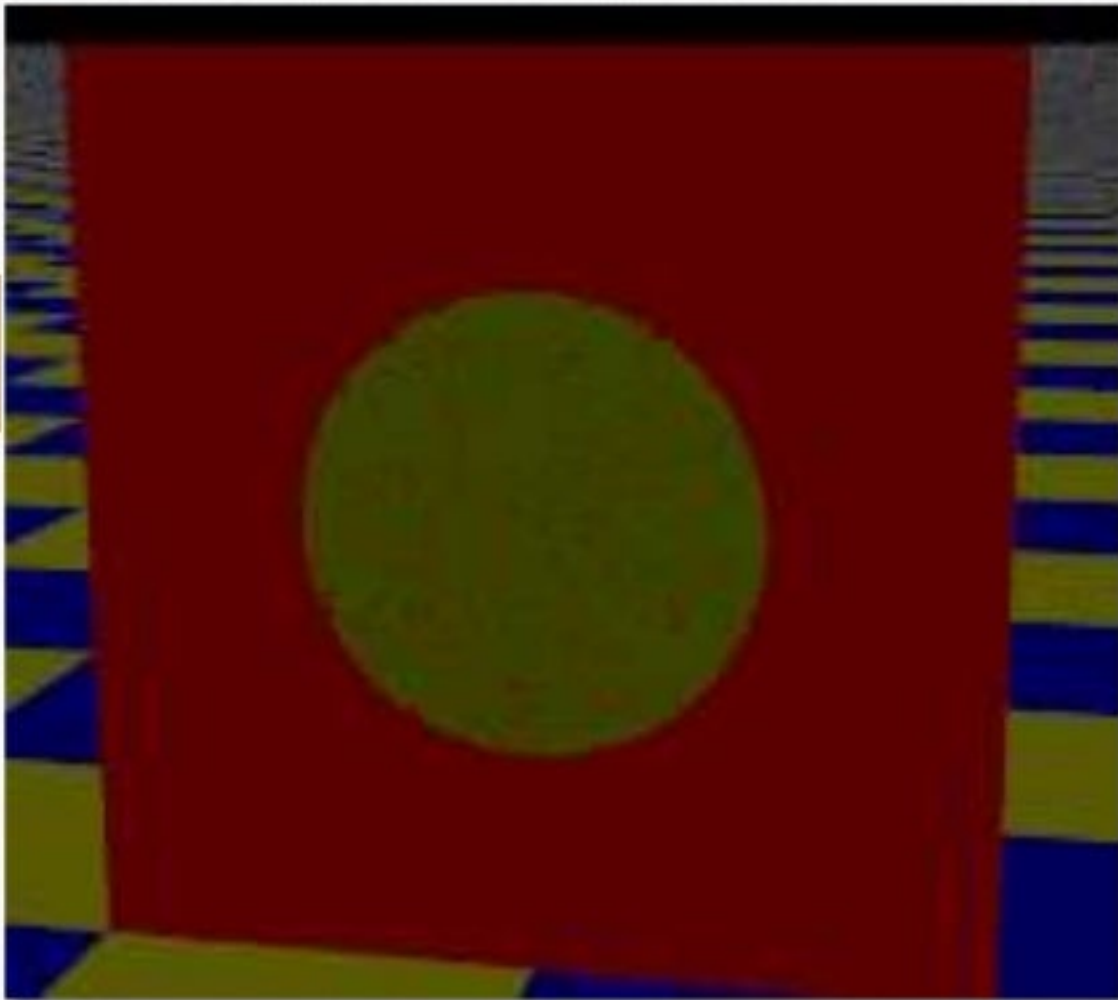
```
camera { location <1, 1, -5> look_at <0, 0, 0> }
```

```
plane { <0, 1, 0>, -1  
    pigment { checker color Yellow, color Blue } }
```



Проблема нуля в 3D-графике

POV-Ray. Зеленый полый цилиндр с крышкой отсекали ровно по крышке при помощи красной плоскости



Числа убийцы

Для примера можно привести баг [№53632](#) для PHP, который вызвал панику в начале 2011 года

```
<html>
  <body>
    <?php $d = 2.2250738585072011e-308; ?>
  </body>
</html>
```

Ввод числа 2.2250738585072011e-308 вызывал зависание процесса почти с 100% загрузкой CPU. Другие числа проблем не вызвали.

Сообщение о баге поступило 30.12.2010, исправлено разработчиком 10.01.2011.

Так как PHP препроцессор использовало множество серверов, то у любого пользователя сети в течение 10 дней была возможность "вырубить" такие сервера.

Граница нормализованных и денормализованных чисел

Описанный способ представления чисел $\pm \text{знак} \cdot 2^E - 1023 \cdot (1 + M/2^{52})$ не единственный, а относится к т.н. нормализованным числам.

Недостаток нормализованных чисел – они недостаточно близко примыкают к нулю. Поэтому **в стандарт были введены еще денормализованные числа**. Которые работают только в диапазоне около нуля. Они **хранятся и вычисляются слегка по другой формуле**.

Когда сопроцессор получает число – ему надо решить это нормализованное или денормализованное число. Т.к. от это будет зависеть способ его представления и обработки в двоичном виде.

Неопределённость переходной зоны заключается в том, что стандарт не определяет конкретного значения границы перехода.

Число $2.2250738585072011e-308$ попало где-то близко к границе денормализации. И цифровому устройству или программе не хватило разрядности для принятия решения к какому диапазону отнести данное число.

Что же делать. Как считать центы

Не используйте float и double 😊

Используйте:

- Числа с **фиксированной точкой**. Примеры:
decimal(n,p) в MS SQL Server;
BigDecimal в Java.
- Числа с **плавающей точкой**, реализованные «нормально», а не по стандарту IEEE754. Например тип **decimal** в C#.
- Ищите в интернете или напишите свой собственный класс с фиксированной точкой на C++. Например на основе целых чисел **long long** (_int64). Переопределите операторы + - * /. И конструкторы для приведения типов.

Что же делать. Как считать центы

Главное правило:

- Будьте внимательны и всегда **проверяйте качество кода**.
- Тестируйте числа и даты для различных региональных настроек.

Заметка про внимательность:

Класс `java.math.BigDecimal` имеет конструктор из строки:

```
bd = new BigDecimal("1.5");
```

Однако разработчики подложили «бомбу» и создали еще конструктор из типа `double`. Забыть кавычки легко (или не читать мануал), результат налицо:

```
bd = new BigDecimal(1.5);
```

bd.toString();

```
// => 0.1499999999999999994448884876874217297881841659545898437
```

Проблемы с датами

```
// Расчет депозита на 500 дней
for (int day_count = 0; day_count <= 500; ++day_count)
{
    CTime et = CTime(2011, 9, 1, 0, 0, 0) + CTimeSpan(day_count, 0, 0, 0);
    listBox.AddString(et.Format(_T("%d.%m.%Y")));
}
```

// Вывод

60 дней 30.11.2011 // Два 30-х ноября в году

61 день 30.11.2011

24.03.2012 // Исчезло 25 марта

26.03.2012

... ..

424 дня 28.10.2012

Проблема в переход на зимнее / летнее время.
На 1 час смещается время

Спасибо за внимание. Пожалуйста вопросы

Подсчет денег и времени на компьютере. Ни цента мимо

Дмитрий Баровик
BarovikD@gmail.com

Использованные источники:

<http://www.softelectro.ru/ieee754.html>

<http://www.h-schmidt.net/FloatConverter/IEEE754.html>

http://dmilvdv.narod.ru/Translate/MISC/how_to_use_java_bigdecimal.html