

# Modern C++ Programming

## 3. BASIC CONCEPTS II

### INTEGRAL AND FLOATING-POINT TYPES

---

*Federico Busato*

2023-11-20

## 1 Integral Data Types

- Fixed Width Integers
- `size_t` and `ptrdiff_t`
- When Use Signed/Unsigned Integer?
- Promotion, Truncation
- Undefined Behavior

## 2 Floating-point Types and Arithmetic

- IEEE Floating-point Standard and Other Representations
- Normal/Denormal Values
- Infinity
- Not a Number (NaN)
- Machine Epsilon
- Units at the Last Place (ULP)
- Cheatsheet
- Summary
- Arithmetic Properties
- Detect Floating-point Errors ★

## **3** Floating-point Issues

- Catastrophic Cancellation
- Floating-point Comparison

# Integral Data Types

---

## A Firmware Bug

*“Certain SSDs have a firmware bug causing them to irrecoverably fail after exactly 32,768 hours of operation. SSDs that were put into service at the same time will fail simultaneously, so RAID won’t help”*

HPE SAS Solid State Drives - Critical Firmware Upgrade





Google AI Blog

The latest news from Google AI

## Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Friday, June 2, 2006

Posted by Joshua Bloch, Software Engineer

Note: Computing the average in the right way is not trivial, see `On finding the average of two unsigned integers without overflow`

related operations: ceiling division, rounding division

# Potentially Catastrophic Failure



$$51 \text{ days} = 51 \cdot 24 \cdot 60 \cdot 60 \cdot 1000 = 4\,406\,400\,000 \text{ ms}$$

---

Boeing 787s must be turned off and on every 51 days to prevent 'misleading data' being shown to pilots



# C++ Data Model

**LP32** Windows 16-bit APIs (no more used)

**ILP32** Windows 32-bit APIs, Unix 32-bit (Linux, Mac OS)

**LLP64** Windows 64-bit APIs

**LP64** Linux 64-bit APIs

Model/Bits	short	int	long	long long	pointer
ILP32	16	32	32	64	32
LLP64	16	32	32	64	64
LP64	16	32	64	64	64

`char` is always 1 byte

```
int*_t <cstdint>
```

C++ provides fixed width integer types.

They have the same size on any architecture:

`int8_t`, `uint8_t`

`int16_t`, `uint16_t`

`int32_t`, `uint32_t`

`int64_t`, `uint64_t`

*Good practice:* Prefer fixed-width integers instead of native types. `int` and `unsigned` can be directly used as they are widely accepted by C++ data models

`int*_t` types are not “real” types, they are merely *typedefs* to appropriate fundamental types

C++ standard does not ensure a one-to-one mapping:

- There are **five** distinct *fundamental types* ( `char` , `short` , `int` , `long` , `long long` )
- There are **four** `int*_t` *overloads* ( `int8_t` , `int16_t` , `int32_t` , and `int64_t` )

Warning: I/O Stream interprets `uint8_t` and `int8_t` as `char` and not as integer values

```
int8_t var;  
cin >> var; // read '2'  
cout << var; // print '2'  
int a = var * 2;  
cout << a; // print '100' !!
```

## size\_t and ptrdiff\_t

```
size_t ptrdiff_t <cstdlib>
```

`size_t` and `ptrdiff_t` are *aliases* data types capable of storing the biggest representable value on the current architecture

- `size_t` is an unsigned integer type (of at least 16-bit)
- `ptrdiff_t` is the signed version of `size_t` commonly used for computing pointer differences
- `size_t` is commonly used to represent size measures
- `size_t` / `ptrdiff_t` are 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures
- C++23 adds `uz` / `UZ` literal for `size_t`
- C++23 adds `z` / `Z` literal for `ptrdiff_t`

Signed and unsigned integers use the same hardware for their operations, but they have very different semantic:

## signed integers

- Represent positive, negative, and zero values ( $\mathbb{Z}$ )
- More negative values ( $2^{31} - 1$ ) than positive ( $2^{31} - 2$ )
- Overflow/underflow is undefined behavior

Possible behavior:

overflow:  $(2^{31} - 1) + 1 \rightarrow \text{min}$

underflow:  $-2^{31} - 1 \rightarrow \text{max}$

- Bit-wise operations are implementation-defined
- Commutative, reflexive, not associative (overflow/underflow)

### unsigned integers

- Represent only *non-negative* values ( $\mathbb{N}$ )
- Overflow/underflow is well-defined (modulo  $2^{32}$ )
- Discontinuity in  $0, 2^{32} - 1$
- Bit-wise operations are well-defined
- Commutative, reflexive, associative

## Google Style Guide

Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers - many members of the standards body believe this to be a mistake, but it is effectively impossible to fix at this point

**Solution:** use `int64_t`

**max value:**  $2^{63} - 1 = 9,223,372,036,854,775,807$  or  
9 quintillion (9 billion of billion),  
about 292 years (nanoseconds),  
9 million terabytes



# Arithmetic Type Limits

Query properties of arithmetic types in C++11:

```
#include <limits>

std::numeric_limits<int>::max();      //  $2^{31} - 1$ 
std::numeric_limits<uint16_t>::max(); // 65,535

std::numeric_limits<int>::min();      //  $-2^{31}$ 
std::numeric_limits<unsigned>::min(); // 0
```

\* this syntax will be explained in the next lectures

# Promotion and Truncation

**Promotion** to a larger type keeps the sign

```
int16_t x = -1;
int      y = x; // sign extend
cout << y;      // print -1
```

**Truncation** to a smaller type is implemented as a modulo operation with respect to the number of bits of the smaller type

```
int      x = 65537; // 2^16 + 1
int16_t y = x;      // x % 2^16
cout << y;          // print 1

int      z = 32769; // 2^15 + 1 (does not fit in a int16_t)
int16_t w = z;      // (int16_t) (x % 2^16 = 32769)
cout << w;          // print -32767
```

```
unsigned a = 10; // array is small
int      b = -1;
array[10ull + a * b] = 0; // ?
```

💀 Segmentation fault!

```
int f(int a, unsigned b, int* array) { // array is small
    if (a > b)
        return array[a - b]; // ?
    return 0;
}
```

💀 Segmentation fault for `a < 0` !

```
// v.size() return unsigned
for (size_t i = 0; i < v.size() - 1; i++)
    array[i] = 3; // ?
```

💀 Segmentation fault for `v.size() == 0` !

Easy case:

```
unsigned x = 32;      // x can be also a pointer
x          += 2u - 4;  // 2u - 4 = 2 + (2^32 - 4)
                  //          = 2^32 - 2
                  // (32 + (2^32 - 2)) % 2^32
cout << x;           // print 30 (as expected)
```

What about the following code?

```
uint64_t x = 32;      // x can be also a pointer
x          += 2u - 4;
cout << x;
```

*More negative values than positive*

```
int x = std::numeric_limits<int>::max() * -1; // (231 - 1) * -1
cout << x;                                // -231 + 1 ok

int y = std::numeric_limits<int>::min() * -1; // -231 * -1
cout << y; // hard to see in complex examples // 231 overflow!!
```

*A practical example:*

```
void f(int* ptr, int pos) {
    pos++;
    if (pos < 0)
        return; // <-- the compiler assumes that
    ptr[pos] = 0; // signed overflow never happen
} // and removes the if statement

int main() { // compiled with optimizations
    int tmp[10]; // leads to segmentation faults
    f(tmp, INT_MAX);
}
```

*Initialize* an integer with a value larger than its range is undefined behavior

```
int z = 3000000000; // undefined behavior!!
```

*Bitwise operations* on signed integer types is undefined behavior

```
int y = 1 << 12;    // undefined behavior!!
```

*Shift* larger than #bits of the data type is undefined behavior even for unsigned

```
unsigned y = 1u << 32u; // undefined behavior!!
```

*Undefined behavior in implicit conversion*

```
uint16_t a = 65535; // 0xFFFF
uint16_t b = 65535; // 0xFFFF
cout << (a * b);    // print '-131071' undefined behavior!! (int overflow)
```

*expected: 4'294'836'225*

*Even worse example:*

```
#include <iostream>

int main() {
    for (int i = 0; i < 4; ++i)
        std::cout << i * 1000000000 << std::endl;
}

// with optimizations, it is an infinite loop
// --> 1000000000 * i > INT_MAX
// undefined behavior!!

// the compiler translates the multiplication constant into an addition
```

Is the following loop safe?

```
void f(int size) {  
    for (int i = 1; i < size; i += 2)  
        ...  
}
```

- What happens if `size` is equal to `INT_MAX` ?
- How to make the previous loop safe?
- `i >= 0 && i < size` is not the solution because of *undefined behavior* of signed overflow
- Can we generalize the solution when the increment is `i += step` ?



# Overflow / Underflow

Detecting overflow/underflow for unsigned integral types is **not trivial**

```
// some examples
bool is_add_overflow(unsigned a, unsigned b) {
    return (a + b) < a || (a + b) < b;
}

bool is_mul_overflow(unsigned a, unsigned b) {
    unsigned x = a * b;
    return a != 0 && (x / a) != b;
}
```

Overflow/underflow for signed integral types is **not defined** !! *Undefined behavior* must be checked before performing the operation

# Floating-point Types and Arithmetic

---

# IEEE Floating-Point Standard

**IEEE754** is the technical standard for floating-point arithmetic

The standard defines the binary format, operations behavior, rounding rules, exception handling, etc.

- First Release: 1985
- Second Release: 2008. Add 16-bit floating point
- Third Release: 2019. Specify min/max behavior

see The IEEE Standard 754: One for the History Books

IEEE764 technical document:

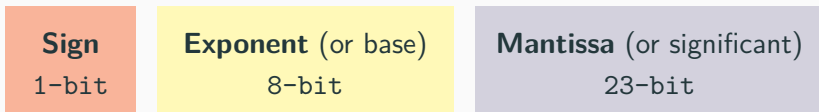
754-2019 - IEEE Standard for Floating-Point Arithmetic

In general, **C/C++ adopts IEEE754 floating-point standard:**

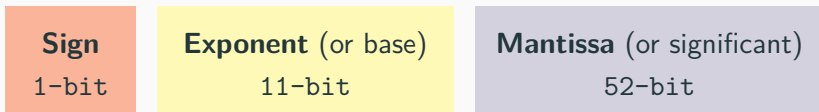
[en.cppreference.com/w/cpp/types/numeric\\_limits/is\\_iec559](https://en.cppreference.com/w/cpp/types/numeric_limits/is_iec559)

# 32/64-bit Floating-Point

- IEEE764 Single precision (32-bit) float



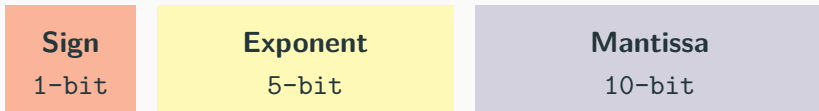
- IEEE764 Double precision (64-bit) double



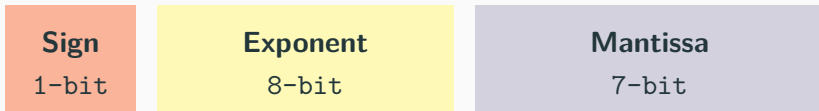
# 16-bit Floating-Point

Standardized in C++23

- **IEEE754 16-bit Floating-point** ( `std::binary16` ) → GPU, Arm7

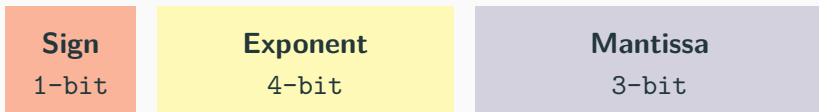


- **Google 16-bit Floating-point** ( `std::bfloat16` ) → TPU, GPU, Arm8

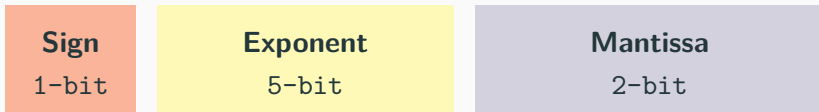


# 8-bit Floating-Point (Non-Standardized in C++/IEEE)

- E4M3



- E5M2



- 
- Floating Point Formats for Machine Learning, *IEEE draft*
  - FP8 Formats for Deep Learning, *Intel, Nvidia, Arm*

- **TensorFloat-32 (TF32)** Specialized floating-point format for deep learning applications
- **Posit** (John Gustafson, 2017), also called *unum III* (*universal number*), represents floating-point values with *variable-width* of exponent and mantissa. It is implemented in experimental platforms

- 
- NVIDIA Hopper Architecture In-Depth
  - Beating Floating Point at its Own Game: Posit Arithmetic
  - Posits, a New Kind of Number, Improves the Math of AI
  - Comparing posit and IEEE-754 hardware cost

- **Microscaling Formats (MX)** Specification for low-precision floating-point formats defined by AMD, Arm, Intel, Meta, Microsoft, NVIDIA, and Qualcomm. It includes FP8, FP6, FP4, (MX)INT8
- **Fixed-point** representation has a fixed number of digits after the radix point (decimal point). The gaps between adjacent numbers are always equal. The range of their values is significantly limited compared to floating-point numbers. It is widely used on embedded systems



## Floating-point number:

- *Radix* (or base):  $\beta$
- *Precision* (or digits):  $p$
- *Exponent* (magnitude):  $e$
- *Mantissa*:  $M$

$$n = \underbrace{M}_p \times \beta^e \rightarrow \text{IEEE754: } 1.M \times 2^e$$

```
float f1 = 1.3f;    // 1.3
float f2 = 1.1e2f;  // 1.1 · 102
float f3 = 3.7E4f;  // 3.7 · 104
float f4 = .3f;     // 0.3
double d1 = 1.3;    // without "f"
double d2 = 5E3;    // 5 · 103
```

## Exponent Bias

In IEEE754 floating point numbers, the exponent value is offset from the actual value by the **exponent bias**

- The exponent is stored as an unsigned value suitable for comparison
- Floating point values are lexicographic ordered
- For a single-precision number, the exponent is stored in the range [1, 254] (0 and 255 have special meanings), and is biased by subtracting 127 to get an exponent value in the range  $[-126, +127]$

0

+

10000111

$$2^{(135-127)} = 2^8$$

110000000000000000000000

$$\frac{1}{2^1} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75 \xrightarrow{\text{normal}} 1.75$$

$$+1.75 * 2^8 = 448.0$$

### Normal number

A **normal** number is a floating point value that can be represented with *at least one bit set in the exponent* or the mantissa has all 0s

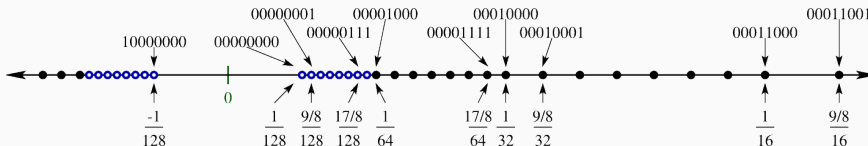
### Denormal number

**Denormal** (or subnormal) numbers fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is denormal

A **denormal** number is a floating point value that can be represented with *all 0s in the exponent*, but the mantissa is non-zero

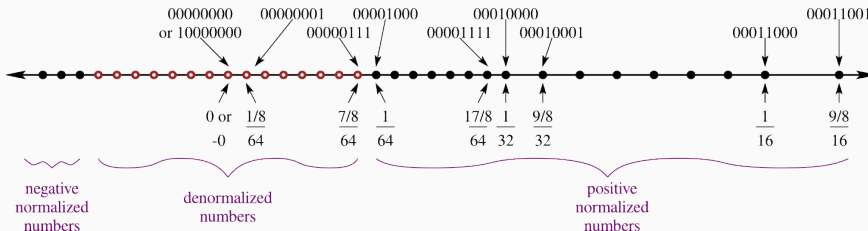
## Why denormal numbers make sense:

(↓ normal numbers)



## The problem: distance values from zero

(↓ denormal numbers)



## Infinity

In the IEEE754 standard, `inf` (infinity value) is a numeric data type value that exceeds the maximum (or minimum) representable value

Operations generating `inf`:

- $\pm\infty \cdot \pm\infty$
- $\pm\infty \cdot \pm\text{finite\_value}$
- $\text{finite\_value} \text{ op } \text{finite\_value} > \text{max\_value}$
- $\text{non-NaN} / \pm 0$

There is a single representation for `+inf` and `-inf`

Comparison:  $(\text{inf} == \text{finite\_value}) \rightarrow \text{false}$   
 $(\pm\text{inf} == \pm\text{inf}) \rightarrow \text{true}$

```
cout << 0 / 0;           // undefined behavior
cout << 0.0 / 0.0;       // print "nan"
cout << 5.0 / 0.0;       // print "inf"
cout << -5.0 / 0.0;      // print "-inf"

auto inf = std::numeric_limits<float>::infinity;
cout << (-0.0 == 0.0);    // true, 0 == 0
cout << ((5.0f / inf) == (-5.0f / inf)); // true, 0 == 0
cout << (10e40f) == (10e40f + 9999999.0f); // true, inf == inf
cout << (10e40)  == (10e40f + 9999999.0f); // false, 10e40 != inf
```

# Not a Number (NaN)

## NaN

In the IEEE754 standard, NaN (not a number) is a numeric data type value representing an undefined or unrepresentable value

Operations generating NaN :

- Operations with a NaN as at least one operand
- $\pm\infty \cdot \mp\infty$  ,  $0 \cdot \infty$
- $0/0$ ,  $\infty/\infty$
- $\sqrt{x}$ ,  $\log(x)$  for  $x < 0$
- $\sin^{-1}(x)$ ,  $\cos^{-1}(x)$  for  $x < -1$  or  $x > 1$

There are many representations for NaN (e.g.  $2^{24} - 2$  for float)

Comparison:  $(\text{NaN} == x) \rightarrow \text{false}$ , for every  $x$

$(\text{NaN} == \text{NaN}) \rightarrow \text{false}$

## Machine epsilon

**Machine epsilon**  $\epsilon$  (or *machine accuracy*) is defined to be the smallest number that can be added to 1.0 to give a number other than one

IEEE 754 Single precision :  $\epsilon = 2^{-23} \approx 1.19209 * 10^{-7}$

IEEE 754 Double precision :  $\epsilon = 2^{-52} \approx 2.22045 * 10^{-16}$



# Units at the Last Place (ULP)

## ULP

**Units at the Last Place** is the gap between consecutive floating-point numbers

$$ULP(p, e) = \beta^{e-(p-1)} \rightarrow 2^{e-(p-1)}$$

Example:

$$\beta = 10, p = 3$$

$$\pi = 3.1415926... \rightarrow x = 3.14 \times 10^0$$

$$ULP(3, 0) = 10^{-2} = 0.01$$

Relation with  $\epsilon$ :

- $\epsilon = ULP(p, 0)$
- $ULP_x = \epsilon * \beta^{e(x)}$

# Floating-Point Representation of a Real Number

The machine floating-point representation  $\text{fl}(x)$  of a *real number*  $x$  is expressed as  $\text{fl}(x) = x(1 + \delta)$ , where  $\delta$  is a small constant

The approximation of a *real number*  $x$  has the following properties:

**Absolute Error:**  $|\text{fl}(x) - x| \leq \frac{1}{2} \cdot \text{ULP}_x$

**Relative Error:**  $\left| \frac{\text{fl}(x) - x}{x} \right| \leq \frac{1}{2} \cdot \epsilon$

- NaN (mantissa  $\neq 0$ )

*	11111111	*****
---	----------	-------

- $\pm$  infinity

*	11111111	000000000000000000000000
---	----------	--------------------------

- Lowest/Largest ( $\pm 3.40282 * 10^{+38}$ )

*	11111110	111111111111111111111111
---	----------	--------------------------

- Minimum (normal) ( $\pm 1.17549 * 10^{-38}$ )

*	00000001	000000000000000000000000
---	----------	--------------------------

- Denormal number ( $< 2^{-126}$ )(minimum:  $1.4 * 10^{-45}$ )

*	00000000	*****
---	----------	-------

- $\pm 0$

*	00000000	000000000000000000000000
---	----------	--------------------------

	E4M3	E5M2	half
<b>Exponent</b>	4 [0*-14] (no inf)	5-bit [0*-30]	
<b>Bias</b>	7	15	
<b>Mantissa</b>	4-bit	2-bit	10-bit
<b>Largest (<math>\pm</math>)</b>	$1.75 * 2^8$ 448	$1.75 * 2^{15}$ 57,344	$2^{16}$ 65,536
<b>Smallest (<math>\pm</math>)</b>	$2^{-6}$ 0.015625	$2^{-14}$ 0.00006	
<b>Smallest (denormal*)</b>	$2^{-9}$ 0.001953125	$2^{-16}$ $1.5258 * 10^{-5}$	$2^{-24}$ $6.0 \cdot 10^{-8}$
<b>Epsilon</b>	$2^{-4}$ 0.0625	$2^{-2}$ 0.25	$2^{-10}$ 0.00098

	bfloat16	float	double
<b>Exponent</b>	8-bit [0*-254]		11-bit [0*-2046]
<b>Bias</b>	127		1023
<b>Mantissa</b>	7-bit	23-bit	52-bit
<b>Largest (<math>\pm</math>)</b>	$2^{128}$ $3.4 \cdot 10^{38}$		$2^{1024}$ $1.8 \cdot 10^{308}$
<b>Smallest (<math>\pm</math>)</b>	$2^{-126}$ $1.2 \cdot 10^{-38}$		$2^{-1022}$ $2.2 \cdot 10^{-308}$
<b>Smallest (denormal*)</b>	/	$2^{-149}$ $1.4 \cdot 10^{-45}$	$2^{-1074}$ $4.9 \cdot 10^{-324}$
<b>Epsilon</b>	$2^{-7}$ 0.0078	$2^{-23}$ $1.2 \cdot 10^{-7}$	$2^{-52}$ $2.2 \cdot 10^{-16}$

# Floating-point - Limits

```
#include <limits>
// T: float or double

std::numeric_limits<T>::max();           // largest value

std::numeric_limits<T>::lowest();        // lowest value (C++11)

std::numeric_limits<T>::min();           // smallest value

std::numeric_limits<T>::denorm_min()     // smallest (denormal) value

std::numeric_limits<T>::epsilon();       // epsilon value

std::numeric_limits<T>::infinity()       // infinity

std::numeric_limits<T>::quiet_NaN()      // NaN
```

# Floating-point - Useful Functions

```
#include <cmath> // C++11

bool std::isnan(T value)      // check if value is NaN
bool std::isinf(T value)      // check if value is  $\pm$ infinity
bool std::isfinite(T value)    // check if value is not NaN
                                // and not  $\pm$ infinity

bool std::isnormal(T value); // check if value is Normal

T    std::ldexp(T x, p)        // exponent shift  $x * 2^p$ 
int  std::ilogb(T value)       // extracts the exponent of value
```

Floating-point operations are written

- $\oplus$  addition
- $\ominus$  subtraction
- $\otimes$  multiplication
- $\oslash$  division

$$\odot \in \{\oplus, \ominus, \otimes, \oslash\}$$

$op \in \{+, -, *, \backslash\}$  denotes exact precision operations



(P1) In general,  $a \text{ op } b \neq a \odot b$

(P2) **Not Reflexive**  $a \neq a$

- *Reflexive* without NaN

(P3) **Not Commutative**  $a \odot b \neq b \odot a$

- *Commutative* without NaN ( $\text{NaN} \neq \text{NaN}$ )

(P4) In general, **Not Associative**  $(a \odot b) \odot c \neq a \odot (b \odot c)$

(P5) In general, **Not Distributive**  $(a \oplus b) \otimes c \neq (a \cdot c) \oplus (b \cdot c)$

(P6) **Identity on operations is not ensured**  $(k \oslash a) \otimes a \neq k$

(P7) **No overflow/underflow** Floating-point has “saturation” values  $\text{inf}$ ,  $-\text{inf}$

- Adding (or subtracting) can “saturate” before  $\text{inf}$ ,  $-\text{inf}$

C++11 allows determining if a floating-point exceptional condition has occurred by using floating-point exception facilities provided in `<cfenv>`

```
#include <cfenv>
// MACRO
FE_DIVBYZERO    // division by zero
FE_INEXACT      // rounding error
FE_INVALID      // invalid operation, i.e. NaN
FE_OVERFLOW     // overflow (reach saturation value +inf)
FE_UNDERFLOW    // underflow (reach saturation value -inf)
FE_ALL_EXCEPT // all exceptions

// functions
std::feclearexcept(FE_ALL_EXCEPT); // clear exception status
std::fetestexcept(<macro>);          // returns a value != 0 if an
                                     // exception has been detected
```

```
#include <cfenv>    // floating point exceptions
#include <iostream>
#pragma STDC FENV_ACCESS ON // tell the compiler to manipulate the floating-point
                             // environment (not supported by all compilers)
                             // gcc: yes, clang: no

int main() {
    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x = 1.0 / 0.0;                  // all compilers
    std::cout << (bool) std::fetestexcept(FE_DIVBYZERO); // print true

    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x2 = 0.0 / 0.0;                  // all compilers
    std::cout << (bool) std::fetestexcept(FE_INVALID);  // print true

    std::feclearexcept(FE_ALL_EXCEPT); // clear
    auto x4 = 1e38f * 10;                 // gcc: ok
    std::cout << std::fetestexcept(FE_OVERFLOW);        // print true
}
```

# Floating-point Issues

---



**Ariane 5:** data conversion from 64-bit floating point value to 16-bit signed integer → *\$137 million*



**Patriot Missile:** small chopping error at each operation, 100 hours activity → *28 deaths*

## Integer type is more accurate than floating type for large numbers

```
cout << 16777217;           // print 16777217  
cout << (int) 16777217.0f;  // print 16777216!!  
cout << (int) 16777217.0;   // print 16777217, double ok
```

## float numbers are different from double numbers

```
cout << (1.1 != 1.1f); // print true !!!
```

## The floating point precision is finite!

```
cout << setprecision(20);  
cout << 3.33333333f; // print 3.333333254!!  
cout << 3.33333333; // print 3.333333333  
cout << (0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1); // print 0.59999999999999998
```

## Floating point arithmetic is not associative

```
cout << 0.1 + (0.2 + 0.3) == (0.1 + 0.2) + 0.3; // print false
```

IEEE764 Floating-point computation guarantees to produce **deterministic** output, namely the exact bitwise value for each run, if and only if the **order of the operations is always the same**

→ *same result on any machine and for all runs*

*“Using a double-precision floating-point value, we can represent easily the number of atoms in the universe.*

*If your software ever produces a number so large that it will not fit in a double-precision floating-point value, chances are good that you have a bug”*

**Daniel Lemire**, Prof. at the University of Quebec

*“ NASA uses just 15 digits of  $\pi$  to calculate interplanetary travel.  
With 40 digits, you could calculate the circumference of a circle the size of the visible universe with an accuracy that would fall by less than the diameter of a single hydrogen atom”*

**Latest in space**, Twitter



# Floating-point Algorithms

- **addition algorithm** (simplified):

- (1) Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
- (2) Add the mantissa
- (3) Normalize the sum if needed (shift right/left the exponent by 1)

- **multiplication algorithm** (simplified):

- (1) Multiplication of mantissas. The number of bits of the result is twice the size of the operands (46 + 2 bits, with +2 for implicit normalization)
- (2) Normalize the product if needed (shift right/left the exponent by 1)
- (3) Addition of the exponents

- **fused multiply-add (fma):**

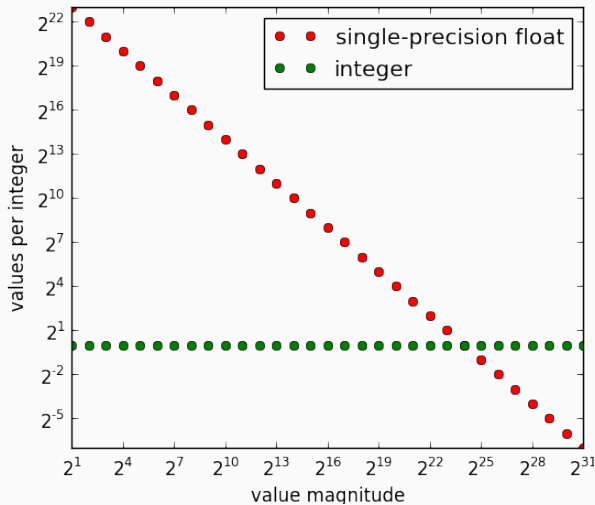
- Recent architectures (also GPUs) provide *fma* to compute addition and multiplication in a single instruction (performed by the compiler in most cases)
- The rounding error of  $fma(x, y, z)$  is less than  $(x \otimes y) \oplus z$

## Catastrophic Cancellation

**Catastrophic cancellation** (or *loss of significance*) refers to loss of relevant information in a floating-point computation that cannot be reversed

Two cases:

- (C1)  $\mathbf{a} \pm \mathbf{b}$ , where  $\mathbf{a} \gg \mathbf{b}$  or  $\mathbf{b} \gg \mathbf{a}$ . The value (or part of the value) of the smaller number is lost
- (C2)  $\mathbf{a} - \mathbf{b}$ , where  $\mathbf{a}, \mathbf{b}$  are approximation of exact values and  $\mathbf{a} \approx \mathbf{b}$ , namely a loss of precision in both  $\mathbf{a}$  and  $\mathbf{b}$ .  $\mathbf{a} - \mathbf{b}$  cancels most of the relevant part of the result because  $\mathbf{a} \approx \mathbf{b}$ . It implies a *small absolute error* but a *large relative error*



**Intersection** = 16,777,216 =  $2^{24}$

*How many iterations performs the following code?*

```
while (x > 0)
    x = x - y;
```

How many iterations?

```
float:  x = 10,000,000  y = 1      -> 10,000,000
float:  x = 30,000,000  y = 1      -> does not terminate
float:  x =    200,000   y = 0.001 -> does not terminate
bfloat: x =           256 y = 1     -> does not terminate !!
```

## Floating-point increment

```
float x = 0.0f;  
for (int i = 0; i < 200000000; i++)  
    x += 1.0f;
```

What is the value of `x` at the end of the loop?

---

## Ceiling division $\left\lceil \frac{a}{b} \right\rceil$

```
//      std::ceil((float) 101 / 2.0f) -> 50.5f -> 51  
float x = std::ceil((float) 20000001 / 2.0f);
```

What is the value of `x`?

Let's solve a quadratic equation:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x^2 + 5000x + 0.25$$

```
(-5000 + std::sqrt(5000.0f * 5000.0f - 4.0f * 1.0f * 0.25f)) / 2 // x2  
(-5000 + std::sqrt(25000000.0f - 1.0f)) / 2 // catastrophic cancellation (C1)  
(-5000 + std::sqrt(25000000.0f)) / 2  
(-5000 + 5000) / 2 = 0 // catastrophic cancellation (C2)  
// correct result: 0.00005!!
```

$$\text{relative error: } \frac{|0 - 0.00005|}{0.00005} = 100\%$$

## The problem

```
cout << (0.11f + 0.11f < 0.22f); // print true!!  
cout << (0.1f + 0.1f > 0.2f);    // print true!!
```

Do not use absolute error margins!!

```
bool areFloatNearlyEqual(float a, float b) {  
    if (std::abs(a - b) < epsilon); // epsilon is fixed by the user  
        return true;  
    return false;  
}
```

Problems:

- Fixed epsilon “looks small” but it could be too large when the numbers being compared are very small
- If the compared numbers are very large, the epsilon could end up being smaller than the smallest rounding error, so that the comparison always returns false

**Solution:** Use relative error  $\frac{|a-b|}{b} < \epsilon$

```
bool areFloatNearlyEqual(float a, float b) {  
    if (std::abs(a - b) / b < epsilon); // epsilon is fixed  
        return true;  
    return false;  
}
```

Problems:

- $a=0$ ,  $b=0$  The division is evaluated as  $0.0/0.0$  and the whole if statement is  $(\text{nan} < \text{epsilon})$  which always returns false
- $b=0$  The division is evaluated as  $\text{abs}(a)/0.0$  and the whole if statement is  $(+\text{inf} < \text{epsilon})$  which always returns false
- $a$  and  $b$  very small. The result should be true but the division by  $b$  may produces wrong results
- It is not commutative. We always divide by  $b$



Possible solution:  $\frac{|a-b|}{\max(|a|,|b|)} < \varepsilon$

```
bool areFloatNearlyEqual(float a, float b) {
    constexpr float normal_min      = std::numeric_limits<float>::min();
    constexpr float relative_error = <user_defined>

    if (!std::isfinite(a) || !isfinite(b)) // a = ±∞, NaN or b = ±∞, NaN
        return false;
    float diff = std::abs(a - b);
    // if "a" and "b" are near to zero, the relative error is less effective
    if (diff <= normal_min) // or also: user_epsilon * normal_min
        return true;

    float abs_a = std::abs(a);
    float abs_b = std::abs(b);
    return (diff / std::max(abs_a, abs_b)) <= relative_error;
}
```

## Minimize Error Propagation - Summary

- Prefer **multiplication/division** rather than addition/subtraction
- Try to reorganize the computation to **keep near** numbers with the same scale (e.g. sorting numbers)
- Consider to **put a zero** very small number (under a threshold). Common application: iterative algorithms
- Scale by a **power of two** is safe
- **Switch to log scale**. Multiplication becomes Add, and Division becomes Subtraction
- Use a **compensation algorithm** like Kahan summation, Dekker's FastTwoSum, Rump's AccSum

# References

## Suggest readings:

- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Do Developers Understand IEEE Floating Point?
- Yet another floating point tutorial
- Unavoidable Errors in Computing

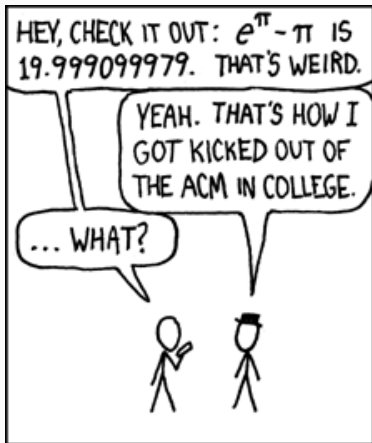
## Floating-point Comparison readings:

- The Floating-Point Guide - Comparison
- Comparing Floating Point Numbers, 2012 Edition
- Some comments on approximately equal FP comparisons
- Comparing Floating-Point Numbers Is Tricky

## Floating point tools:

- IEEE754 visualization/converter
- Find and fix floating-point problems

## On Floating-Point



DURING A COMPETITION, I TOLD THE PROGRAMMERS ON OUR TEAM THAT  $e^\pi - \pi$  WAS A STANDARD TEST OF FLOATING-POINT HANDLERS -- IT WOULD COME OUT TO 20 UNLESS THEY HAD ROUNDING ERRORS.

