# Modern C++ Programming

## 5. Basic Concepts III

### Floating-point Types

*Federico Busato*

2026-01-06

# Table of Contents

# Table of Contents

# Floating-point Types

# Floating-Point Types

| Standard | Type | IEEE754 | Bytes | Min | Max | C++23 `<stdfloat>` |
|---|---|---|---|---|---|---|
| C++23 | (bfloat16) | N | 2 | $\pm 1.18 \times 10^{-38}$ | $\pm 3.4 \times 10^{+38}$ | std::bfloat16_t |
| C++23 | (float16) | Y | 2 | 0.00006 | 65,536 | std::float16_t |
| | `float` | Y | 4 | $\pm 1.18 \times 10^{-38}$ | $\pm 3.4 \times 10^{+38}$ | std::float32_t |
| | `double` | Y | 8 | $\pm 2.23 \times 10^{-308}$ | $\pm 1.8 \times 10^{+308}$ | std::float64_t |
| C++23 | (float128) | Y | 16 | $\pm 3.36 \times 10^{-4032}$ | $\pm 1.18 \times 10^{+4032}$ | std::float128_t |

C++23 Fixed width floating-point types

# Floating-Point Suffix Literals

| Standard | Type | SUFFIX | Example |
|---|---|---|---|
| | `float` | f, F | 3.0f |
| | `double` | | 3.0 |
| C++23 | `std::bfloat16_t` | bf16, BF16 | 3.0bf16 |
| C++23 | `std::float16_t` | f16, F16 | 3.0f16 |
| C++23 | `std::float32_t` | f32, F32 | 3.0f32 |
| C++23 | `std::float64_t` | f64, F64 | 3.0f64 |
| C++23 | `std::float128_t` | f128, F128 | 3.0f128 |

# Floating-Point Limits

```cpp
#include <limits>

std::numeric_limits<int>::max();          // 2^31 − 1
std::numeric_limits<uint16_t>::max();     // 65,535
std::numeric_limits<float>::max();        // 3.4 × 10^38

std::numeric_limits<int>::min();          // −2^31
std::numeric_limits<unsigned>::min();     // 0
std::numeric_limits<float>::min();        // 1.18 × 10^−38

std::numeric_limits<int>::lowest();       // −2^31          same as min()
std::numeric_limits<unsigned>::lowest(); // 0              same as min()
std::numeric_limits<float>::lowest();     // −3.4 × 10^38   NOT the same as min()
```

* this syntax will be explained in the next lectures

# 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, 128-bit, 256-bit floating-point types

*Third Release* : 2019. Specify `min/max` behavior

References:

- `The IEEE Standard 754: One for the History Books`
- `IEEE Standard for Floating-Point Arithmetic` (2019)

# IEEE Floating-Point Standard and C++

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

The supports can be verified with:

```
#include <limits>
std::numeric_limits<float>::is_iec559;
std::numeric_limits<double>::is_iec559;
```

en.cppreference.com/w/cpp/types/numeric_limits/is_iec559

*C++ adopts IEEE754 in most platform, not all!* This allows some operations to have undefined behavior even if IEEE754 is supported

# 32/64-bit Floating-Point

- **IEEE754 Single-precision** (32-bit) `float`

| **Sign** | **Exponent** (or base) | **Mantissa** (or significant) |
|---|---|---|
| 1-bit | 8-bit | 23-bit |

- **IEEE754 Double-precision** (64-bit) `double`

| **Sign** | **Exponent** (or base) | **Mantissa** (or significant) |
|---|---|---|
| 1-bit | 11-bit | 52-bit |

# 128/256-bit Floating-Point

- **IEEE754 Quad-Precision** (128-bit) `std::float128_t` C++23

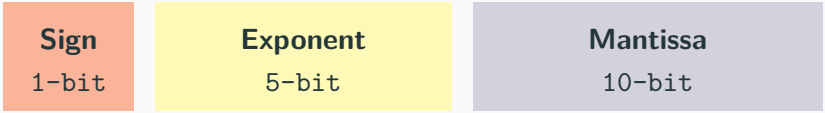
| **Sign** | **Exponent** (or base) | **Mantissa** (or significant) |
|---|---|---|
| 1-bit | 15-bit | 112-bit |

- **IEEE754 Octuple-Precision** (256-bit) (not standardized in C++)

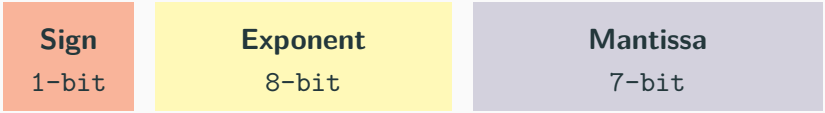
| **Sign** | **Exponent** (or base) | **Mantissa** (or significant) |
|---|---|---|
| 1-bit | 19-bit | 236-bit |

# 16-bit Floating-Point

- **IEEE754 16-bit Floating-point** ( `std::binary16_t` ) C++23 → GPU, Arm7

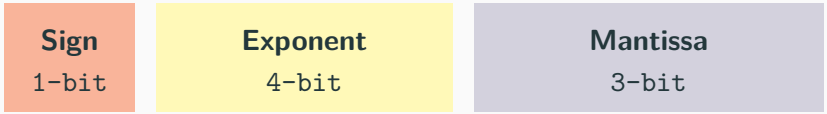



- **Google 16-bit Floating-point** ( `std::bfloat16_t` ) C++23 → TPU, GPU, Arm8

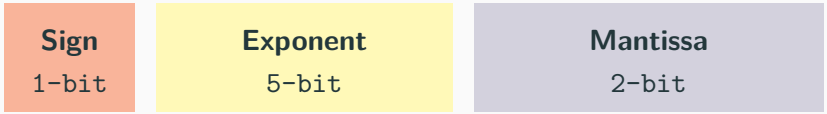


# 8-bit Floating-Point (Non-Standardized in C++/IEEE)

- `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

- OCP Microscaling Formats (MX) Specification

**Floating-point number**:

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

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

```
float  f1 = 1.3f;   // 1.3
float  f2 = 1.1e2f; // 1.1 · 10^2
float  f3 = 3.7E4f; // 3.7 · 10^4
float  f4 = .3f;    // 0.3
double d1 = 1.3;    // without "f"
double d2 = 5E3;    // 5 · 10^3
```

## Exponent Bias

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

- For a single-precision number, the exponent is stored in the range $[1, 254]$.
- 0 and 255 have special meanings: *denormal numbers* and `NaN`.
- The exponent is biased by subtracting 127 to get an exponent value in the range $[-126, +127]$.

| | | |
|---|---|---|
| 0 | 10000111 | 11000000000000000000000 |
| + | $2^{(135-127)} = 2^8$ | $\frac{1}{2^1} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75 \overset{normal}{\rightarrow} 1.75$ |

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

The exponent is stored as an unsigned value *suitable for comparison*.

Except `NaN`, floating point values are lexicographic ordered.

```
int make_float_comparible_as_int(float v) {
    int v_int = std::bit_cast<int>(v); // convert to 'int' without changing the
                                       // underlying representation
    return  v_int < 0 ? 0x80000000 - v_int : v_int;
}

float v1 = ...
float v2 = ...
int   w1 = make_float_comparible_as_int(v1);
int   w2 = make_float_comparible_as_int(v2);
// v1 < v2 -> w1 < w2
```

## Normal number

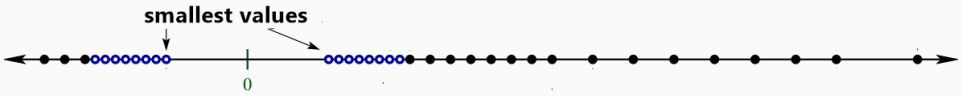
A **normal** number is a floating point value that can be represented with *at least one bit set in the exponent* or the mantissa has all 0s

## Denormal number

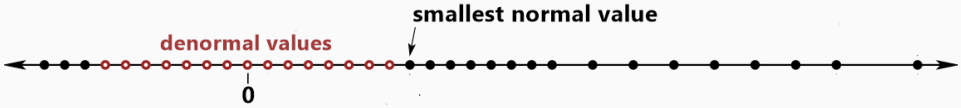
**Denormal** (or subnormal) numbers fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is denormal

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`:

- $\pm\infty \cdot \pm\infty$
- $\pm\infty \cdot \pm$`finite_value`
- `finite_value` *op* `finite_value` $>$ `max_value`
- *finite value* $/ \pm 0$

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

Comparison: `(inf == finite_value)` $\rightarrow$ `false`
`(`$\pm$`inf == `$\pm$`inf)` $\rightarrow$ `true`

```
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
```

**NaN**

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

Floating-point 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* $x > 1$

Comparison: `(NaN == x)` $\rightarrow$ `false`, for every `x`

`(NaN == NaN)` $\rightarrow$ `false`

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

The specific (bitwise) `NaN` value returned by an operation is implementation/compiler specific

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

`quiet_NaN`

# Machine Epsilon

**Machine epsilon**

**Machine epsilon** $\varepsilon$ (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 : $\varepsilon = 2^{-23} \approx 1.19209 * 10^{-7}$

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

# 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 $\varepsilon$:

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

# Floating-Point Representation of a Real Number

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

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

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

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

- NaN (mantissa $\neq 0$)

| * | 11111111 | ********** |
|---|---|---|

- $\pm$ infinity

| * | 11111111 | 00000000000000000000000 |
|---|---|---|

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

| * | 11111110 | 11111111111111111111111 |
|---|---|---|

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

| * | 00000001 | 00000000000000000000000 |
|---|---|---|

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

| * | 00000000 | ********** |
|---|---|---|

- $\pm 0$

| * | 00000000 | 00000000000000000000000 |
|---|---|---|

| | E4M3 | E5M2 | float16_t |
|---|---|---|---|
| **Exponent** | 4 [0*-14] (no inf) | 5-bit [0*-30] | |
| **Bias** | 7 | 15 | |
| **Mantissa** | 4-bit | 2-bit | 10-bit |
| **Largest** ($\pm$) | $1.75 * 2^8$<br>448 | $1.75 * 2^{15}$<br>57,344 | $2^{16}$<br>65,536 |
| **Smallest** ($\pm$) | $2^{-6}$<br>0.015625 | $2^{-14}$<br>0.00006 | |
| **Smallest Denormal** | $2^{-9}$<br>0.001953125 | $2^{-16}$<br>$1.5258 * 10^{-5}$ | $2^{-24}$<br>$6.0 \cdot 10^{-8}$ |
| **Epsilon** | $2^{-4}$<br>0.0625 | $2^{-2}$<br>0.25 | $2^{-10}$<br>0.00098 |

| | `bfloat16_t` | `float` | `double` |
|---|---|---|---|
| **Exponent** | 8-bit [0*-254] | | 11-bit [0*-2046] |
| **Bias** | 127 | | 1023 |
| **Mantissa** | 7-bit | 23-bit | 52-bit |
| **Largest** ($\pm$) | $2^{128}$<br>$3.4 \cdot 10^{38}$ | | $2^{1024}$<br>$1.8 \cdot 10^{308}$ |
| **Smallest** ($\pm$) | $2^{-126}$<br>$1.2 \cdot 10^{-38}$ | | $2^{-1022}$<br>$2.2 \cdot 10^{-308}$ |
| **Smallest Denormal** | / | $2^{-149}$<br>$1.4 \cdot 10^{-45}$ | $2^{-1074}$<br>$4.9 \cdot 10^{-324}$ |
| **Epsilon** | $2^{-7}$<br>0.0078 | $2^{-23}$<br>$1.2 \cdot 10^{-7}$ | $2^{-52}$<br>$2.2 \cdot 10^{-16}$ |

# Floating-point - Limits

```cpp
#include <limits>
// T: float, double, etc.

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

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

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 ±infinity
bool std::isfinite(T value)   // check if value is not NaN
                              // and not ±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

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

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

*(P1)* In general, $a \; op \; b \neq a \odot b$

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

- *Reflexive* without `NaN`

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

- *Commutative* without `NaN` (`NaN` $\neq$ `NaN`)

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

- even excluding `NaN` and `inf` in intermediate computations

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

- even excluding `NaN` and `inf` in intermediate computations

*(P6)* **Identity on operations is not ensured**

- $(a \ominus b) \oplus b \neq a$
- $(a \oslash b) \otimes b \neq a$

*(P7)* **Overflow/Underflow** Floating-point has *"saturation"* values `inf`, `-inf`

- as opposite to integer arithmetic with wrap-around behavior

# Special Values Behavior

## Zero behavior

- $a \oslash 0 =$ `inf`, $a \in \{finite - 0\}$ [IEEE-754], undefined behavior in C++
- $0 \oslash 0$, `inf` $\oslash 0 =$ `NaN` [IEEE-754], undefined behavior in C++
- $0 \otimes$ `inf = NaN`
- `+0 = -0` but they have a different binary representation

## `Inf` behavior

- `inf` $\odot a =$ `inf`, $a \in \{finite - 0\}$
- `inf` $\oplus\otimes$ `inf = inf`
- `inf` $\ominus\oslash$ `inf = NaN`
- $\pm$ `inf` $\oplus\oslash \mp$ `inf = NaN`
- $\pm$ `inf` $= \pm$ `inf`

## `NaN` behavior

- `NaN` $\odot a =$ `NaN`
- `NaN` $\neq a$

# Floating-Point Undefined Behavior

- **Division by zero**
  e.g., $10^8/0.0$
- **Conversion to a narrower floating-point type of a non-representable value**:
  e.g., `0.1 double → float`
- **Conversion from floating-point to integer of a non-representable value**:
  e.g., $10^8$ `float → int`
- **Operations on signaling NaNs**: Arithmetic operations that cause an "invalid operation" exception to be signaled
  e.g., `inf - inf`
- **Incorrectly assuming IEEE-754 compliance for all platforms**:
  e.g., Some embedded Linux distribution on ARM

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
```

```cpp
#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)
int main() {                // gcc: yes, clang: no
    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


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

**Patriot Missile:** small chopping error at each operation, 100 hours activity $\rightarrow$ *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
int   x  = 20000001;
float y  = x;
bool  z1 = (x == y);        // true
bool  z2 = (x == (int) y); // false!!
```

### `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
```

IEEE754 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 $(x \otimes y) \oplus z$

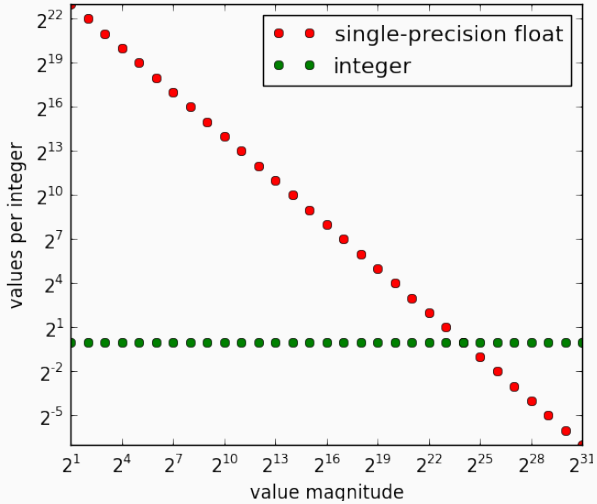
**Catastrophic Cancellation**

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

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 < 20000000; 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!!
```

*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 $\frac{|a-b|}{b} < \varepsilon$

```
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 `(nan < espilon)` which always returns `false`
- **b=0** The division is evaluated as `abs(a)/0.0` and the whole `if` statement is `(+inf < espilon)` 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 **putting 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

**Suggested 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

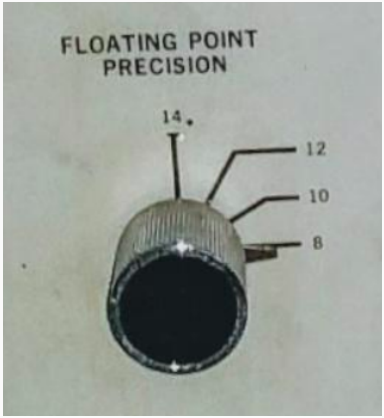
**Floating-point Comparison readings**:

- 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
- float.exposed
- Float Toy
- Find and fix floating-point problems

# System/360 Model 44




Ken Shirriff: Want to adjust your computer's floating point precision by turning a knob? You could do that on the System/360 Model 44

# On Floating-Point