Modern C++ Programming

3. Basic Concepts II

Arithmetic Types

Federico Busato

2025-01-19

Table of Contents

Table of Contents

Table of Contents

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`

Via `twitter.com/martinkl/status/1202235877520482306?s=19`

Overflow Implementations

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

`ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`

Potentially Catastrophic Failure

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

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

C++ Data Model

C++ Data Model	OS	Number of Bits `short`	`int`	`long`	`long long`	`pointer`
ILP32	Windows/Unix 32-b	16	32	32	64	32
LLP64	Windows 64-bit	16	32	32	64	64
LP64	Linux 64-bit	16	32	64	64	64

`char` is always 1 byte

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

int*_t <cstdint>

C++11 provides fixed width integer types.

They have the same number of bits on all architecture:

Number of bits	Signed	Unsigned
8-bit	int8_t	uint8_t
16-bit	int16_t	uint16_t
32-bit	int32_t	uint32_t
64-bit	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`)

ithare.com/c-on-using-int_t-as-overload-and-template-parameters

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

size_t <cstddef>

`size_t` is an *alias* data type capable of storing the biggest representable value on the current architecture

- `size_t` is an unsigned integer type (of at least 16-bit)
- `size_t` is the return type of `sizeof()` and commonly used to represent size measures
- `size_t` is 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures
- C++23 adds `uz` / `UZ` literals for `size_t`, e.g. `5uz`
- C++23 adds `z` / `Z` for the signed version of `size_t`, e.g. `5z`

ptrdiff_t ★

ptrdiff_t <cstddef>

`ptrdiff_t` is an *aliases* data type used to store the results of the difference between pointers or iterators

- `ptrdiff_t` is the signed version of `size_t` commonly used for computing pointer differences
- `ptrdiff_t` are 4 bytes on 32-bit architectures, and 8 bytes on 64-bit architectures

`uintptr_t` ★

`uintptr_t` <cstdint>

`uintptr_t` (C++11) is an integer type that can be converted from and to a `void` pointer

- `uintptr_t` is an unsigned type
- `sizeof(uintptr_t) == sizeof(void*)`
- `uintptr_t` is an *optional* type of the standard and compilers may not provide it

Arithmetic Operation Semantics

Overflow The result of an arithmetic operation exceeds the word length, namely the largest positive/negative values

Wraparound The result of an arithmetic operation is reduced modulo 2^N where N is the number of bits of the word

Saturation The result of an arithmetic operation is constrained within a fixed range defined by a minimum and maximum value. If the result of an operation exceeds this range, it is *"clamped"* to the boundary value

Signed/Unsigned Integer Characteristics

Without undefined behavior, ***signed*** and ***unsigned*** integers use the same exact hardware, and they are equivalent at binary level thanks to the two-complement representation

```
#include <cstdint>
int      a1 = INT_MAX;
int      b1 = a1 + 4; // 10000000000000000000000000000011

unsigned a2 = INT_MAX;
unsigned b2 = a2 + 4; // 10000000000000000000000000000011
```

However, ***signed*** and ***unsigned*** integers have different semantics in C++. The compiler can exploit undefined behavior to optimize the code even if such operations are well-defined at hardware level

Signed Integer

- Represent positive, negative, and zero values ($\mathbb{Z}$)
- *Properties*: Commutative, reflexive, not associative (overflow/underflow)
  ```
  (x + y) + z != x + (y + z)
  ```
- ✅ Represent the human intuition of numbers
- ✅ All bitwise operations are well-defined, except shift

Signed Integer - Problems

⚠ More negative values ($2^{31} - 1$) than positive ($2^{31} - 2$)

Even multiply, division, and modulo by -1 can fail, e.g. `INT_MIN * -1`

⚠ *Overflow/underflow semantic* → undefined behavior

Possible behavior: *overflow*: $(2^{31} - 1) + 1 \rightarrow min$, *underflow*: $-2^{31} - 1 \rightarrow max$

⚠ Shift could lead to undefined behavior `x << y`

- undefined behavior if `y` is larger than the number of bits of `x`
- implementation-defined if `x` is negative (until C++20)
- undefined behavior if `y` is negative

Unsigned Integer

- Represent only *non-negative* values ($\mathbb{N}$)
- *Properties*: commutative, reflexive, associative
- Discontinuity in 0, $2^{32} - 1$
- ✅ Wraparound semantic $\rightarrow$ well-defined (modulo 2^{32})
- ✅ Bit-wise operations are well-defined, except shift
- ⚠ Shift could lead to undefined behavior `x << y`
  - undefined behavior if `y` is larger than the number of bits of `x`

When Use Signed/Unsigned Integer? 1/2

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 in nanoseconds,
9 million terabytes

When Use Signed/Unsigned Integer?

When use `signed` integer?

- if it can be mixed with negative values, e.g. subtracting byte sizes
- prefer expressing non-negative values with signed integer and assertions
- optimization purposes, e.g. exploit undefined behavior for overflow or in loops

When use `unsigned` integer?

- if the quantity can never be mixed with negative values (?)
- bitmask values
- optimization purposes, e.g. division, modulo
- safety-critical system, signed integer overflow could be "non-deterministic"

Subscripts and sizes should be signed, *Bjarne Stroustrup*
Don't add to the signed/unsigned mess, *Bjarne Stroustrup*
Integer Type Selection in C++: in Safe, Secure and Correct Code, *Robert C. Seacord*

```
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;
```

A real-world case:

```
// allocate a zero rtx vector of N elements
//
// sizeof(struct rtvec_def) == 16
// sizeof(rtunion) == 8
rtvec rtvec_alloca(int n) {
    rtvec rt;
    int i;
    rt = (rtvec)obstack_alloc(
        rtl_obstack,
        sizeof(struct rtvec_def) + ((n - 1) * sizeof(rtunion)));
//  ...
    return rt;
}
```

Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Daemons, Chandler Carruth, CppCon 2016

The C++ standard does not prescribe any specific behavior (undefined behavior) for several integer/unsigned arithmetic operations

- *Signed integer overflow/underflow*

```
int x = std::numeric_limits<int>::max() + 20; // x=?
if (y < y + 5) // y: int -> can be always true (compiler optmizes it away)
```

- *More negative values than positive*

```
int x = std::numeric_limits<int>::max() * -1; //  (2^31 -1) * -1
cout << x;                                    //  -2^31 +1 ok

int y = std::numeric_limits<int>::min() * -1; // -2^31 * -1
cout << y; // hard to see in complex examples // 2^31 overflow!!
```

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

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

- *Bitwise operations* on signed integer types with negative value is undefined behavior

```
int y = -1 << 12;    // undefined behavior!!, until C++20
int z = 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                  expected: 4'294'836'225
cout << (a * b);    // print '-131071' undefined behavior!! (int overflow)
```

Undefined Behavior - Signed Overflow Example 1


```
#include <climits>
#include <cstdio>

void f(int* ptr, int pos) {
    pos++;
    if (pos < 0)    // <-- the compiler could assume that signed overflow never
        return;     //     happen and "simplify" the condition to check
    ptr[pos] = 0;
}

int main() {                    // the code compiled with optimizations, e.g. -O3
    int* tmp = new int[10];  // leads to segmentation faults with clang, while
    f(tmp, INT_MAX);            // it terminates correctly with gcc
    printf("%d\n", tmp[0]);
}
```

s/open.c of the Linux kernel

```
int do_fallocate(..., loff_t offset, loff_t len) {
    inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes) || (offset + len < 0))
        return -EFBIG;      // the compiler is able to infer that both 'offset' and
    ...                     // 'len' are non-negative and can eliminate this check,
}                           // without verify integer overflow
```

`src/backend/utils/adt/int8.c` of PostgreSQL

```
if (arg2 == 0) {
    ereport(ERROR, (errcode(ERRCODE_DIVISION_BY_ZERO), // the compiler is not aware
            errmsg("division by zero")));              // that this function
}                                                      // doesn't return
/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);  // the compiler assumes that the divisor is
                                       // non-zero and can move this statement on
                                       // the top  (always executed)
```

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
```


Why does this loop produce undefined behavior?

Is the following loop safe?

```
void f(int size) {
    for (int i = 0; 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`?

Detecting Overflow / Underflow

Detecting wraparound 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;
}
```

Detecting overflow/underflow for signed integral types is even harder and must be checked before performing the operation

Safe Comparison Operators ★

C++20 introduces a set of functions `<utility>` to safely compare integers of different types (`signed`, `unsigned`)

```
bool cmp_equal(T1 a, T2 b)
bool cmp_not_equal(T1 a, T2 b)
bool cmp_less(T1 a, T2 b)
bool cmp_greater(T1 a, T2 b)
bool cmp_less_equal(T1 a, T2 b)
bool cmp_greater_equal(T1 a, T2 b)
```

example:

```
#include <utility>
unsigned a  = 4;
int      b  = -3;
bool     v1 = (a > b);                // false!!!
bool     v2 = std::cmp_greater(a, b); // true
```

How to compare signed and unsigned integers in C++20?

Saturation Arithmetic ⋆

C++26 adds four main functions to perform **saturation arithmetic** with integer types in the `<numeric>` library. In other words, the undefined behavior or the wrap-around behavior for overflow/underflow is replaced by **saturation** values, namely the *minimum* or *maximum* values of the operands

- `T add_sat(T x, T y)`
- `T sub_sat(T x, T y)`
- `T mul_sat(T x, T y)`
- `T div_sat(T x, T y)`
- `R saturate_cast<R>(T x)`

Arithmetic Types - Promotion and Conversion Rules

Arithmetic Types Conversion Rules

Implicit type conversion rules, applied in order, before any operation:

$\otimes$: any arithmetic and bitwise operation (`*, +, /, -, %, &` , etc.), except shift `<<`

(A) Floating point promotion

`floating_type` $\otimes$ `integer_type` $\rightarrow$ `floating_type`

(B) Implicit integer promotion

`small_integral_type` := any signed/unsigned integral type smaller than `int`

`small_integral_type` $\otimes$ `small_integral_type` $\rightarrow$ `int`

(C) Size promotion

`small_type` $\otimes$ `large_type` $\rightarrow$ `large_type`

(D) Sign promotion

`signed_type` $\otimes$ `unsigned_type` $\rightarrow$ `unsigned_type`

Examples and Common Errors

```
float    f = 1.0f;
unsigned u = 2;
int      i = 3;
short    s = 4;
uint8_t  c = 5; // unsigned char

f * u; // float × unsigned → float: 2.0f
s * c; // short × unsigned char → int: 20
u * i; // unsigned × int → unsigned: 6u
+c;    // unsigned char → int: 5
```

Integers are not floating points!

```
int   b = 7;
float a = b / 2;   // a = 3 not 3.5!!
int   c = b / 2.0; // again c = 3 not 3.5!!
```

Integral Promotion

Integral data types smaller than 32-bit, both *signed* or *unsigned*, are *implicitly* promoted to `int` for any arithmetic and bitwise operation, except shift `«`

- Unary `+, -, ~, ++, -`
- Binary `+, *, -, %, \, &, |, ^` and compound operators:

```
char a = 48;      // '0'
cout << a;        // print '0'
cout << +a;       // print '48'
cout << (a + 0); // print '48'

uint8_t a1 = 255;
uint8_t b1 = 255;
cout << (a1 + b1);  // print '510' (no overflow)
```

Integral Promotion - Special Cases

`Logical not` `!` converts to boolean

```
int x = 4;
int y = !!x; // y = 1, not 4
```

`Bitwise shift` `<<` result type is the same of the *left operand*

```
unsigned long long power_of_two_64bit(unsigned long long exponent) {
    return 1 << (exponent % 64);
}

power_of_two_64bit(40); // undefined behavior
```

Truncation

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
```

Floating-point Types

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

Sign	**Exponent**	**Mantissa**
1-bit	5-bit	10-bit

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

Sign	**Exponent**	**Mantissa**
1-bit	8-bit	7-bit

half-precision-arithmetic-fp16-versus-bfloat16

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

- `E4M3`

Sign	Exponent	Mantissa
1-bit	4-bit	3-bit

- `E5M2`

Sign	Exponent	Mantissa
1-bit	5-bit	2-bit

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

Other Real Value Representations (Non-standardized in C++/IEEE) 1/2

- **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

Other Real Value Representations (Non-standardized in C++/IEEE) 2/2

- **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): β
- *Precision* (or digits): p
- *Exponent* (magnitude): e
- *Mantissa*: M

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

```cpp
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**

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

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

Floating-point - Normal/Denormal 1/2

Normal number

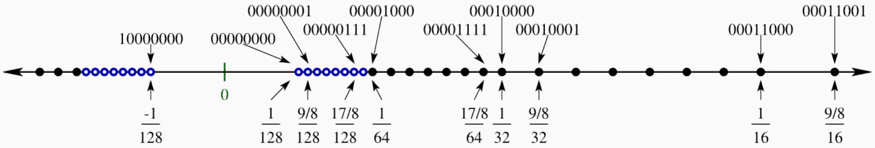
A **normal** number is a floating point value that can be represented with *at least one bit set in the exponent* or the mantissa has all 0s

Denormal number

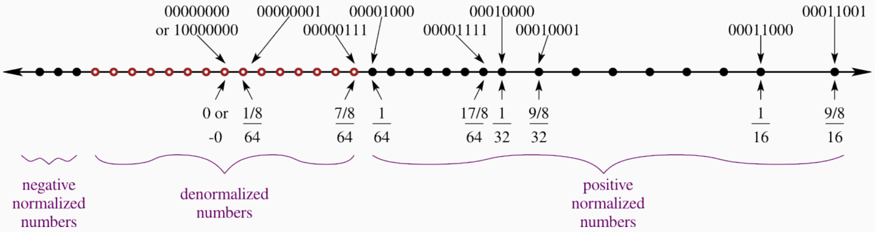
Denormal (or subnormal) numbers fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is denormal

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)



Floating-point representation, by Carl Burch

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`

`(±inf == ±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 ε (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 ε:

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

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 δ 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: $\left|\frac{fl\,(x) - x}{x}\right| \leq \frac{1}{2} \cdot \varepsilon$

Floating-point - Cheatsheet 1/3

- 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	***********************

- ± 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 (±)	$1.75 * 2^8$ 448	$1.75 * 2^{15}$ $57,344$	2^{16} $65,536$
Smallest (±)	2^{-6} 0.015625	2^{-14} 0.00006	
Smallest Denormal	2^{-9} 0.001953125	2^{-16} $1.5258 * 10^{-5}$	2^{-24} $6.0 \cdot 10^{-8}$
Epsilon	2^{-4} 0.0625	2^{-2} 0.25	2^{-10} 0.00098

Floating-point - Cheatsheet 3/3

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

Floating-point - Limits

```
#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 Arithmetic Properties 1/3

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

Floating-point Arithmetic Properties

(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

- $a \oslash 0 = \texttt{inf}, \; a \in \{finite - 0\}$ [IEEE-764], undefined behavior in C++
- $0 \oslash 0, \;\; \texttt{inf} \oslash 0 = \texttt{NaN}$ [IEEE-764], undefined behavior in C++
- $0 \otimes \texttt{inf} = \texttt{NaN}$
- `+0 = -0` but they have a different binary representation

`Inf` behavior

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

`NaN` behavior

- $\texttt{NaN} \odot a = \texttt{NaN}$
- $\texttt{NaN} \neq a$

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
```

Detect Floating-point Errors ★

```
#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

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
```

`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 π 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*

`Number of atoms in the universe versus floating-point values`

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 $(x \otimes y) \oplus z$

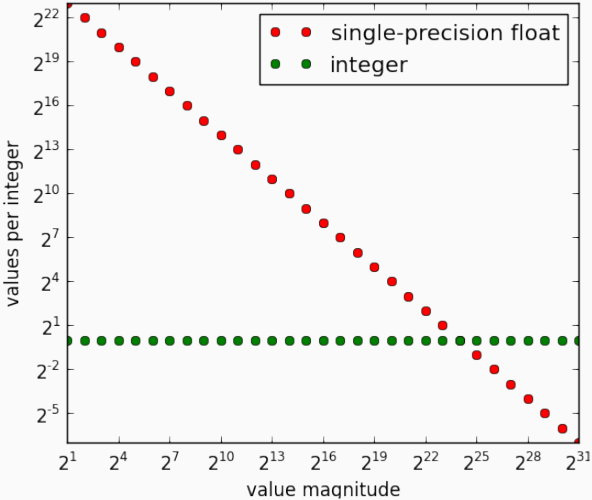
Catastrophic Cancellation

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!!
```

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

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`

Floating-point Comparison

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

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

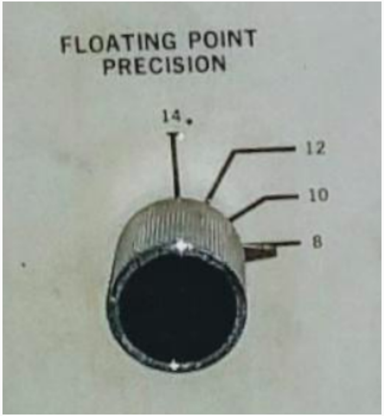
Floating-point Comparison readings:

- 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

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