

# Modern C++ Programming

## 2. BASIC CONCEPTS I - FUNDAMENTAL TYPES

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2021, v3.23



## 1 Preparation

- What compiler should I use?
- What editor/IDE compiler should I use?
- How to compile?

## 2 Hello World

- I/O Stream

## 3 Fundamental Types Overview

- Arithmetic Types
- Non-Standard Arithmetic Types
- `void` Type
- Pointer Type and `nullptr`

## 4 Conversion Rules

## 5 C++ Operators

- Operators Precedence
- Prefix/Postfix Increment/Decrement Semantic
- Assignment, Compound, and Comma Operators
- Spaceship Operator <=>

## 6 Integral Data Types

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

## 7 Floating-point Types and Arithmetic

- Normal/Denormal Values
- Summary
- Not a Number (NaN)
- Infinity
- Properties

## 8 Floating-point Issues

- Floating-point Comparison
- Catastrophic Cancellation

# Preparation

---

# What Compiler Should I Use?

Most popular compilers:

- Microsoft Visual Code (**MSVC**) is the compiler offered by Microsoft
- The GNU Compiler Collection (**GCC**) contains the most popular C++ Linux compiler
- **Clang** is a C++ compiler based on LLVM Infrastructure available for Linux/Windows/Apple (default) platforms

Suggested compiler for beginner: **Clang**

- Comparable performance with GCC/MSVC and low memory usage
- Expressive diagnostics (examples and propose corrections)
- Strict C++ compliance. GCC/MSVC compatibility (inverse direction is not ensured)
- Includes very useful tools: memory sanitizer, static code analyzer, automatic formatting, linter, etc.

# Install the Compiler on Linux

Install the last gcc/g++ (v11)

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install gcc-11 g++-11
$ gcc-11 --version
```

Install the last clang/clang++ (v13)

```
$ wget https://github.com/llvm/llvm-project/releases/download/\
llvmorg-13.0.0/clang+llvm-13.0.0-x86_64-linux-gnu-ubuntu-20.04.tar.xz
$ tar xf clang+llvm-13.0.0-x86_64-linux-gnu-ubuntu-20.04.tar.xz
$ PATH=$PATH:$(pwd)/bin
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(pwd)/lib64 # $
$ clang-13 --version
```

# Install the Compiler on Windows

## Microsoft Visual Studio

- Direct Installer: Visual Studio Community 2019

## Clang on Windows

Two ways:

- Windows Subsystem for Linux (WSL)
  - Run → optionalfeatures
  - Select Windows Subsystem for Linux, Hyper-V, Virtual Machine Platform
  - Run → ms-windows-store: → Search and install Ubuntu 18.04 LTS
- Clang + MSVC Build Tools
  - Download Build Tools per Visual Studio
  - Install Desktop development with C++



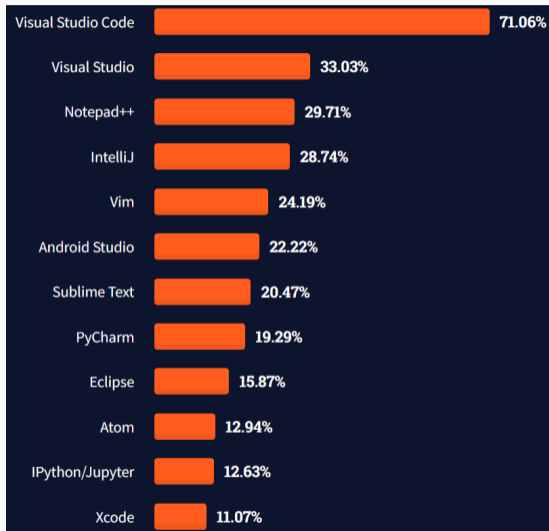
Popular C++ IDE (Integrated Development Environment):

- **Microsoft Visual Studio** (MSVC) ([link](#)). Most popular IDE for Windows
- **Clion** ([link](#)). (free for student). Powerful IDE with a lot of options
- **QT-Creator** ([link](#)). Fast (written in C++), simple
- **XCode**. Default on Mac OS
- **Cevelop** (Eclipse) ([link](#))

Standalone editors for coding (multi-platform):

- **Microsoft Visual Studio Code** (VSCode) ([link](#))
- **Atom** ([link](#)) by GitHub/Microsoft
- **Sublime Text editor** ([link](#)), written in C++
- **Vim**. Powerful, but needs expertise

*Not suggested:* Notepad, Gedit, and other similar editors (lack of support for programming)



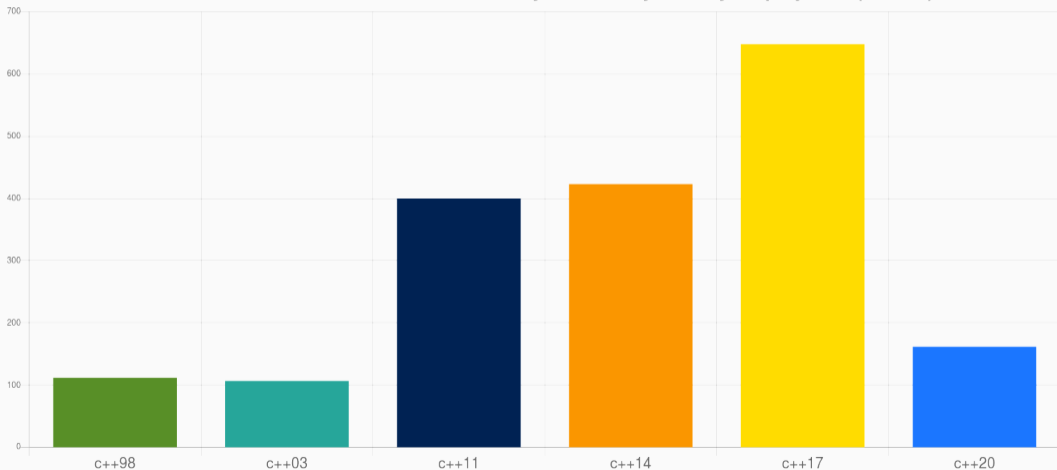
# How to Compile?

Compile C++11, C++14, C++17, C++20 programs:

```
g++ -std=c++11 <program.cpp> -o program
g++ -std=c++14 <program.cpp> -o program
g++ -std=c++17 <program.cpp> -o program
g++ -std=c++20 <program.cpp> -o program
```

Compiler	C++11		C++14		C++17		C++20	
	Core	Library	Core	Library	Core	Library	Core	Library
g++	4.8.1	5.1	5.1	5.1	7.1	9.0	11+	11+
clang++	3.3	3.3	3.4	3.5	5.0	11.0	12+	14+
MSVC	19.0	19.0	19.10	19.0	19.15	19.15	19.29+	19.29

Meeting C++ Community Survey  
Results for 2020 - Which C++ Standards do you currently use in your projects? (n=1030)



# Hello World

---

C code with `printf` :

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

`printf`

prints on standard output

C++ code with `streams` :

```
#include <iostream>

int main() {
    std::cout << "Hello World!\n";
}
```

`cout`

represent the standard output stream

The previous example can be written with the global `std` namespace:

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

int main() {
    cout << "Hello World!\n";
}
```

**Note:** For sake of space and for improving the readability, we intentionally omit the `std` namespace in the next slides

`std::cout` is an example of *output* stream. Data is redirected to a destination, in this case the destination is the standard output

C:

```
#include <stdio.h>
int main() {
    int    a    = 4;
    double b    = 3.0;
    char   c[]  = "hello";
    printf("%d %f %s\n", a, b, c);
}
```

C++:

```
#include <iostream>
int main() {
    int    a    = 4;
    double b    = 3.0;
    char   c[]  = "hello";
    std::cout << a << " " << b << " " << c << "\n";
}
```



- **Type-safe:** The type of object pass to I/O stream is known statically by the compiler. In contrast, `printf` uses `%` fields to figure out the types dynamically
- **Less error prone:** With IO Stream, there are no redundant `%` tokens that have to be consistent with the actual objects pass to I/O stream. Removing redundancy removes a class of errors
- **Extensible:** The C++ IO Stream mechanism allows new user-defined types to be pass to I/O stream without breaking existing code
- **Comparable performance:** If used correctly may be faster than C I/O (`printf`, `scanf`, etc.) .

- Forget the number of parameters:

```
printf("long phrase %d long phrase %d", 3);
```

- Use the wrong format:

```
int a = 3;  
...many lines of code...  
printf(" %f", a);
```

- The `%c` conversion specifier does not automatically skip any leading white space:

```
scanf("%d", &var1);  
scanf(" %c", &var2);
```

# Fundamental Types

## Overview

---

# Arithmetic Types

Type	Bytes	Range	Fixed width types
bool	1	true, false	
char <sup>†</sup>	1	-127 to 127	
signed char	1	-128 to 127	int8_t
unsigned char	1	0 to 255	uint8_t
short	2	$-2^{15}$ to $2^{15}-1$	int16_t
unsigned short	2	0 to $2^{16}-1$	uint16_t
int	4	$-2^{31}$ to $2^{31}-1$	int32_t
unsigned int	4	0 to $2^{32}-1$	uint32_t
long int	4/8		int32_t/int64_t
long unsigned int	4/8*		uint32_t/uint64_t
long long int	8	$-2^{63}$ to $2^{63}-1$	int64_t
long long unsigned int	8	0 to $2^{64}-1$	uint64_t
float (IEEE 754)	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	
double (IEEE 754)	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{+308}$	

\* 4 bytes on Windows64 systems, <sup>†</sup> one-complement

## Arithmetic Types - Short Name

---

Signed Type	short name
signed char	/
signed short int	short
signed int	int
signed long int	long
signed long long int	long long

---

---

Unsigned Type	short name
unsigned char	/
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long
unsigned long long int	unsigned long long

---

## Arithmetic Types - Suffix and Prefix

Type	SUFFIX	example
int	/	2
unsigned int	u	3u
long int	l	8l
long unsigned	ul	2ul
long long int	ll	4ll
long long unsigned int	ull	7ull
float	f	3.0f
double		3.0

Representation	PREFIX	example
Binary C++14	0b	0b010101
Octal	0	0308
Hexadecimal	0x or 0X	0xFFA010

C++14 allows also *digit separators* for improving the readability `1'000'000`

## Other Arithmetic Types

- C++ provides also `long double` (no IEEE-754) of size 8/12/16 bytes depending on the implementation
- C++ (until C++23\*) does not provide support for **16-bit float** data type (IEEE 754-2008)
  - Some compilers already provide support for half float (GCC for ARM: `__fp16` , LLVM compiler: `half` )
  - Some modern CPUs (+ Nvidia GPUs) provide half-float instructions
  - Software support (OpenGL, Photoshop, Lightroom, `half.sourceforge.net`)
- C++ does not provide **128-bit integers** even if some architectures support it. `clang` and `gcc` allow 128-bit integers as compiler extension (`__int128` )

---

\* Extended floating-point types and standard names

## size\_t and std::byte

### size\_t <cstdint>

`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)
- In common C++ implementations:
  - `size_t` is 4 bytes on 32-bit architectures
  - `size_t` is 8 bytes on 64-bit architectures
- `size_t` is commonly used to represent size measures

C++17 defines also `std::byte` type to represent a collection of bit ( `<cstdint>` ).  
It supports only bitwise operations (no conversions or arithmetic operations)



# void Type

`void` is an incomplete type (not defined) without a value

- `void` indicates also a function with no return type or no parameters  
e.g. `void f()`, `f(void)`
- In C `sizeof(void) == 1` (GCC), while in C++ `sizeof(void)` does not compile!!

```
int main() {  
    // sizeof(void); // compile error  
}
```

# Pointer Type

The **type of a pointer** (e.g. `void*`) is an *unsigned* integer of 32-bit/64-bit depending on the underlying architecture

- It only supports the operators `+`, `-`, `++`, `--` and comparisons `==`, `!=`, `<`, `<=`, `>`, `>=`
- A pointer can be *explicitly* converted to an integer type

```
void* x;  
size_t y = (size_t) x; // ok (explicit)  
// size_t y = x;      // compile error (implicit)
```

## nullptr Keyword

C++11 introduces the new keyword `nullptr` to represent null pointers (instead of the `NULL` macro)

```
int* p1 = NULL;      // ok, equal to int* p1 = 0l
int* p2 = nullptr;  // ok, nullptr is a pointer not a number

int n1 = NULL;      // ok, we are assigning 0 to n1
// int n2 = nullptr; // compile error we are assigning
//                               a null pointer to an integer variable

// int* p2 = true ? 0 : nullptr; // compile error
//                               // incompatible types
```

Remember: `nullptr` is not a pointer, but an object of type `nullptr_t` → safer

# Fundamental Types Summary

The *fundamental types*, also called *primitive* or *built-in*, are organized into three main categories:

- Integer
- Floating-point
- `void`

## Any other entity in C++ is

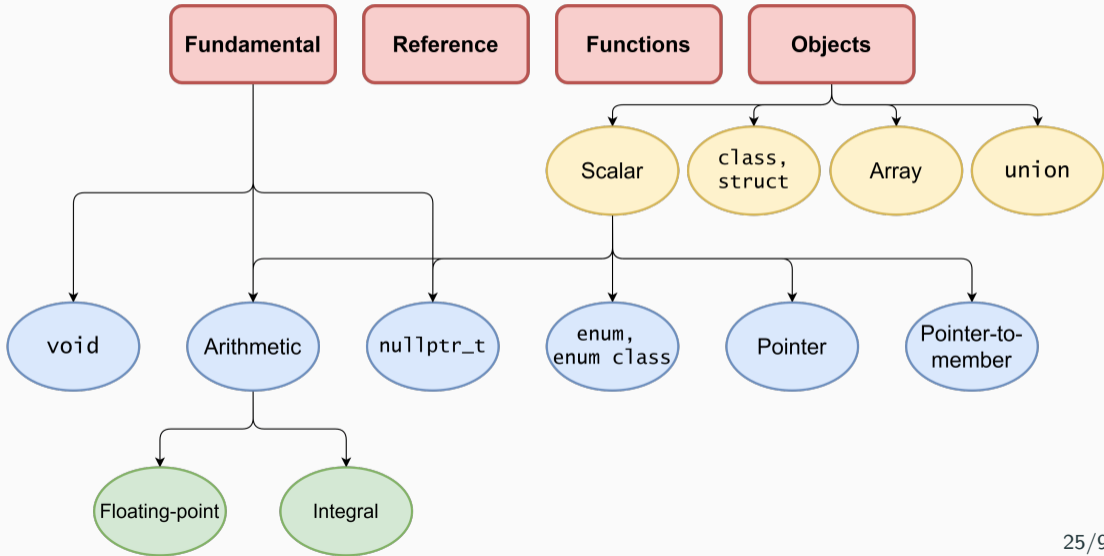
- an *alias* to the correct type depending to the context and the architectures
- a *composition* of builtin types: struct/class, array, union

---

[en.cppreference.com/w/cpp/language/types](https://en.cppreference.com/w/cpp/language/types)

[en.cppreference.com/w/cpp/types/integer](https://en.cppreference.com/w/cpp/types/integer)

# C++ Types Summary



# Conversion Rules

---

**Implicit type conversion rules** (applied in order) :

⊗: any operation (\*, +, /, -, %, etc.)

**(A) Floating point promotion**

`floating_type` ⊗ `integer_type` → `floating_type`

**(B) Implicit integer promotion**

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

`small_integral_type` ⊗ `small_integral_type` → `int`

**(C) Size promotion**

`small_type` ⊗ `large_type` → `large_type`

**(D) Sign promotion**

`signed_type` ⊗ `unsigned_type` → `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  a = b / 2.0; // again a = 3 not 3.5!!
```



# Pointer Conversion

- Any pointer type can be implicitly converted to `void*`
- Non-`void` pointers must be explicitly converted
- `static_cast`<sup>†</sup> is not allowed for pointer conversion for safety reasons, except for `void*`

```
int* ptr1 = ...;
void* ptr2 = ptr1;           // implicit conversion

void* ptr3 = ...;
int* ptr4 = (int*) ptr3;    // explicit conversion, static_cast allowed

int* ptr5 = ...;
char* ptr6 = (char*) ptr5; // explicit conversion, static_cast not allowed
```

---

<sup>†</sup> see next lectures for `static_cast` details

# C++ Operators

---

Precedence	Operator	Description	Associativity
1	a++ a--	Suffix/postfix increment and decrement	Left-to-right
2	++a --a ! ~	Prefix increment/decrement, Logical/Bitwise Not	Right-to-left
3	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
4	a+b a-b	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	Left-to-right
6	< <= > >=	Relational operators	Left-to-right
7	== !=	Equality operators	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR	Left-to-right
10		Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12		Logical OR	Left-to-right

- **Unary** operators have higher precedence than **binary operators**
- **Standard math operators** (+, \*, etc.) have higher precedence than **comparison**, **bitwise**, and **logic** operators
- **Comparison** operators have higher precedence than **bitwise** and **logic operators**
- **Bitwise** operators have higher precedence than **logic** operators
- **Compound assignment** operators +=, -=, \*=, /=, %=, ^=, !=, &=, >>=, <<= have lower priority
- The **comma** operator has the lowest precedence (see next slides)

Examples:

```
a + b * 4;           // a + (b * 4)
```

```
a * b / c % d;      // ((a * b) / c) % d
```

```
a + b < 3 >> 4;     // (a + b) < (3 >> 4)
```

```
a && b && c || d;     // (a && b && c) || d
```

```
a | b & c || e && d; // ((a | (b & c)) || (e && d))
```

**Important:** sometimes parenthesis can make expression worldly... but they can help!

# Prefix/Postfix Increment Semantic

**Prefix Increment/Decrement** `++i` , `--i`

- (1) Update the value
- (2) Return the new (updated) value

**Postfix Increment/Decrement** `i++` , `i--`

- (1) Save the old value
- (2) Update the value
- (3) Return the old (original) value

Prefix/Postfix increment/decrement semantic applies not only to built-in types but also to objects

# Operation Ordering Undefined Behavior

Expressions with undefined (implementation-defined) behavior:

```
int i = 0;
i = ++i + 2;      // until C++11: undefined behavior
                  // since C++11: i = 3

i = 0;
i = i++ + 2;     // until C++17: undefined behavior
                  // since C++17: i = 3

f(i = 2, i = 1); // until C++17: undefined behavior
                  // since C++17: i = 2

i = 0;
a[i] = i++;     // until C++17: undefined behavior
                  // since C++17: a[1] = 1

f(++i, ++i);    // undefined behavior
i = ++i + i++;  // undefined behavior
```

## Assignment, Compound, and Comma Operators

**Assignment** and **compound assignment** operators have *right-to-left associativity* and their expressions return the assigned value

```
int y = 2;  
int x = y = 3; // y=3, then x=3  
           // the same of x = (y = 3)  
if (x = 4)   // assign x=4 and evaluate to true
```

The **comma** operator has *left-to-right associativity*. It evaluates the left expression, discards its result, and returns the right expression

```
int x = (3, 4); // discards 3, then x=4  
int y = 0;  
int z;  
z = y, x;      // z=y (0), then returns x (4)
```



# Structure Binding

*Structure Binding* declaration C++17 binds the specified names to elements of initializer:

```
struct A {  
    int x = 1;  
    int y = 2;  
} a;  
  
auto [x1, y1] = a; // x1=1, y1=2  
  
int b[2] = {1,2};  
auto [x2, y2] = b; // x2=1, y2=2  
  
A f() { return A{4, 5}; }  
auto [x3, y3] = f(); // x3=4, y3=5
```

## Spaceship Operator <=>

C++20 provides the **three-way comparison operator** `<=>`, also called *spaceship operator*, which allows comparing two objects in a similar way of `strcmp`. The operator returns an object that can be directly compared with a positive, 0, or negative integer value

```
(3 <=> 5)      == 0; // false
('a' <=> 'a') == 0; // true

(3 <=> 5)      < 0;  // true
(7 <=> 5)      < 0;  // false
```

The semantic of the *spaceship operator* can be extended to any object (see next lectures) and can greatly simplify the comparison operators overloading

# 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





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

other examples: average, 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

---

<b>Model/Bits</b>	<b>short</b>	<b>int</b>	<b>long</b>	<b>long long</b>	<b>pointer</b>
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 <stdint>
```

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 an 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' !!
```

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)

## 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

int64_t z = 4294967296; // 2^32 ok
// int64_t z1 = 1 << 32; // wrong!! z is (potentially) 0
// (1) signed shift, (2) shift > bits
```

**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
int16_t w = z;    // (int16_t) (x % 2^16)
cout << w;      // print -32767
```

# Implicit Promotion

Integral data types smaller than 32-bit are *implicitly* promoted to `int`, independently if they are *signed* or *unsigned*

- Unary `+`, `-`, `~` and Binary `+`, `-`, `&`, `etc.` promotion:

```
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)
```



```
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!

```
// 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);
}
```

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

```
unsigned x = 1;
unsigned y = x >> 32; // undefined behavior!!
```

*Undefined behavior in implicit conversion*

```
uint16_t a2 = 65535; // 0xFFFF
uint16_t b2 = 65535; // 0xFFFF
cout << (a2 * b2); // print '-131071' (0xFFFE0001)
                // undefined behavior!! (int overflow)
```

*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` ?

# 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

---



## 32/64-bit Floating-Point

**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

IEEE764 technical document:

754-2019 - IEEE Standard for Floating-Point Arithmetic

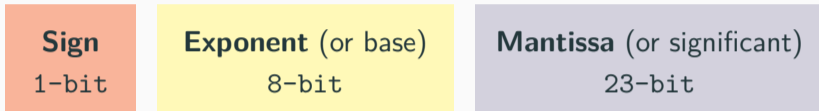
The IEEE Standard 754: One for the History Books

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

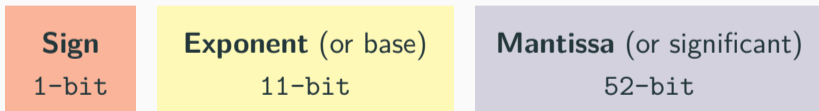
[en.cppreference.com/w/cpp/types/numeric\\_limits/is\\_iec559](http://en.cppreference.com/w/cpp/types/numeric_limits/is_iec559)

## 32/64-bit Floating-Point

- IEEE764 Single precision (32-bit) float

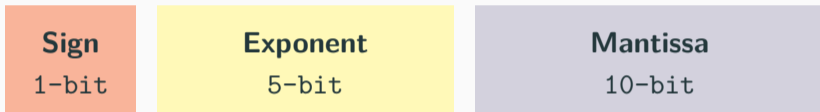


- IEEE764 Double precision (64-bit) double

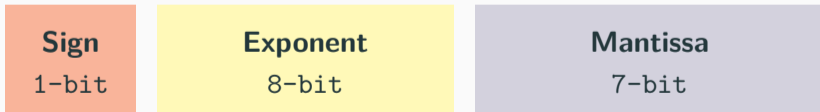


# 16-bit Floating-Point (non-standard)

- IEEE754 16-bit Floating-point (fp16)



- Google 16-bit Floating-point (bfloat16)



## Other Real Value Representations (non-standard)

- **Posit** (John Gustafson, 2017), also called *unum III (universal number)*, represents floating-point values with *variable-width* of exponent and mantissa
- **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

- 
- Beating Floating Point at its Own Game: Posit Arithmetic
  - Comparing posit and IEEE-754 hardware cost

## 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 · 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	110000000000000000000000
+	$2^{(135-127)} = 2^8$	$\frac{1}{2^1} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75 \xrightarrow{\text{normal}} 1.75$

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

## 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

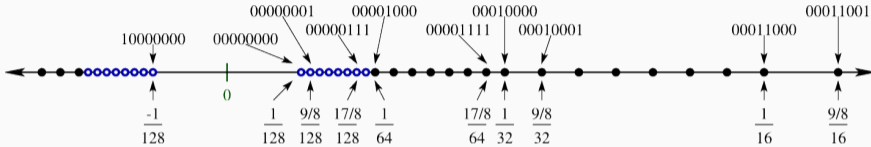
## Denormal number

**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

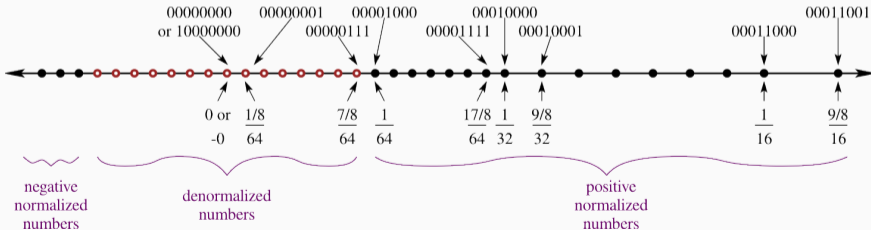
## Why denormal numbers make sense:

(↓ normal numbers)



## The problem: distance values from zero

(↓ denormal numbers)





# Floating-point - Special Values

- $\pm$  infinity

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

- NaN (mantissa  $\neq$  0)

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

- $\pm 0$

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

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

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

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

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

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

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

## 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 :  $\epsilon = 2^{-52} \approx 2.22045 * 10^{-16}$

## ULP

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

$$ULP(p, e) = 1.0 \times \beta^{e-(p-1)}$$

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 Error

Machine floating-point representation of  $x$  is denoted  $\mathbf{fl}(x)$

$$fl(x) = x(1 + \delta)$$

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

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

# Floating-point Summary

	half	bfloat16	float	double
<b>exponent</b>	5-bit [0*-30]	8-bit [0*-254]		11-bit [0*-2046]
<b>bias</b>	15	127		1023
<b>mantissa</b>	10-bit	7-bit	23-bit	52-bit
<b>largest (<math>\pm</math>)</b>	$2^{16}$ 65,536	$2^{128}$ $3.4 \cdot 10^{38}$		$2^{1024}$ $1.8 \cdot 10^{308}$
<b>smallest (<math>\pm</math>)</b>	$2^{-14}$ 0.00006	$2^{-126}$ $1.2 \cdot 10^{-38}$		$2^{-1022}$ $2.2 \cdot 10^{-308}$
<b>smallest (denormal)</b>	$2^{-24}$ $6.0 \cdot 10^{-8}$	/	$2^{-149}$ $1.4 \cdot 10^{-45}$	$2^{-1074}$ $4.9 \cdot 10^{-324}$
<b>epsilon</b>	$2^{-10}$ 0.00098	$2^{-7}$ 0.0078	$2^{-23}$ $1.2 \cdot 10^{-7}$	$2^{-52}$ $2.2 \cdot 10^{-16}$

# Floating-point Type Limits

T: float or double

```
#include <limits>

// Check if the actual C++ implementation adopts
// the IEEE754 standard:
std::numeric_limits<T>::is_iec559;    // should return true

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
```

## 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  $x$

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

## 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`:

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

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

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



# Floating-point - Useful Functions

```
#include <limits>
```

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

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

```
#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 Special Values Behavior

```
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
cout << ((5.0f / inf) == (-5.0f / inf)); // true
cout << (10e40f) == (10e40f + 9999999.0f); // true
cout << (10e40) == (10e40f + 9999999.0f); // false
```

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 (NaN  $\neq$  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**  $(k \otimes a) \otimes a \neq a$

(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
```

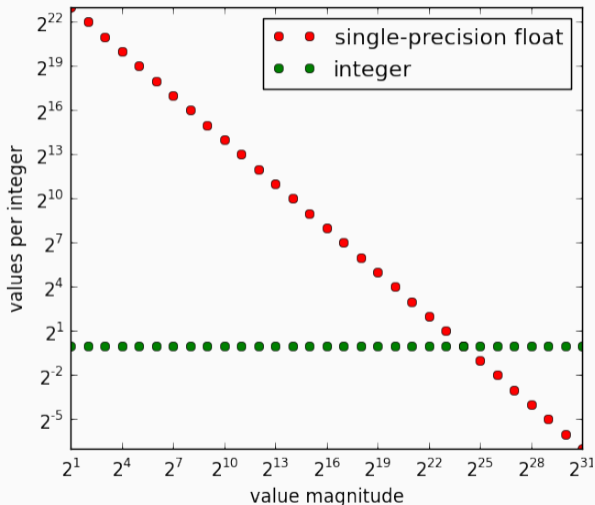
IEEE764 Floating-point computation guarantees to produce **deterministic** output, namely the exact bitwise value for each run, if and only if the **order of the operations is always the same**

→ *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***



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

## 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`?

## 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) {
    const float normal_min      = std::numeric_limits<float>::min();
    const float relative_error = <user_defined>

    if (std::isfinite(a) || isfinite(b)) // a = ±∞, b = ±∞ and 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)
        return true; // or also: user_epsilon * normal_min

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



# 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)

- **multiplication algorithm** (simplified):

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

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

- Recent architectures (also GPUs) provide `fma` to compute these two operations in a single instruction (performed by the compiler)
- The rounding error is lower  $fl(fma(x, y, z)) < fl((x \otimes y) \oplus z)$

## 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:

- (1)  $\mathbf{a} \pm \mathbf{b}$ , 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
- (2)  $\mathbf{a} - \mathbf{b}$ , where  $\mathbf{a} \approx \mathbf{b}$ . Loss of precision in both  $a$  and  $b$ . It implies large relative error

*How many iterations performs the following code?*

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

```
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 !!
```

Let's solve a quadratic equation:

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

$$x^2 + 5000x + 0.25 \quad x_{1,2} = 0.00005, -5000$$

```
(-5000 + std::sqrt(5000.0f * 5000.0f - 4.0f * 1.0f * 0.25f)) / 2
(-5000 + std::sqrt(25000000.0f - 1.0f)) / 2 // !! case 1
(-5000 + std::sqrt(25000000.0f)) / 2
(-5000 + 5000) / 2 = 0 // !! case 2
```

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

# Minimize Error Propagation

- Prefer **multiplication/division** rather than addition/subtraction
- Scale by a **power of two** is safe
- 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
- **Switch to log scale.** Multiplication becomes Add, and Division becomes Subtraction

## 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`

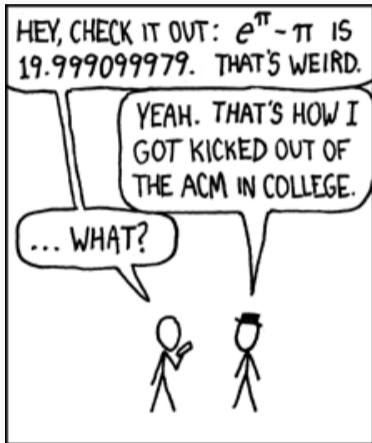
## Floating-point Comparison readings:

- `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 online visualization tool:

`www.h-schmidt.net/FloatConverter/IEEE754.html`

## 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.

