Modern C++ Programming

2. Basic Concepts I

Federico Busato

University of Verona, Dept. of Computer Science

2019, v2.01

Agenda

What compiler should I use?

Popular (free) compilers:

- Microsoft Visual C++ (**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: **Clang**

- Comparable performance with GCC/MSVC and low memory usage [`compilers comparison link`]
- 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.
- Easy to install: `releases.llvm.org`

C++ compiler features support: `en.cppreference.com/w/cpp/compiler_support`

Install the compiler

Install the last gcc/g++ (v8)

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install gcc-8 g++-8
$ gcc-8 --version
```

Install the last clang/clang++ (v7)

```
$ wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key \
    | sudo apt-key add -
$ sudo apt-add-repository \
    "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-6.0 main"
$ sudo apt update
$ sudo apt install -y clang-7.0
$ clang-7.0 --version
```

What editor/IDE compiler should I use?

Popular C++ IDE (Integrated Development Environment) and editors:

- **Microsoft Visual Studio**. (free, Windows)
- **QT-Creator** (link). Fast (written in C++), simple
- **Clion** (link). (free for student). Powerful IDE with a lot of options
- **Atom** (link). Standalone editor oriented for programming (GitHub). Many useful plugins and modular
- **Sublime Text editor** (link). Stand-alone editor oriented to programming
- **XCode**, **Eclipse** (**Cevelop**, `www.cevelop.com`), **Vim**, etc.

Not suggested:

- `Notepad, Gedit, and other similar editors`
  Lack of support for programming

How to compile?

Compile C++11, C++14, C++17 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
```

Compiler version and C++ Standard:

Compiler	C++11		C++14		C++17	
	Core	Library	Core	Library	Core	Library
`g++`	4.8.1	5.1	5.1	5.1	7.1	ongoing
`clang++`	3.3	3.3	3.4	3.5	5.0	ongoing
`MSVC`	19.0	19.0	19.10	19.0	19.14	19.14+

en.cppreference.com/w/cpp/compiler_support

Hello World

Hello World

C code with `printf`:

```c
#include <stdio.h>

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

`printf` prints on standard output

C++ code with streams:

```cpp
#include <iostream>

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

`cout` : represent the standard output stream

Hello World

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

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

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

I/O Stream (std::cout)

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

C:
```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++:
```cpp
#include <iostream>

int main() {
    int    a = 4;
    double b = 3.0;
    char*  c = "hello";
    std::cout << a << " " << b << " " << c << "\n";
}
```

I/O Stream (Why should we prefer I/O stream?) 2/3

- **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`)

I/O Stream (Common C errors) 3/3

- Forget the number of parameters:

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

- Use the wrong format:

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

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

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

C++ Primitive Types

Builtin Types

Type	Size (bytes)	Range	Fixed width types
`bool`	1	true, false	
`char` [†]	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, † one-complement

Builtin Types

- C++ provides also `long double` (no IEEE-754) of size 8/12/16 bytes depending on the implementation

- **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`, `union`, etc.

- Interesting: C++ does not explicitly define the size of a byte

Other Data Types

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

- C++ does not provide support for **half float** (16-bit) 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
  - There is a proposal (next standard) since 2016
  - Software support (OpenGL, Photoshop, Lightroom, `half.sourceforge.net`)

Builtin 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`

en.cppreference.com/w/cpp/language/types

en.cppreference.com/w/cpp/types/integer

Builtin Types (suffix and prefix)

Builtin types suffix:

Type	Suffix	example
`int`	`NO PREFIX`	`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`

Builtin types representation prefix:

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

Fixed Width Integers

C++ provides fixed width integer types. They have the same size on any architecture (`#include <cstdint>`)

int8_t, uint8_t,

int16_t, uint16_t,

int32_t, uint32_t,

int64_t, uint64_t

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

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

Fixed Width Integers

`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`)

```
#include <cstddef>
void f(int8_t x)  {}
void f(int16_t x) {}
void f(int32_t x) {}
void f(int64_t x) {}
int main() {
    int x = 0;
    f(x); // compile error!! under 32-bit ARM GCC
} // "int" is not mapped to int*_t type in this (very) particular case
```

Full Story: `ithare.com/c-on-using-int_t-as-overload-and-template-parameters`

Pointer type and `size_t`

The **type of a pointer** (e.g. `void*`) is an unsigned integer of 32-bit/64-bit depending on the underlying architectures. It only supports the operators `+, -, ++, --` and comparisons `==, !=, <, <=, >, >=`

`size_t`

`size_t` is a data type capable of storing the biggest representable value on the current architecture (defined in `<cstddef>`)

- `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

Conversion Rules

Implicit type conversion rules (applied in order) :

$\otimes$: any operations (*, +, /, -, %, etc.)

(a) Floating point promotion

`floating_type` $\otimes$ `integer_type` = `floating_type`

(b) Size promotion

`small_type` $\otimes$ `large_type` = `large_type`

(c) Sign promotion

`signed_type` $\otimes$ `unsigned_type` = `unsigned_type`

Common Errors

- 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!!
```

- Integer type are more accurate than floating types 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 !!!
```

Implicit Conversions

- Unary `+, -, ~` promotion:

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

- Binary `+, -, &, etc.` promotion:

```
unsigned char a = 255;
unsigned char b = 255;
cout << (a + b);  // print '510' (no overflow)

unsigned short a = 65535;
unsigned short b = 65535;
cout << (a + b);  // print '131070' (no overflow)
```

When Use Signed/Unsigned Integer? 1/3

`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 ($\mathbb{Z}$)
- overflow/underflow is undefined
- discontinuity in -2^{31}, $2^{31} - 1$
- bit-wise operations are implementation-defined

`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

When Use Signed/Unsigned Integer? 2/3

Common errors:

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

☠ Segmentation fault!

```
int f(int a, unsigned b, int* array) {
    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!

When Use Signed/Unsigned Integer? 3/3

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

Overflow/Underflow

Detect overflow/underflow for floating point types is **easy** ($\pm$`inf`)

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

```
bool isAddOverflow(unsigned a, unsigned b) {
    return (a + b) < a || (a + b) < b;
}

bool isMulOverflow(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 perform the operation

`void` Type

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

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

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

nullptr Keyword

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

```
int* p1 = NULL;    // ok, equal to int* p1 = 0
int* p2 = nullptr; // ok

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

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

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

auto Keyword

The auto keyword (C++11) specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;   // 1 is int, 2 is int, 1 + 2 is int!
//    -> 'a' must be int
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double
//    -> 'b' must be double
```

auto keyword may be very useful for maintainability.

```
for (auto i = k; i < size; i++)
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Note: `auto x = 0;` in general makes no sense (`x` is int)

Builtin type limits

Query properties of arithmetic types C++11:

```cpp
#include <limits>

std::numeric_limits<int>::max();        // 2^31 − 1
std::numeric_limits<float>::max();      // 3.4 ∗ 10^38

std::numeric_limits<int>::min();        // −2^31
std::numeric_limits<float>::min();      // 1.17 ∗ 10^−38    !!!

std::numeric_limits<int>::lowest();     // −2^31
std::numeric_limits<float>::lowest();   // −3.4 ∗ 10^38

std::numeric_limits<float>::infinity(); // inf
```

Floating-point Arithmetic

Floating-Point

In general, C/C++ adopt IEEE754 floating-point standard

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

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

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

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

Floating-Point (Exponent Bias)

Exponent Bias

In IEEE 754 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 $[-126, +127]$

- Example

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

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

Floating-Point (Normal/Denormal)

Normal number

A **normal** number is a floating point number that can be represented without *leading zeros* in its mantissa (one in the first left position) and at least one bit set in the exponent

Denormal number

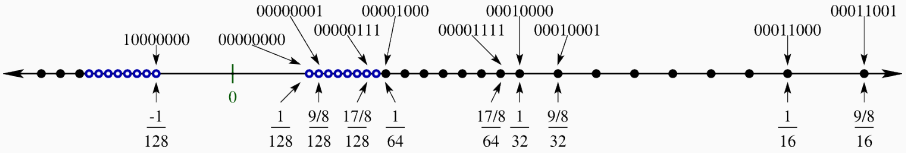
Denormal (or subnormal) numbers fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest normal number is denormal

If the exponent is all 0s, but the mantissa is non-zero (else it would be interpreted as zero), then the value is a denormal number

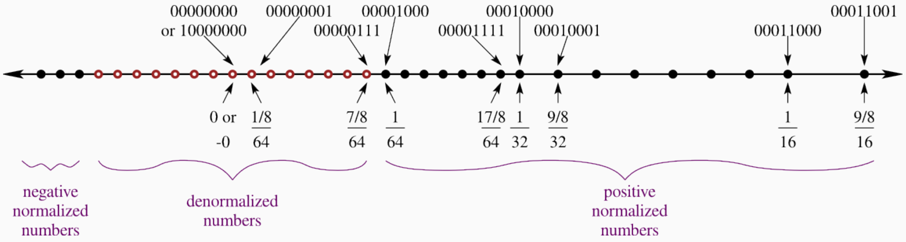
Floating point online tool:
`www.h-schmidt.net/FloatConverter/IEEE754.html`

Floating-Point (Normal/Denormal)

Why denormal numbers make sense: (↓ normal numbers)



The problem: distance values from zero (↓ denormal numbers)



reference: `www.toves.org/books/float/`

Floating-Point (Special Values)

- $\pm$ infinity

*	11111111	00000000000000000000000

- `NaN` (mantissa $\neq 0$)

*	11111111	***********************

- ± 0

*	00000000	00000000000000000000000

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

*	00000000	***********************

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

*	00000001	00000000000000000000000

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

*	11111110	11111111111111111111111

`NaN` Properties

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
- $\pm\infty \mp \infty$
- $0 \cdot \infty$
- $0/0, \infty/\infty$
- $\sqrt{x} \mid x < 0$
- $\log(x) \mid x < 0$
- $\sin^{-1}(x), \cos^{-1}(x) \mid x < -1 \textit{ or } x > 1$

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

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

Floating-Point Issues

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 commutative, but not associative and not reflexive (see `NaN`) !!

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

Floating-point computation guarantee 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**

Floating-Point Issues

Floating point type has special values (C++11):

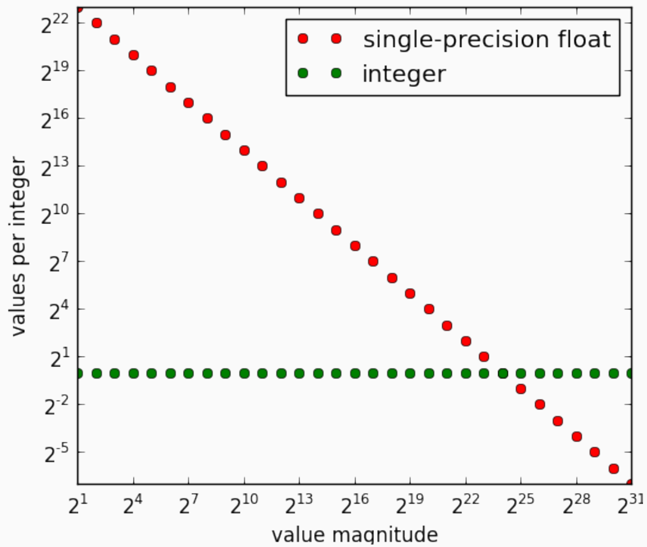
```
#include <limits>
std::numeric_limits<float>::infinity;  // float infinity
std::numeric_limits<float>::quiet_NaN; // float NaN
```

```
#include <cmath>
INFINITY // float infinity
NAN      // float NaN
```

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

cout << (-0.0 == 0.0); // true
cout << 5.0 / 0.0;      // print "inf"
cout << -5.0 / 0.0;     // print "-inf"
cout << ((5.0 / 0.0) == ((5.0 / 0.0) + 9999999.0);  // true
cout << ((5.0f / INFINITY) == ((-5.0f / INFINITY)); // true
```

Floating-Point Granularity



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

Floating-Point Granularity 2/2

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);
```

Floating-Point - Useful Functions

where T is a numeric type C++11

```
#include <cmath>

bool isnan(T value) // returns true if value is nan, false otherwise
bool isinf(T value) // returns true if value is ±inf, false otherwise
bool isfinite(T value) // returns true if value is not nan or infinite,
                       // false otherwise
bool isnormal(T value); // true if normal, false otherwise

T ldexp(T x, p)     // multiplies a number by 2 raised to a power.
                    // returns x * 2^p
int ilogb(T value) // extracts exponent of the number

#include <limits>
// Check if the actual C++ implementation adopts the IEEE754 standard:
std::numeric_limits<float>::is_iec559;  // should return true
std::numeric_limits<double>::is_iec559; // should return true
```

Floating-Point Comparison

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`

Floating-Point Comparison

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) {
    const float epsilon = <user_defined>

    if (a == b) // a=0,b=0 and a = ±∞, b = ±∞
        return true;
    if (std::isnan(a) || std::isnan(b)) // optional
        return false;

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

References:

[1] floating-point-gui.de/errors/comparison

[2] www.cygnus-software.com/papers/comparingfloats

Floating-Point Comparison

Relative errors is susceptible to the scale of the computation

Suppose you have two non-trivial computations. At the end you get `0.0` and `0.5`

- if the scale of the computation is small e.g. $[0.0, 1.0]$ the relative error is high
- if the scale of the computation is large e.g. $[-100.0, 100.0]$ the relative error is small

Many times it is not possible to compare the result with the **true** value obtained by using other methods (symbolic computation, external sources, etc.)

Floating-Point (In)Accuracy

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 : $\varepsilon = 1.17549435 * 10^{-38}$

```
#include <limits>
T std::numeric_limits<T>:: epsilon()  // returns the machine epsilon
```

Truncation error

A number x that is **Truncated** (or *Chopped*) at the m^{th} digit means that all $n - m$ digits after the n^{th} digit are removed.

- Machine floating-point representation of x is denoted **fl(x)**

The relative error as a result of truncation is

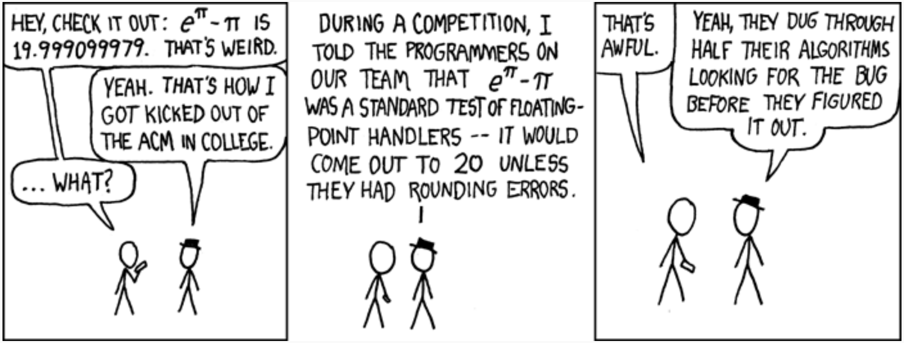
$$\left|\frac{fl\left(x\right)-x}{x}\right| \leq \frac{1}{2}\varepsilon \qquad fl\left(x\right)=x\left(1+\delta\right) \qquad \left|\delta\right| \leq \frac{1}{2}\varepsilon$$

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 (maybe 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 reading:
D. Golberg, *"What Every Computer Scientist Should Know About Floating-Point Arithmetic*, 1991, link

Minimize Error Propagation


HEY, CHECK IT OUT: $e^{\pi} - \pi$ IS 19.999099979. THAT'S WEIRD.
YEAH. THAT'S HOW I GOT KICKED OUT OF THE ACM IN COLLEGE.
... WHAT?
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.
THAT'S AWFUL.
YEAH, THEY DUG THROUGH HALF THEIR ALGORITHMS LOOKING FOR THE BUG BEFORE THEY FIGURED IT OUT.

Enumerators

Enumerated Types

Enumerator

An **enumerator** (`enum`) is a data type that groups a set of named integral constants

```
enum color_t { BLACK, BLUE, GREEN = 2 };

color_t color = BLUE;
cout << (color == BLACK); // print false
```

The problem:

```
enum color_t { BLACK, BLUE, GREEN };
enum fruit_t { APPLE, CHERRY };

color_t color = BLACK;    // int: 0
fruit_t fruit = APPLE;    // int: 0
cout << (color == fruit); // print 'true'!!
// and, most importantly, does the match between a color and
// a fruit makes any sense?
```

Enumerated Types (Strongly Typed)

enum class

C++11 introduces a *type safe* enumerator `enum class` (scoped enum) data type that are not implicitly convertible to `int`

Syntax: `<enum_class>::<enum_value>`

```
enum class color_t { BLACK, BLUE, GREEN = 2 };
enum class fruit_t { APPLE, CHERRY };

color_t color = color_t::BLUE;
fruit_t fruit = fruit_t::APPLE;

// cout << (color == fruit); // compile error!!
//     we are trying to match colors with fruits
//     BUT, they are different things entirely

// int a = color_t::GREEN;  // compile error!!
```

Enumerated Types Features

- Strongly typed enumerators can be compared

```cpp
enum class Colors { RED = 1, GREEN = 2, BLUE = 3 };

cout << (Colors::RED < Colors::GREEN); // print true
```

- Strongly typed enumerators do not support other operations

```cpp
enum       WColors { RED = 1, GREEN = 2, BLUE = 3 };
enum class SColors { RED = 1, GREEN = 2, BLUE = 3 };

int v = RED + GREEN; // ok
// int v = SColors::RED + SColors::GREEN; // compile error!
```

- The size of `enum class` can be set

```cpp
#include <cstdint>
enum class Colors : int8_t { RED = 1, GREEN = 2, BLUE = 3 };
```

Enumerated Types Features

- Strongly typed enumerators can be converted

```
int a = (int) color_t::GREEN; // ok
```

- Enum class objects should be always initialized

```
enum class SColors { RED = 1, GREEN = 2, BLUE = 3 };

int main() {
    SColors my_color;  // "my_color" maybe 0!!
}
```

- Enum (class) objects are automatically enumerated

```
enum class SColors { RED, GREEN = -1, BLUE, BLACK };
//                   (0)  (-1)        (0)   (1)
int main() {
    SColors::RED == SColors::BLUE; // true
}
```

Enumerated Types Features (C++17) 3/3

- Cast from *out-of-range values* to enum object leads to undefined behavior (C++17)

```cpp
enum Colors { RED = 0, GREEN = 1, BLUE = 2 };

int main() {
    Colors value = (int) 3; // undefined behavior
}
```

- C++17 Enum class objects support *direct-list-initialization*

```cpp
enum class Colors { RED = 0, GREEN = 1, BLUE = 2 };

int main() {
    Colors a{2};          // ok, equal to Colors:BLUE
//  Colors b{4};          // compile error!!
//  Colors c = {2};       // compile error!!
    Colors d = Colors{2}; // ok, equal to Colors:BLUE
}
```

Union and Bitfield

Union

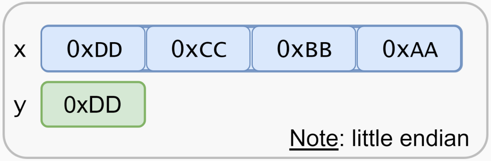
Union

A **union** is a special data type that allows to store different data types in the same memory location

- The `union` is only as big as necessary to hold its *largest* data member
- The `union` is a kind of *"overlapping"* storage

```
union A {
  int  x;
  char y;
};
```

```
A a;
A.x = 0xAABBCCDD
```



Union

```
union A {
    int  x;
    char y;
};  // sizeof(A): 4

A a;
a.x = 1023;        // bits: 00..000001111111111
a.y = 0;           // bits: 00..000001100000000
std::cout << a.x; // print 512 + 256 = 768
```

C++17 introduces `std::variant` to represent a type-safe union

Bitfield

Bitfield

A **bitfield** is variable of a structure with a predefined bit width.

A bitfield can hold bits instead byte

```
struct S1 {
    int b1 : 10;  // range [0, 1023]
    int b2 : 10;  // range [0, 1023]
    int b3 : 8;   // range [0, 255]
}; // sizeof(S1): 4 bytes

struct S2 {
    int b1 : 10;
    int    : 0;   // reset: force the next field
    int b2 : 10;  //        to start at bit 32
}; // sizeof(S1): 8 bytes
```

`using` and `decltype`

using and decltype

- In C++11, the using keyword has the same semantics of typedef specifier (alias-declaration), but with better syntax

```
typedef int distance_t;  // equal to:
using distance_t = int;
```

- In C++11, decltype captures the type of an object or an expression

```
int a = 3;
decltype(a) b = 5;              // 'b' is int
decltype(2.0f) c = 3.0f;        // 'c' is float
decltype(a + 2.0f) d = 3.0f;    // 'd' is float
decltype(f(a)) e = ...;         // 'e' depends on f(a)

using T = decltype(a);          // T is int
T value = 3;
```

Math Operators

Math Operators (Precedence) 1/3

Precedence	Operator	Description	Associativity
1	`a++` `a--`	Suffix/postfix increment and decrement	Left-to-right
2	`++a` `--a`	Prefix increment and decrement	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	$\ll$ $\gg$	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	ˆ	Bitwise XOR	Left-to-right
10	|	Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

Math Operators (Precedence)

In general:

- **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

Full table

`en.cppreference.com/w/cpp/language/operator_precedence`

Math Operators (Precedence) 3/3

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!

Undefined Behavior

Expressions with undefined (implementation-defined) behavior:

```cpp
int i = 0;
i = ++i + 2;       // undefined behavior until C++11,
// otherwise i = 3
i = 0;
i = i++ + 2;       // undefined behavior until C++17,
// modern compilers (clang, gcc): i = 3

f(i = 2, i = 1);   // undefined behavior until C++17
// modern compilers (clang, gcc): i = 2
i = 0;
a[i] = i++;        // undefined behavior until C++17
// modern compilers (clang, gcc): a[1] = 1

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

n = ++i + i;       // undefined behavior
```

Statements and Control Flow

Assignment and Ternary Operator

- Assignment special cases:

```
int a;
int b = a = 3;  // (a = 3) return value 3
if (b = 4)      // it is not an error, but BAD programming
```

- *Structure Binding* declaration: C++17

```
struct A {
    int x = 1;
    int y = 2;
} a;

auto [x, y] = a;
cout << x << " " << y;
```

- Ternary operator:

```
<cond> ? <expression1> : <expression2>
```

<expression1> and <expression2> must return a value of the same type

```
int value = (a == b) ? a : (b == c ? b : 3); // nested
```

`if` Statement

- *Short-circuiting*:

```
if (<true expression> || array[-1] == 0)
... // no error!! even though index is -1
// left-to-right evaluation
```

- C++17 `if` statement with *initializer*:

```
void f(int x, int y) {
    if (int ret = x + y; ret < 10)
        cout << "a";
}
```

It aims at simplifying complex statement before the condition evaluation. Available also for `switch` statements

Loops

C++ provides three kinds of loop:

- **for loop**

  ```
  for ([init]; [cond]; [increment]) {
      ...
  }
  ```

  To use when number of iterations is known

- **while loop**

  ```
  while (cond) {
      ...
  }
  ```

  To use when number of iterations is not known

- **do while loop**

  ```
  do {
  ...
  } while (cond);
  ```

  To use when number of iterations is not known, but there is at least one iteration

`for` Loop

- C++ allows "in loop" definitions:

```
for (int i = 0, k = 0; i < 10; i++, k += 2)
    ...
```

- Infinite loop:

```
for (;;)
...
```

- Jump statements:

```
for (int i = 0; i < 10; i++) {
    if (<condition>)
        break;    // exit from the loop
    if (<condition>)
        continue; // continue with a new iteration and exec. i++
    return;       // exit from the function
}
```

Range loop

C++11 introduces the **range loop** to simplifies the verbosity of traditional **for** loop constructs. They are equivalent to the **for** loop operating over a range of values

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")    // RAW STRING
    cout << c << " ";    // print: a b c d

int values[] = { 3, 2, 1 };
for (int v : values)     // ARRAY OF VALUES
    cout << v << " ";    // print: 3 2 1

char letters[] = "abcd";
for (auto c : letters)   // ARRAY OF CHARS
    cout << c << " ";    // print: a b c d
```

Range loop

C++17 extends the concepts of **range loop** for *structure binding*

```
struct A {
    int x;
    int y;
};

A array[10] = { {1,2}, {5,6}, {7,1} };
for (auto [x, y] : array)
    cout << x << "," << y << "  ";  // print: 1,2  5,6  7,1
```

Switch

C++ `switch` can be defined over `int`, `char`, `enum class`, `enum`, `etc.`

```
int f(char x) {
    int y;
    swicth (x) {
        case 'a': y = 1; break;
        default:  return -1;
    }
    return y;
}
```

```
int f(MyEnum x) {
    int y = 0;
    swicth (x) {
        case MyEnum::A:         // fallthrough
        case MyEnum::B:         // fallthrough
        case MyEnum::C: return 0;
        default: return -1;
    }
}
```

Switch

C++17 `[[fallthrough]]` attribute

```
int f(char x) {
    swicth (x) {
        case 'a': x++;
                  [[fallthrough]]; // C++17: avoid warning
        case 'b': return 0;
        default:  return -1;
    }
}
```

Switch scope:

```
int x = 1;
swicth (1) {
    case 0: int x;       // nearest scope
    case 1: cout << x;   // undefined!!
    case 2: { int y; }   // ok
//  case 3: cout << y;   // compile error!!
//  case 4: int x;       // compile error!!
}
```

goto

When it is useful:

```
bool flag = true;
for (int i = 0; i < N && flag; i++) {
    for (int j = 0; j < M && flag; j++) {
        if (<condition>)
            flag = false;
    }
}
```

become:

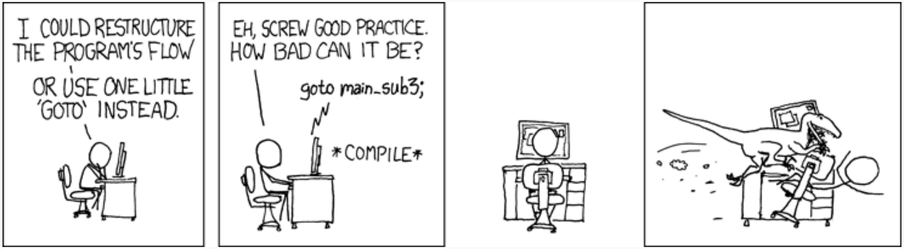
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (<condition>)
            flag = false;
            goto LABEL;
    }
}
LABEL: ;
```

goto

Best solution:

```
bool my_function(int M, int M) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (<condition>)
                return false;
        }
    }
    return true;
}
```

goto


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE