

Modern C++ Programming

2. BASIC CONCEPTS I

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.5



Agenda

- **Before Start**
 - What compiler?
 - What editor/IDE?
 - How to compile?
- **Hello World**
- **I/O Stream**
- **C++ Primitive Types**
 - Built-in types
 - `size_t`, `void`, `auto`, `nullptr`
 - Conversion rules
- **Floating Point**
 - Floating point representation
 - Floating point issues
 - Floating point comparison
 - Overflow/Underflow
- **Strongly Typed Enumerators**
- **Math Operators**
- **Statement and Control Flow**
 - Loop
 - Range Loop
 - Undefined behavior
 - `goto`

What C++ compiler should I use?

Popular (free) compilers:

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

Suggested compiler: **Clang**

- Faster compiles, low memory use, and in general faster code (compared to GCC/MSVC). [[compiler comparison link](#)]
- Expressive diagnostics (examples and propose corrections)
- Strict C++ compliance. GCC/MSVC compatibility (inverse direction is not ensured)
- Includes many very useful tools: memory sanitizer, static code analyzed, automatic formatting, linter, etc.
- Easy to install: releases.llvm.org

Install the compiler

Install the last gcc/g++

```
$ 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++

```
$ 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-6.0
$ clang-6.0 --version
```

What editor/IDE compiler should I use?

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

- **Microsoft Visual C++**. (It does not support all C++ features and it not strictly compliant with the standard)
- **QT-Creator** ([link](#)). Fast (written in C++), simple
- **Clion** ([link](#)). (free for student). Most powerful IDE, but may be slow (written in java) and a lot of options may make it not intuitive
- **Atom** ([link](#)). Standalone editor oriented for programming (GitHub). A lot of useful plugins and modular
- **Sublime Text editor** ([link](#)). Standalone editor oriented for programming. Faster than Atom, but less complete

Not suggested:

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

How to compile?

Compile C++11 programs:

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

- requires g++ version $\geq 4.8.1$
- requires clang++ version ≥ 3.3

Compile C++14 programs:

```
clang++ -std=c++14 <program.cpp> -o program
```

- requires g++ version ≥ 5
- requires clang++ version ≥ 3.4

Compile C++17 programs:

```
clang++ -std=c++17 <program.cpp> -o program
```

- requires g++ version ≥ 5
- requires clang++ version ≥ 5

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";
}
```

`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 whitespace:

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

C++ Primitive Types

Builtin Types (range)

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*		
long unsigned int	4/8*		
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 Win64 systems, † one-complement

Other Data Types

- C++ provides also `long double` (no IEEE-754) of size 8/12/16 bytes depending on the implementation
- C++17 defines also `std::byte` type to represent a collection of bit (`<cstdint>`). 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, <http://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
long unsigned int	unsigned long
long long unsigned int	unsigned long long

<http://en.cppreference.com/w/cpp/language/types>

<http://en.cppreference.com/w/cpp/types/integer>

Builtin Types (suffix and prefix)

Builtin types suffix:

Type	Suffix	example
int	<u>NO PREFIX</u>	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

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'  
std::cout << var * 2;     // print 100 !!
```

`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 <cstdint>
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
```

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 a typedef
- `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

Implicit type conversion rules (applied in order) :

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

(a) Floating point promotion

`floating_type` ⊗ `integer_type` = `floating_type`

(b) Size promotion

`small_type` ⊗ `large_type` = `large_type`

(c) Sign promotion

`signed_type` ⊗ `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 !!!
```

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

Common errors:

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

☠ Segmentation fault!

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

☠ Segmentation fault!

```
// all elements except the last one  
for (size_t i = 0; i < v.size() - 1; i++)  
    ...
```

☠ Segmentation fault!

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** ($\pm\text{inf}$)

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** !!

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

```
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:

```
#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

Floating Point

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

- Single precision (32-bit) (float)

Sign

1-bit

Exponent (or base)

8-bit

Mantissa (or significant)

23-bit

- Double precision (64-bit) (double)

Sign

1-bit

Exponent (or base)

11-bit

Mantissa (or significant)

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

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 number that can be represented without leading zeros in its mantissa (one in the first left position)

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

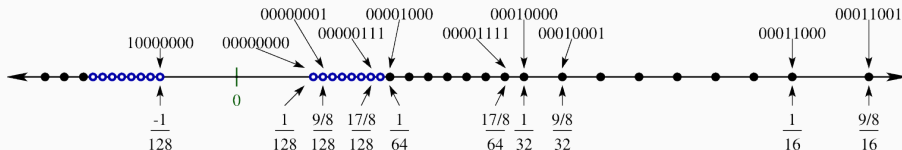
If the exponent is all 0s, but the mantissa is non-zero (else it would be interpreted as zero), then the value is a denormal number

Floating point online tool:

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

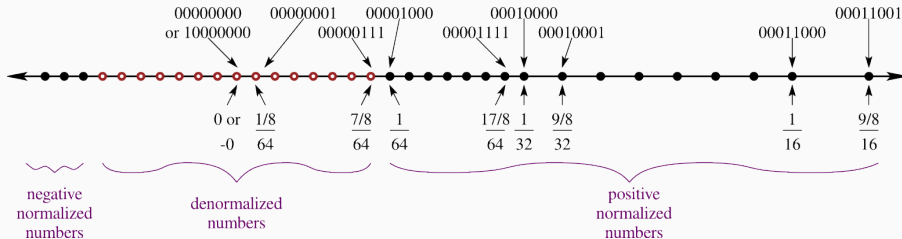
Why denormal numbers make sense:

(↓ normal numbers)



The problem: distance values from zero

(↓ denormal numbers)



Floating Point (Special Values)

- \pm infinity

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

- NaN (mantissa \neq 0)

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

- ± 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
---	----------	--------------------------

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
```

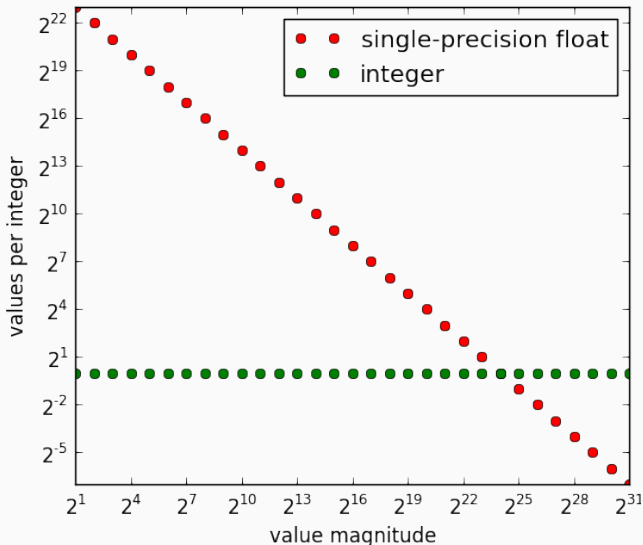
Floating point arithmetic is commutative, but not associative and not reflexive (see NaN) !!

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

Floating point type has special values:

```
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"  
cout << ((5.0 / 0.0) == ((5.0 / 0.0) + 9999999.0)) // print true
```

Floating Point Granularity



Intersection $\approx 16,777,217$

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

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

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

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 * 2p
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
```


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 produce wrong results
- It is not commutative. We always divide by b

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

```
bool areFloatNearlyEqual(float a, float b) {
    const float epsilon = <user_defined>
    float abs_a = std::abs(a);
    float abs_b = std::abs(b);

    if (a == b) // a=0,b=0 and a = ±∞, b = ±∞
        return true;

    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.cygnum-software.com/papers/comparingfloats

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 $\mathbf{fl}(x)$

The relative error as a result of truncation is

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

Minimize Error Propagation

- Prefer **multiplication/division** 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

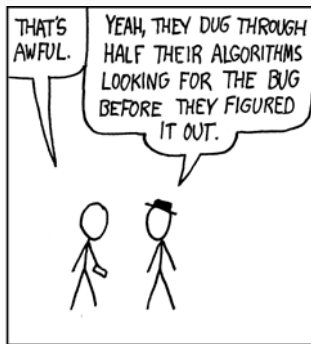
Suggest reading:

D. Golberg, *"What Every Computer Scientist Should Know About Floating-Point Arithmetic*, 1991, [link](#)

Minimize Error Propagation



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.



Enumerators

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 = BLUE;
```

```
fruit_t fruit = APPLE;
```

```
cout << (color == fruit); // generally true, but undefined !!
```

```
// and, most importantly, does the match between a color and
```

```
// a fruit makes any sense?
```


C++11 introduces the `enum class` (scoped enum) data type that are not implicitly convertible to `int`

Type safe enumerator: `enum class`

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!!
```

- Strongly typed enumerators can be compared:

```
enum class Colors { RED = 1, GREEN = 2, BLUE = 3 };  
  
cout << (Colors::RED < Colors::GREEN); // print true
```

- Strongly typed enumerators do not support other operations:

```
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:

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

- 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!!  
}
```

Union and Bitfield

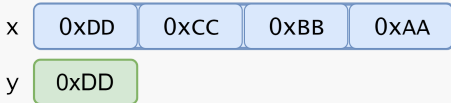
Union

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
```



Note: little endian

```
union A {  
    int x;  
    char y;  
}; // sizeof(A): 4  
  
A a;  
a.x = 1023; // bits: 00..0000011111111111  
a.y = 0; // bits: 00..0000011000000000  
std::cout << a.x; // print 512 + 256 = 768
```

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

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
              // to start at bit 32
    int b2 : 10;
}; // 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

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	<< >>	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

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

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!

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:

```
struct A {  
    int x = 1;  
    int y = 2;  
} a;  
  
auto [x, y] = a;  
std::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)
        std::cout << "a";
}
```

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

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
}
```

C++11 introduces the **range loop** to simplify 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
```

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

```
struct A {  
    int x = 1;  
    int y = 2;  
};  
  
A array[10];  
for (auto [x, y] : array)  
    std::cout << x << " " << y << " ";    // print: 1 2 1 2 ...
```

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 scope:

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

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;
    }
}
```

Undefined Behavior

Expressions with undefined (implementation-defined) behavior:

```
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
```

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: ;
```

Best solution:

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


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

