



C++ Programming

Application to Computer Graphics

[CSC-43043-EP]

2026

Contents

1	Introduction to C++	2
1.1	Preamble	2
1.2	First C++ Program	4
1.3	Declaration of variables	5
1.4	Formatted output and input: printf, scanf	7
1.5	Contiguous-element containers, arrays	8
1.6	Conditionals and Loops	10
1.7	Associative containers: <code>std::map</code>	12
1.8	Variable lifetimes	13
1.9	Functions	14
1.10	Passing Arguments: Copy, Reference	16
1.11	Classes	18
1.12	Writing/reading of external files	21
1.13	Organization of code files	22
1.14	Compilation	24
2	Fundamental types, encoding	27
2.1	Encoding of integers	27
2.2	Encoding of floating-point numbers	28
2.3	Notion of Endianness	29
2.4	Summary of Fundamental Types	29
2.5	Getting the size with <code>sizeof</code>	30
2.6	Important notes	30
2.7	Types with specific sizes	30
2.8	Bitwise operations	31
2.9	Résumé	32
3	Pointers	33
3.1	Concept of storage and memory addressing	33
3.2	Address of a variable	34
3.3	Argument passing	35
3.4	Case of contiguous arrays	37
3.5	Contiguity in classes and structs	40
3.6	Memory Organization AoS vs SoA	41
3.7	Dynamic allocation	43
3.8	Memory copy: <code>memcpy</code>	49
3.9	The generic pointer <code>void*</code>	51
3.10	References	54
4	Classes	58
4.1	Introduction	58
4.2	Initialization, Constructors	60
4.3	Operators	62
4.4	Inheritance	64
4.5	Polymorphism	67
4.6	Access control: <code>const</code>	70
4.7	Access control: the <code>static</code> keyword in classes	72
4.8	Namespaces (<code>namespace</code>)	74
5	Threads and Parallelism	77
5.1	Concept of a thread	77
5.2	Creating a thread in C++	77
5.3	Example of parallel execution	77
5.4	Argument passing to threads	78
5.5	Multiple threads and real parallelism	79

5.6	Shared memory	79
5.7	Synchronization and critical sections	79
5.8	Atomic variables	80
6	Generic programming, templates	81
6.1	General principle of templates	81
6.2	Compilation Principles: duck typing, instantiation, and header files	83
6.3	Static metaprogramming	85
6.4	Type deduction in templates	87
6.5	Template specialization	89
6.6	Priority between specialization and overloading	93
6.7	Aliases	95
7	Development methodologies and best practices	98
7.1	Code quality: concrete objectives	98
7.2	General principles: KISS, DRY, YAGNI	98
7.3	Invariants, assertions et contrat de fonction	100
7.4	Tests and Test-Driven Development (TDD)	104
7.5	Guided example: unit tests for <code>clamp</code>	106
7.6	Test-Driven Development (TDD)	107
7.7	TDD Example: normalization of a 3D vector	107
7.8	Error Handling: Principles and Methodology	109
7.9	Best practices for API design	112

1 Introduction to C++

1.1 Preamble

The C++ language, created in the early 1980s by researcher Bjarne Stroustrup at Bell Labs, was initially introduced as an extension of the C language with which it is intrinsically linked. The C language is a "low-level" language, being close to hardware (processor, memory) particularly suited for coding efficient applications related to the operating system. The C++ language was introduced to preserve the possibilities of the C language, while extending it with structuring and abstraction mechanisms for the description of large-scale software.

C++ stands out from other programming languages by its unique ability to combine **low-level performance** and **high-level abstraction**. Direct heir of C, it allows precise control of memory and hardware, indispensable in areas where efficiency is critical (embedded systems, scientific computing, game engines, etc.). Unlike languages such as Python or Java, which rely on a virtual machine or an interpreter that adds a step of indirection during execution, C++ is a compiled language that produces optimized machine code directly read and executed by the processor, thus guaranteeing very fast execution.

Another major peculiarity of C++ is its support for multiple programming paradigms, called **programming paradigms**:

- **Procedural**, inherited from C, for a classic approach based on functions and control structures.
- **Object-oriented**, introduced with classes, encapsulation, inheritance and polymorphism, facilitating modular software design.
- **Generic**, thanks to the **templates** (type-parametric generics), which allow writing reusable code independent of types.
- **Functional**, increasingly present since C++11 with the **lambdas** (anonymous functions) and the algorithms of the standard library.

This mix of paradigms today makes C++ a language recognized as extremely flexible, capable of adapting to a wide variety of contexts. It remains indispensable for domains where performance and fine-grained memory management are essential, such as game engines, embedded software, numerical simulation, high-performance computing or finance.

Evolutions of C++

The C++ language continues to incorporate regular evolutions.

- **C++98** and **C++03** have standardized the language and its standard libraries.
- **C++11**, called "modern C++", marked an important turning point with, notably, the arrival of range-based loops, easier initialization of structs, the keyword `auto` (type deduction), the appearance of smart pointers (e.g. `std::unique_ptr`, `std::shared_ptr`) and lambda functions (anonymous functions).
- **C++14** and **C++17** enriched the syntax and the standard library (structured bindings, filesystem, parallelism).
- **C++20** brought in the principles of **concepts** (constraints for templates), the **coroutines** (functions whose execution can be suspended and resumed) and the **ranges** (operations on sequences).
- **C++23** continues this modernization, refining the libraries and simplifying the use of the language.

Why use C++?

C++ is currently one of the indispensable languages when it comes to designing applications with high performance, real-time, or compute-intensive constraints.

Application domains

- **Scientific and real-time applications**: physical simulations, numerical computations, embedded systems.

- **Game engines (Game Engines):** Unity, Unreal Engine, Godot, as well as virtually all AAA games heavily use C++.
- **2D/3D software:** Maya, Blender, Photoshop, Premiere Pro, Catia, SolidWorks rely largely on C++.
- **Parallel computing and GPU:** CUDA (NVIDIA) is based on C++.
- **Deep Learning and Vision frameworks:** PyTorch, TensorFlow, OpenCV rely on C++ cores to optimize performance.
- **Operating systems:** Windows is largely written in C and C++.
- **Web and large-scale services:** browsers (Chrome, Firefox) and critical infrastructures (AWS, Facebook, etc.) use C++ for core high-performance parts.

Strengths (+)

- **Performance:** direct compilation to machine code, very fine-grained optimizations possible.
- **Robustness:** a mature language, used and tested at very large scales.
- **High and low level:** a rare combination that lets you write code close to hardware as well as use modern high-level abstractions.
- **Specificity:** this duality is present mainly in C++ (and more recently Rust).
- **Freedom of programming:** support for multiple paradigms (procedural, object-oriented, generic, functional).
- **C compatibility:** ability to reuse the vast C ecosystem.

Weaknesses (-)

- **Complexity:** the language's richness and the multiplicity of paradigms can be hard to master.
- **Memory management:** manual memory management is a source of complexity and major programming errors.
- **Build/compile chain:** compilation is heavier and sometimes slower than in other modern languages.

Quick comparison with other languages

• C++ vs Java

Both are object-oriented, but their philosophies differ.

- C++ is compiled to native machine code, which makes it very fast and suitable for systems where every calculation cycle counts.
- Java runs on a virtual machine (JVM), which facilitates portability but adds a layer of abstraction.
- Java manages memory automatically via a **garbage collector**, while C++ leaves fine-grained control over allocation and deallocation to the programmer.

• C++ vs Python

Python is renowned for its ease of writing and rapid development, but remains an interpreted language, thus much slower in execution.

- C++ demands more rigor and syntax, but allows reaching maximum performance.
- In practice, Python is often used for prototyping, scripting, and data analysis, while C++ is preferred for performance-critical parts (3D engines, scientific computation, simulations).
- The two languages are sometimes used together: Python as a high-level layer, C++ for compute modules.

- **C++ vs Rust**

Rust is a newer language (2010), designed to offer the same efficiency as C++ but with safer memory management.

- Rust eliminates any possibility of leaks or illegal memory access thanks to its system of **borrowing and ownership**.
- C++ offers more flexibility and has a vast existing software ecosystem, but at the cost of the rigor required to avoid errors and security vulnerabilities.
- Rust is seen as a modern and secure alternative, but C++ remains today broadly dominant in industry and in available libraries.

1.2 First C++ Program

We consider the following C++ program:

```
// bibliothèque standard pour les entrées/sorties
#include <iostream>

int main() {
    // affichage 'dun message sur la ligne de commande
    std::cout << "Hello, world!" << std::endl;

    // fin du programme
    return 0;
}
```

Line-by-line explanations

1. `#include <iostream>`

- This directive tells the compiler to include the standard **iostream** library, which allows the use of input and output streams (`std::cin`, `std::cout`, etc.).

2. `int main()`

- It is the main function of the program.
- Every C++ program must have a `main` function.
- Its execution always begins here.
- The word `int` indicates that the `main` function returns an integer to the operating system (0 on success, another value on error).

3. `std::cout << "Hello, world!" << std::endl;`

- `std::cout` is the standard output stream (usually the screen).
- The `<<` operator sends data into the stream.
- "Hello, world!" is a string.
- `std::endl` inserts a newline and forces immediate display of the output.

4. `return 0;`

- Indicates that the program has terminated successfully.
- The returned value is passed back to the system.

Note. Each instruction ends with a semicolon ";" in C++. Indentation and line breaks are optional; they are useful for readability but do not change the program's structure.

First compilation (on Linux/macOS)

To transform the C++ source file (for example `hello.cpp`) into an executable, you use a **C++ compiler**. On Linux or macOS, the most common compilers are:

- **g++** (GNU C++ Compiler, part of GCC)
- **clang++** (C++ compiler developed as part of the LLVM project)

Suppose the file is named `hello.cpp`. Type the command in the terminal in the directory containing the `hello.cpp` file

```
g++ hello.cpp -o hello
```

- `g++` : runs the C++ compiler.
- `hello.cpp` : source file to compile.
- `-o hello` : option that indicates the name of the produced executable (`hello`).

Execution of the program is performed with the command

```
./hello
```

Which should display the following result

```
Hello, world!
```

1.3 Declaration of variables

In C++, a **variable** is a region of memory that contains a value and is identified by a name.

Each variable has a **type** that defines the nature of the values it can contain (integers, floating-point numbers, text, etc.).

Simple example

```
#include <iostream>
#include <string>

int main() {
    int age = 20;           // entier
    float taille = 1.75f;  // nombre à virgule (simple précision)
    double pi = 3.14159;   // nombre à virgule (double précision)
    std::string nom = "Alice"; // chaîne de caractères

    std::cout << "Nom : " << nom << std::endl;
    std::cout << "Age : " << age << std::endl;
    std::cout << "Taille : " << taille << " m" << std::endl;
    std::cout << "Valeur de pi : " << pi << std::endl;

    return 0;
}
```

Fundamental types

You will mainly use two fundamental types in your code:

- **int**: integer. On our machines, an `int` is encoded on **4 bytes**.

```
int entier = 325;
```

- **float**: floating-point number, referred to as "single precision". Encoded on **4 bytes**.

```
float reel = 3.2f;
```

You will also encounter the following types:

- **bool**: boolean value (`true` or `false`). Introduced by C++ (absent from C), it makes the code more readable than an integer.

```
bool estEtudiant = true;
```

- **double**: floating-point number with "double precision", encoded on **8 bytes**.

```
double pi = 3.14159;
```

By default, a decimal number without a suffix is interpreted as a **double**.

> In our context, we will more often use **float** to stay compatible with the graphics card.

- **char**: character (1 byte). The mapping between values and characters is given by the **ASCII table**.
“cpp

`char initiale = 'A'`; “`achar`” can also be used to directly manipulate memory at the byte level.

Important Notes

1. Integer division vs floating-point division

When dividing two integers, the result is **truncated** (Euclidean division):

```
int a = 5 / 2; // equals 2
int b = 5 % 2; // equals 1 (remainder of the division)
```

To obtain a decimal result, at least one of the operands must be floating-point:

```
float c = 5 / 2.0f; // 2.5
float d = 5.0f / 2; // 2.5
float e = float(5) / 2; // 2.5
```

2. The `auto` keyword

It allows the compiler to automatically deduce the type:

```
auto a = 5; // int
auto b = 8.4f; // float
auto c = 4.2; // double
```

[Note] For simple types, it is preferable to explicitly specify the type for better readability. `auto` is mainly useful for generic functions or complex types.

Declaration without initialization (example)

```
int compteur; // not initialized compteur = 10; // assignment of a value later
```

[Warning]: an uninitialized variable contains an undefined value and should not be used before assignment.

Constant variables (`const`)

In C++, a variable can be declared **constant** using the keyword `const`. Such a variable must be **initialized at the time of declaration** and **cannot be modified** thereafter.

```

const int joursParSemaine = 7;
const float pi = 3.14159f;

int main() {
    std::cout << "Pi = " << pi << std::endl;
    // pi = 3.14; // ERROR: cannot modify a constant
    return 0;
}

```

Benefits

- Guarantees that the value will not be accidentally modified in the code.
- Makes the program **more readable** and **safer**.
- Can allow the compiler to optimize certain expressions.

Type conversions (cast)

In C++, it is common to convert a value from one type to another: this is called a **cast** (type conversion).

Examples: implicit and explicit conversions

```

int i = 3;
float f = i; // implicit conversion: int -> float

double d = 3.9;
int j = (int)d; // C-style cast: truncates the decimal part (narrowing)
int k = static_cast<int>(d); // C++-style cast: recommended as safer

```

Best practices:

- Prefer `static_cast<T>(expr)` for conversions between numeric types and between compatible pointers.
- `(int)d` is the C-style cast notation; you can also find `int(d)` which is the functional (function-style) form of cast. For fundamental types, both behave equivalently (they truncate the decimal part).
- Note that conversions can reduce precision or range (narrowing): `double -> int` truncates the decimal part; an unsigned integer may overflow.
- There is also `reinterpret_cast<T>(expr)`, which reinterprets the binary representation of an object as another type. It's a low-level operation, potentially dangerous (risks of alignment, aliasing, or undefined behavior); use it only for interoperability or clearly documented binary I/O reading/writing.

This concept is useful for explicitly controlling conversions and avoiding surprising behaviors during arithmetic operations or argument passing.

1.4 Formatted output and input: printf, scanf

printf and scanf (inherited from C)

In addition to `std::cout` and `std::cin`, C++ keeps the classic functions of the C language:

- **printf** (*print formatted*): for formatted output.
- **scanf** (*scan formatted*): for formatted input.

They are defined in the header `<cstdio>` (or `<stdio.h>` in C). Their usage relies on **format specifiers** (`%d`, `%f`, `%s`, etc.) which indicate the type of the variable.

Example of formatted output with printf

```
#include <stdio>

int main() {
    int age = 20;
    float taille = 1.75f;

    printf("Age : %d ans, taille : %.2f m\n", age, taille);
    return 0;
}
```

Output:

```
Age : 20 ans, taille : 1.75 m
```

- %d : signed integer (**int**)
- %u : unsigned integer (**unsigned**)
- %f : floating-point (**float** or **double**)
- %.nf : floating-point with *n* decimals
- %e : floating-point in scientific notation
- %c : character (**char**)
- %s : string (**char***)
- %x : integer in hexadecimal (lowercase)
- %X : integer in hexadecimal (uppercase)

%p | memory address (pointer) | printf("%p", &a); | 0x7ffee3c8a4 |
%% | literal '%' character | printf("%d"); | %d |

1.5 Contiguous-element containers, arrays

In C++, the **standard library (STL, Standard Template Library)** defines several containers that can store sets of values.

Among them, two structures are particularly important:

- **std::array<T, N>** : static array of fixed size.
 - The elements are stored contiguously in memory.
 - The size *N* must be known at **compile time** and cannot change.
 - The data is stored in the **stack memory**: faster access, but limited in size (typically a few MB).
- **std::vector<T>** : dynamic array.
 - The elements are also stored contiguously in memory.
 - The size can be modified during runtime (adding/removing elements).
 - The data is stored in the **heap memory**: slightly more costly in allocation, but allows access to all RAM.
- **Classic C arrays** (`T var[N]`) :
 - Fixed size, known at compile time.
 - No bounds checking.
 - No utility methods (`size()`, `push_back`, etc.).
 - Rarely used in modern C++, except to interact with C code or for very low-level needs.

Simple example with `std::vector`

```
#include <iostream>
#include <vector>

int main() {
    // Create an empty vector of integers
    std::vector<int> vec;

    // Add elements (automatic resizing)
    vec.push_back(5);
    vec.push_back(6);
    vec.push_back(2);

    // Size of the vector
    std::cout << "The vector contains " << vec.size() << " elements" << std::endl;

    // Access elements by index
    std::cout << "First element: " << vec[0] << std::endl;

    // Modify an element
    vec[1] = 12;

    // Iterate through the vector with a loop
    for (int k = 0; k < vec.size(); ++k) {
        std::cout << "Element " << k << " : " << vec[k] << std::endl;
    }

    return 0;
}
```

Access safety

[Warning]: accessing an element outside the bounds is undefined behavior, which can crash the program.

```
// Bad usage: may cause an error or unpredictable behavior
// vec[8568] = 12;

// Safe access (bounds checking)
vec.at(0) = 42;
```

Resizing

A vector can be resized dynamically with the method `.resize(N)` :

```
vec.resize(10000);
// The old elements are preserved
// The new elements are initialized to 0
```

Comparison of `std::array`, `std::vector`, and C arrays

```
#include <array>
#include <vector>
#include <iostream>

int main() {
    // Classic C array
    int tab[5] = {1, 2, 3, 4, 5};

    // std::array (static, fixed size)
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    // std::vector (dynamic, variable size)
    std::vector<int> vec = {1, 2, 3};
}
```

```

std::cout << "Size of the tab: " << 5 << " (fixed, known at compile time)" << std::endl;
std::cout << "Size of the array: " << arr.size() << std::endl;
std::cout << "Size of the vector: " << vec.size() << std::endl;

vec.push_back(10); // possible
// arr.push_back(10); // impossible: fixed size
// tab.push_back(10); // impossible: function does not exist

return 0;
}

```

Summary

- **C arrays** (`T var[N]`): simple, but limited and not very safe.
- `std::array<T, N>`: static array, size fixed at compile time, stored on the stack (stack memory).
- `std::vector<T>`: dynamic array, size modifiable, stored on the heap (heap memory).
- All three store their elements contiguously in memory.
- In practice:
 - Use `std::array` for small fixed sizes known in advance.
 - Use `std::vector` for data whose size can vary during the program.
 - Avoid C arrays except in special cases (interoperability with C code, low level).

1.6 Conditionals and Loops

if / else

General structure:

```

if (condition) {
    // instructions if the condition is true
} else {
    // instructions if the condition is false
}

```

[Note] Braces {} are **optional** if only one statement is present:

```

if (x > 0)
    std::cout << "x is positive" << std::endl;

```

Example:

```

int age = 20;

if (age >= 18) {
    std::cout << "Vous êtes majeur." << std::endl;
} else {
    std::cout << "Vous êtes mineur." << std::endl;
}

```

if / else if / else

General structure:

```

if (condition1) {
    // instructions
}

```

```
} else if (condition2) {  
    // instructions  
} else {  
    // default instructions  
}
```

Example:

```
int note = 15;  
  
if (note >= 16)  
    std::cout << "Très bien !" << std::endl;  
else if (note >= 10)  
    std::cout << "Suffisant." << std::endl;  
else  
    std::cout << "Échec." << std::endl;
```

The loops

The while loop

General structure:

```
while (condition) {  
    // instructions repeated while the condition is true  
}
```

Example:

```
int i = 0;  
while (i < 5) {  
    std::cout << "i = " << i << std::endl;  
    i++;  
}
```

The do ... while loop

General structure:

```
do {  
    // instructions executed at least once  
} while (condition);
```

Example:

```
int i = 0;  
do {  
    std::cout << "i = " << i << std::endl;  
    i++;  
} while (i < 5);
```

The for loop

General structure:

```
for (initialization; condition; increment) {  
    // repeated instructions  
}
```

Example:

```
for (int i = 0; i < 5; i++) {  
    std::cout << "i = " << i << std::endl;
```

```
}
```

The range-based for loop (C++11)

General structure:

```
for (type variable : conteneur) {  
    // instructions utilisant la variable  
}
```

Example:

```
#include <vector>  
  
int main() {  
    std::vector<int> valeurs = {1, 2, 3, 4, 5};  
  
    for (int v : valeurs)  
        std::cout << v << std::endl;  
}
```

Extension : switch / case

The switch statement allows testing several values of the same integer or character variable.

General structure:

```
switch (variable) {  
    case valeur1:  
        // instructions  
        break;  
    case valeur2:  
        // instructions  
        break;  
    default:  
        // default instructions  
}
```

[Warning] It only works with integer or character types.

The keyword `break` prevents the execution of the following blocks.

1.7 Associative containers: `std::map`

A `std::map` is an associative container from the standard library that stores key/value pairs sorted by the key. Each key is unique and allows efficient access to the corresponding value (search in $O(\log n)$).

- **Includes** : `#include <map>`
- **Order** : the elements are sorted by their key (uses the default `operator<`).
- **Access** : `operator[]` creates a default value if the key does not exist ; `find` allows testing for existence without creating.

Example simple: counting word frequency

```
#include <iostream>  
#include <map>  
#include <string>  
  
int main() {  
    std::map<std::string, int> counts;  
  
    // Insertion / increment  
    counts["pomme"] = 5;  
    counts["banane"] = 4;  
    counts["avocat"] = 8;  
}
```

```

counts["pomme"]++;

// Traversal and printing
for (auto pair : counts) {
    std::cout << pair.first << " : " << pair.second << std::endl;
}
// Displays:
// avocat : 8
// banane : 4
// pomme : 6

// Search without creation
auto it = counts.find("orange");
if (it == counts.end())
    std::cout << "orange non trouvé" << std::endl;

// Deletion
counts.erase("banane");

return 0;
}

```

Notes:

- Use `operator[]` to insert/access quickly. An entry is automatically created if the key is absent.
- To test existence without creating, use `find`.

1.8 Variable lifetimes

In C++, the lifetime (or **scope**) of a variable is determined by the **block of statements** in which it is declared.

A block is defined by curly braces { ... }.

The variable exists from its declaration until the closing brace } of the block.

Example 1: local variable within a block

```

int main()
{
    if (true) {
        int x = 5; // x is defined in the "if" block
        std::cout << x << std::endl;
    }
    // Here, x no longer exists: it is destroyed at the end of the block
}

```

Example 2: variable defined in an enclosing block

```

int main()
{
    int x = 5; // x is defined in the main() function's block
    if (true) {
        std::cout << x << std::endl; // x can be used in this sub-block
    }

    // x still exists until the end of main()
}

```

Important notes

- This behavior is **different from Python**, where a variable defined in an `if` or a loop remains accessible until the end of the function.
- It is **forbidden** to define several variables with the same name in the same block.

- This is possible in **sub-blocks**:

```
int x = 5;
{
    int x = 10; // allowed but to be avoided, as it is hard to read
    std::cout << x << std::endl; // prints 10
}
std::cout << x << std::endl; // prints 5
```

- **Best practice**: declare your variables in the block with the shortest possible lifetime. This improves code readability and reduces the risk of errors.

1.9 Functions

In C++, a **function** is a reusable block of code that performs a specific task. The general syntax is as follows:

```
typeRetour nomFonction(type nomArgument1, type nomArgument2, ...)
{
    // body of the function
    return valeur;
}
```

Simple example

```
int addition(int a, int b)
{
    return a + b;
}
```

- A function that does not return a value will have type **void**.
- A function that takes no arguments will simply have empty parentheses.
- The first line describing the name and the types of the function is called the **signature** or **header** of the function.
- The rest is called the **body** or **implementation** of the function.

Declaration and definition

In C++, it is necessary that the **signature** of a function be declared before its use. Otherwise, there will be a compilation error.

Correct example (definition before use)

```
int addition(int a, int b)
{
    return a + b;
}

int main()
{
    int c = addition(5, 3); // OK
}
```

Correct example (declaration then definition)

```
int addition(int a, int b); // Declaration

int main()
```

```

{
    int c = addition(5, 3); // OK
}

int addition(int a, int b) // Definition
{
    return a + b;
}

```

Incorrect example

```

int main()
{
    int c = addition(5, 3); // ERROR: addition has not yet been declared
}

int addition(int a, int b)
{
    return a + b;
}

```

Example: function norm

Let's write a function that calculates the **Euclidean norm** of a 3D vector with coordinates (x, y, z):

```

#include <iostream>
#include <cmath> // for std::sqrt

float norm(float x, float y, float z)
{
    return std::sqrt(x*x + y*y + z*z);
}

int main()
{
    std::cout << "Norm of (1,0,0) : " << norm(1.0f, 0.0f, 0.0f) << std::endl;
    std::cout << "Norm of (0,3,4) : " << norm(0.0f, 3.0f, 4.0f) << std::endl;
    std::cout << "Norm of (1,2,2) : " << norm(1.0f, 2.0f, 2.0f) << std::endl;
}

```

Expected output :

```

Norm of (1,0,0) : 1
Norm of (0,3,4) : 5
Norm of (1,2,2) : 3

```

Useful mathematical functions

- Square: `float x2 = x * x;`
- Square root: `float y = std::sqrt(x);`
- Power: `float y = std::pow(x, p);`

[Attention] Do not use `^` nor `**` in C++: these are **not** exponentiation operators.

Function Overloading

In C++, several functions can share the **same name** as long as their **parameters differ**. This is what we call the **overloading**.

Example

```

#include <iostream>
#include <cmath>

// Résout ax + b = 0
float solve(float a, float b) {
    return -b / a;
}

// Résout ax^2 + bx + c = 0 (une racine)
float solve(float a, float b, float c) {
    float delta = b*b - 4*a*c;
    return (-b + std::sqrt(delta)) / (2*a);
}

int main() {
    float x = solve(1.0f, 2.0f); // Appelle la 1ère version
    float y = solve(1.0f, 2.0f, 1.0f); // Appelle la 2ème version

    std::cout << "Solution linéaire : " << x << std::endl;
    std::cout << "Solution quadratique : " << y << std::endl;
}

```

Summary

- A function has a **signature** (header) and a **body** (implementation).
- It must be declared before use.
- Functions can return a value (return) or be **void**.
- The **overloaded functions** allow using the same name with different parameters.

1.10 Passing Arguments: Copy, Reference

In C++, function arguments are passed by **copy** by default:

- Changes made in the function remain local.
- For large objects (vectors, arrays, structures), copying can be **costly** in terms of performance.

Example with pass-by-copy

```

#include <iostream>

void increment(int a) {
    a = a + 1;
}

int main() {
    int x = 3;
    increment(x);
    std::cout << x << std::endl; // prints 3 (x is not modified)
}

```

Here, the variable `x` is not modified in `main` because `increment` works on a **copy**.

Pass by reference

One can use the symbol `&` in the signature to pass an argument **by reference**.

This allows directly modifying the original variable:

```

#include <iostream>

void increment(int& a) {
    a = a + 1;
}

int main() {

```

```

int x = 3;
increment(x);
std::cout << x << std::endl; // prints 4 (x is modified)
}

```

A **reference** is an alias: the function accesses the original variable and not a copy.

Example with `std::vector`

Consider a function that multiplies the values of a vector:

```

#include <iostream>
#include <vector>

std::vector<float> generate_vector(int N)
{
    std::vector<float> values(N);
    for (int k = 0; k < N; ++k)
        values[k] = k / (N - 1.0f);
    return values;
}

void multiply_values(std::vector<float> vec, float s)
{
    for (int k = 0; k < vec.size(); ++k) {
        vec[k] = s * vec[k];
    }
    std::cout << "Last value in the function: " << vec.back() << std::endl;
}

int main()
{
    int N = 101;
    std::vector<float> vec = generate_vector(N);

    multiply_values(vec, 2.0f);

    std::cout << "Last value in main: " << vec.back() << std::endl;
}

```

Expected output :

```

Last value in the function: 2
Last value in main: 1

```

Here, `vec` is passed **by value** to `multiply_values`.
The modification is made on a local copy, so `vec` in `main` remains unchanged.

Pass-by-reference (correction)

Let's modify the signature to pass the vector by reference:

```

void multiply_values(std::vector<float>& vec, float s)
{
    for (int k = 0; k < vec.size(); ++k) {
        vec[k] = s * vec[k];
    }
    std::cout << "Last value in the function: " << vec.back() << std::endl;
}

```

Expected result :

```

Last value in the function: 2
Last value in main: 2

```

Const references

If one wishes to avoid copying **without modifying** the vector, one can use a **const reference** :

```
float sum(std::vector<float> const& T) {
    float value = 0.0f;
    for (int k = 0; k < T.size(); k++)
        value += T[k];
    return value;
}
```

This type of passing allows :

1. To avoid copying the data.
2. To ensure that the values will not be modified in the function.

Best practice: use **const references** for large objects that should not be modified.

1.11 Classes

In C++, a **class** (or a **struct**) is a way to group in a single entity:

- **attributes** (data members),
- and **methods** (member functions) that operate on these data.

We then speak of an **object** to designate an instance of the class.

Declaration and use of a simple object

```
#include <iostream>
#include <cmath>

// Declaration of a struct
struct vec3 {
    float x, y, z;
};

int main()
{
    // Creation of an uninitialized vec3
    vec3 p1;

    // Creation and initialization of a vec3
    vec3 p2 = {1.0f, 2.0f, 5.0f};

    // Access and modification of the attributes
    p2.y = -4.0f;

    std::cout << p2.x << ", " << p2.y << ", " << p2.z << std::endl;

    return 0;
}
```

Struct vs Class

In C++, objects can be defined with the keyword **struct** or **class** :

```
struct vec3 {
    float x, y, z; // Default: public
};

class vec3 {
public:
    float x, y, z; // Must be specified explicitly
};
```

Main difference :

- In a **struct**, members are **public by default**.
- In a **class**, members are **private by default**.

In practice :

- We often use `struct` for simple objects that group public data.
- We prefer `class` when we want to encapsulate private data with access methods.

Methods (member functions)

A class can define methods, i.e., functions that directly manipulate its attributes.

```
#include <iostream>
#include <cmath>

struct vec3 {
    float x, y, z;

    float norm() const;    // method that does not modify the object
    void display() const; // likewise
    void normalize();      // method that modifies (x,y,z)
};

// Implementation of the methods

float vec3::norm() const {
    return std::sqrt(x * x + y * y + z * z);
}

void vec3::normalize() {
    float n = norm();
    x /= n;
    y /= n;
    z /= n;
}

void vec3::display() const {
    std::cout << "(" << x << ", " << y << ", " << z << ")" << std::endl;
}

int main()
{
    vec3 p2 = {1.0f, 2.0f, 5.0f};

    // Norm
    std::cout << p2.norm() << std::endl;

    // Normalization
    p2.normalize();

    // Display
    p2.display();

    return 0;
}
```

Remarks

- Methods can access the object's attributes directly without using `this->`, although that is possible.
- We generally separate the **declaration** (in the struct/class) and the **implementation** (with `NomClasse::NomMethode`).
- The keyword `const` placed after a method indicates that it does not modify the object. This improves robustness and readability.

Constructors and destructor

A class can define **constructors** to initialize its objects and a **destructor** to run code when they are destroyed.

```

#include <iostream>
#include <cmath>

struct vec3 {
    float x, y, z;

    // Default constructor
    vec3();

    // Custom constructor
    vec3(float v);

    // Destructor
    ~vec3();
};

// Initialization to 0
vec3::vec3() : x(0.0f), y(0.0f), z(0.0f) { }

// Initialization with a common value
vec3::vec3(float v) : x(v), y(v), z(v) { }

// Destructor
vec3::~vec3() {
    std::cout << "Goodbye vec3" << std::endl;
}

int main() {
    vec3 a; // calls vec3()
    vec3 b(1.0f); // calls vec3(float)

    return 0; // calls ~vec3()
}

```

Default constructors and destructors (= default)

In some cases, we do not want to redefine a constructor or a destructor, but simply explicitly ask the compiler to generate the default implementation. We then use the syntax `= default`.

```

struct vec3 {
    float x, y, z;

    // Automatically generates a default constructor
    vec3() = default;

    // Automatically generates a default destructor
    ~vec3() = default;
};

```

This is equivalent to not writing anything, but has two advantages:

- Readability: makes explicit that a constructor or destructor exists and should be the one provided by the compiler.
- Robustness: helps avoid some implicit deletions of constructors/destructors if others are defined in the class.

Member functions vs non-member functions

In C++, the choice between a **method** (member function) and an **external function** is left to the programmer. For example, the norm can also be defined as a standalone function:

```

#include <cmath>

struct vec3 {
    float x, y, z;
};

```

```

// Norm as a non-member function
float norm(const vec3& p) {
    return std::sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main() {
    vec3 p = {1.0f, 2.0f, 3.0f};
    float n = norm(p); // call as a function
}

```

Using `const&` avoids copying the object unnecessarily.

1.12 Writing/reading of external files

In C++, the `<fstream>` library provides the ability to write and read data to and from files. It provides three main classes:

- `std::ifstream` (*input file stream*) : for reading a file (input).
- `std::ofstream` (*output file stream*) : for writing to a file (output).
- `std::fstream` : to combine reading and writing.

Example: writing a vec3 to a file

We want to save the coordinates of a `vec3` in a text file.

```

#include <iostream>
#include <fstream>
#include <cmath>

struct vec3 {
    float x, y, z;
};

int main() {
    vec3 p = {1.0f, 2.0f, 3.5f};

    std::ofstream file("vec3.txt"); // opening for writing
    if (!file.is_open()) {
        std::cerr << "Erreur : impossible 'ouvrir le fichier !" << std::endl;
        return 1;
    }

    file << "Bonjour C++ !" << std::endl;
    file << p.x << " " << p.y << " " << p.z << std::endl;
    file.close(); // closing the file

    return 0;
}

```

After execution, the file `vec3.txt` contains :

```

Bonjour C++ !
1 2 3.5

```

Example: reading a vec3 from a file

We can then reread this `vec3` from the file :

```

#include <iostream>
#include <fstream>
#include <cmath>

struct vec3 {
    float x, y, z;
};

```

```

int main() {
    vec3 p;

    std::ifstream file("vec3.txt"); // opening for reading
    if (!file) {
        std::cerr << "Erreur : fichier introuvable !" << std::endl;
        return 1;
    }

    std::string line;
    std::getline(file, line);
    file >> p.x >> p.y >> p.z; // reading the three values
    file.close();

    std::cout << "vec3 relu : (" << p.x << ", " << p.y << ", " << p.z << ")" << std::endl;
    return 0;
}

```

Expected output :

```
vec3 relu : (1, 2, 3.5)
```

Opening modes

When opening a file, you can specify modes:

- `std::ios::in` : reading (default for `ifstream`).
- `std::ios::out` : writing (default for `ofstream`).
- `std::ios::app` : append to the end of the file without erasing it.
- `std::ios::binary` : read/write in binary mode (e.g. images).

Example :

```

std::ofstream file("log.txt", std::ios::app); // opening for append
file << "Nouvelle entrée" << std::endl;

```

1.13 Organization of code files

When a program becomes large, it is necessary to **separate the code into several files** in order to preserve readability, modularity and ease maintenance.

A typical organization with classes in C++ relies on **three types of files** :

1. Header file (.hpp or .h)

- Contains the **declarations** of classes, structures and functions.
- Serves as the public interface: what other files need to know to use the class.

2. Implementation file (.cpp)

- Contains the **code for the methods** and functions declared in the .hpp.
 - Provides the detailed implementation of the behaviors.

3. Main or usage file (main.cpp, etc.)

- Contains the `main()` function and uses the classes/functions by including the header file.

Example: organization with a vec3 class

Header file — vec3.hpp

```
#pragma once
#include <cmath>

// Déclaration de la classe
struct vec3 {
    float x, y, z;

    float norm() const;
    void normalize();
};

// Fonction non-membre
float dot(vec3 const& a, vec3 const& b);
```

Implementation file — vec3.cpp

```
#include "vec3.hpp"

// Méthodes de vec3
float vec3::norm() const {
    return std::sqrt(x*x + y*y + z*z);
}

void vec3::normalize() {
    float n = norm();
    x /= n; y /= n; z /= n;
}

// Fonction non-membre
float dot(vec3 const& a, vec3 const& b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

Usage file — main.cpp

```
#include "vec3.hpp"
#include <iostream>

int main() {
    vec3 v = {1.0f, 2.0f, 3.0f};

    std::cout << "Norme : " << v.norm() << std::endl;

    v.normalize();
    std::cout << "Norme après normalisation : " << v.norm() << std::endl;

    vec3 w = {2.0f, -1.0f, 0.0f};
    std::cout << "Produit scalaire v.w = " << dot(v, w) << std::endl;

    return 0;
}
```

Remarques importantes

- The directive `#include "vec3.hpp"` **copies** the contents of the `.hpp` file at compile time.
- **All files** that use `vec3` must include its header file (`vec3.hpp`).
- Never include a `.cpp` file directly into another file.
- Shared declarations should always be in a **single header file**, included by all the relevant files.

About #pragma once

The directive `#pragma once` is used in headers to prevent multiple inclusions of the same file. When a `.hpp` file is included multiple times (directly or indirectly), it can cause compilation errors related to redefinitions of classes or functions.

With `#pragma once`, the compiler guarantees that the file's contents will be included only once, even if several files try to include it.

It is a more concise and readable alternative to the classic include guards using `#ifndef`, `#define` and `#endif`.

In practice, it is recommended to always add `#pragma once` at the top of your header files.

1.14 Compilation

In C++, the **compilation** is the process that transforms human-readable source code (fichiers `.cpp` et `.hpp`) into an executable program understandable by the computer. This transformation takes place in several steps. The compiler starts by analyzing the code and translates it into **assembly code**.

The **assembly code** is a low-level language that directly corresponds to instructions understandable by the processor. Unlike C++ which is portable across systems and processors, the assembly is **dependent on the hardware architecture** (Intel x86, ARM, etc.). Each line of C++ can thus give rise to one or more assembly instructions, such as arithmetic operations, memory copy, or conditional jumps.

Next, this assembly code is converted into binary **machine code** which constitutes the processor's native language. This code is stored in a binary object file. Finally, a **linker** (*linker*) assembles the different object files and the libraries used to produce the final executable.

Thus, the role of compilation is to **translate a high-level language (C++) into low-level instructions (assembly, then machine)** that the processor can execute directly, while optimizing performance.

Simple diagram of the compilation pipeline

```
Source file (.cpp)
  ↓ (compiler)
Object file (.o)
  ↓ (linker / linker editor)
Executable (binary program)
```

Diagram with multiple source files

```
main.cpp  vec3.cpp  utils.cpp
  ↓        ↓        ↓
(compiler) (compiler) (compiler)
  ↓        ↓        ↓
main.o    vec3.o    utils.o
  ↓        ↓        ↓
[linker / linker editor]
  ↓
executable program
```

Example of assembly code

C++ Example

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = add(2, 3);
    return x;
}
```

Example of generated assembly (x86-64, simplified)

```
add(int, int):          # Start of function add
    mov    eax, edi     # Copy the 1st argument (a) into eax
    add   eax, esi     # Add the 2nd argument (b)
    ret                               # Return eax (result)

main:                  # Start of function main
    push  rbp          # Save base pointer
    mov   edi, 2       # Load 2 into register edi (1st argument)
    mov   esi, 3       # Load 3 into register esi (2nd argument)
    call  add(int, int) # Call the add function
    pop   rbp          # Restore base pointer
    ret                               # Return the result in eax
```

Explanations

- **edi and esi**: registers used to pass the first and second arguments to functions (x86-64 System V calling convention).
- **eax**: register where the result is stored and returned by the function.
- **mov**: copies a value into a register.
- **add**: performs addition between two registers.
- **ret**: returns from the function, using the value present in `eax` as the result.

On Linux/macOS

On Linux and macOS, the most commonly used compilers are **g++** (GNU) and **clang++** (LLVM).

To compile a simple program (one file):

```
g++ main.cpp -o programme
```

or

```
clang++ main.cpp -o programme
```

- `main.cpp` : C++ source file to compile.
- `-o programme` : name of the produced executable.

If the project contains several files, it becomes tedious to compile everything by hand. One then uses a **Makefile** with the **make** tool, which describes dependencies and compilation rules.

Minimal example of a Makefile:

Here is your **annotated Makefile** with the **general syntax in comments**:

```
# Cible par défaut (ici : "main")
all: main
# Syntaxe générale :
# cible: dépendances
#   commande(s) à exécuter

# Construction de l'exécutable "main"
main: main.o vec3.o
    g++ main.o vec3.o -o main
# Syntaxe générale :
# executable: fichiers_objets
#   compilateur fichiers_objets -o executable

# Règle pour générer l'objet main.o
main.o: main.cpp vec3.hpp
    g++ -c main.cpp
# Syntaxe générale :
# fichier.o: fichier.cpp fichiers_inclus.hpp
#   compilateur -c fichier.cpp

# Règle pour générer l'objet vec3.o
```

```
vec3.o: vec3.cpp vec3.hpp
    g++ -c vec3.cpp
# Syntaxe générale :
# fichier.o: fichier.cpp fichiers_inclus.hpp
#     compilateur -c fichier.cpp

# Nettoyage des fichiers intermédiaires
clean:
    rm -f *.o main
# Syntaxe générale :
# clean:
#     commande pour supprimer les fichiers générés
```

Windows

On Windows, the compiler is provided directly by **Microsoft Visual Studio** (MSVC). It does not rely on `make` nor on Makefiles. Instead, the code is organized into a **Visual Studio project** (`.sln`) that describes the files, dependencies and compilation options.

The Visual Studio IDE automatically handles launching the MSVC compiler when you press "Build" or "Run". Therefore, it is not necessary (and not practical) to manually call `cl.exe` from the command line.

Meta-configuration via CMake

To avoid writing a Linux-specific Makefile **and** a Windows-specific Visual Studio project, we use **CMake**.

- CMake is a **project generation** tool.
- It reads a configuration file (`CMakeLists.txt`) and automatically generates the files suited to your system:
 - **Linux/MacOS** → a **Makefile** usable with `make`.
 - **Windows** → a **Visual Studio project** (`.sln`).

Example usage under Linux/MacOS:

```
# From the project directory
mkdir build
cd build
cmake ..
make           # under Linux/MacOS
```

In summary

- Linux/MacOS: compilation via `g++` or `clang++`, automation via **Makefile**.
- Windows: compilation via MSVC through a Visual Studio project.
- CMake: cross-platform tool that automatically generates the right type of project (Makefile or `.sln`).

2 Fundamental types, encoding

In C++, variables are **typed**: each variable corresponds to a **memory space** (one or more slots) interpreted according to a **type**. Examples of fundamental types:

```
int a = 5; // signed integer (typically 4 bytes) float b = 5.0f; // single-precision floating point (4 bytes) double c = 5.0; // double-precision floating point (8 bytes) char d = 'k'; // character (1 byte = 8 bits), equals 107 in ASCII size_t e = 100; // unsigned integer capable of encoding a memory position (8 bytes on 64-bit machines), used to indicate sizes of arrays e.g. size() of a std::vector.
```

Important remarks:

- The size of types depends on the **architecture** and the compiler (except `char` guaranteed to be 1 byte).
- No type occupies less than a byte (8 bits).
- For efficiency reasons, memory is often **aligned**: some structures add padding (empty bytes) to align on 4 or 8 bytes.

2.1 Encoding of integers

Binary representation

An integer is represented in **binary**:

- Each bit is 0 or 1.
- A group of bits is packed into **octets** (8 bits).
- The values are interpreted in base 2.

Example:

Decimal	Binary (8 bits)
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
156	10011100

An integer can be represented across multiple octets:

- 4 octets (`int` classic) = 32 bits → up to 2^{32} possible values.
- 8 octets (`long long`) = 64 bits → up to 2^{64} possible values.

Unsigned integers

An **unsigned int** on 4 octets (32 bits) encodes values from 0 to $2^{32} - 1 = 4\ 294\ 967\ 295$.

Example in hexadecimal (practical representation of the bytes) :

- 00000000 → 0
- FFFFFFFF → 4294967295

Reminder:

- 1 octet (8 bits) = 2 hexadecimal characters
- E.g.: 10011100 = 9c in hexadecimal = 156 in decimal

Signed integers and two's complement

Signed integers use the leftmost bit (MSB) to encode the sign:

- 0 → positive
- 1 → negative

Encoding method: **two's complement**.

- To obtain the negative value of an integer:
 1. Invert all the bits.
 2. Add 1.

Example on 8 bits:

```
00000101 = +5
Inverting bits -> 11111010
Add +1 -> 11111011 = -5
```

Consequences:

- On 8 bits, value range: from -128 to +127.
- On 32 bits (`int`): from -2,147,483,648 to +2,147,483,647.

Practical example

Let's take a signed integer encoded on 2 octets:

```
C4 8D (hexadecimal)
= 11000100 10001101 (binary)
```

- Interpreted as **unsigned**: 50317.
- Interpreted as **signed two's complement**:
 - Invert bits → 00111011 01110010
 - Add 1 → 00111011 01110011 = 15219
 - Therefore the value = -15219.

2.2 Encoding of floating-point numbers

Floating-point numbers (`float`, `double`) follow the **IEEE 754** standard.

A floating-point number is represented by three parts:

1. **Sign** (1 bit)
2. **Exponent** (8 bits for `float`, 11 bits for `double`)
3. **Mantissa** (23 bits for `float`, 52 bits for `double`)

Formula:

$$x = (-1)^s \times (1 + mantissa) \times 2^{exponent - bias}$$

* `float` (32 bits) → bias = 127 * `double` (64 bits) → bias = 1023

Example: 46 3F CC 30 (float in hexadecimal) = 12275.046875 in decimal.

[Note] Important properties:

- Precision depends on the value: larger near 0, smaller for very large numbers.
- Some numbers are **not exactly representable** (e.g., 0.1, 0.4).
- Always compare two floating-point numbers with a **tolerance** ϵ :

```
if (std::abs(a - b) < 1e-6) { ... }
```

2.3 Notion of Endianness

When an integer occupies several bytes (for example a 4-byte `int`), the computer must decide **in what order the bytes are stored in memory**. This is called **endianness** (or byte order).

Two main conventions

1. Little Endian (Intel x86, ARM in default mode)

- The least significant byte is stored first (at the smallest address).
- Example :

```
int a = 0x12345678;
```

Memory representation (increasing addresses) :

Address:	1000	1001	1002	1003
Contents:	78	56	34	12

2. Big Endian (some network architectures, PowerPC, older processors)

- The most significant byte is stored first.
- For the same value `0x12345678` :

Address:	1000	1001	1002	1003
Contents:	12	34	56	78

Why is this important?

- **Network compatibility** Protocols (TCP/IP, etc.) require Big Endian (network byte order). Typical PCs (Intel) use Little Endian: you must convert before sending or after receiving.
- **Binary files** If a program writes a binary file in Little Endian, it must specify that order. Otherwise, on a Big Endian machine, the values read will be incorrect.
- **Interoperability** Any communication between heterogeneous machines must specify the byte order.

2.4 Summary of Fundamental Types

Type	Description	Typical size (x86/64-bit)	Example declaration
<code>char</code>	ASCII character (or signed small integer)	1 byte	<code>char c = 'A';</code>
<code>bool</code>	Boolean value (<code>true</code> or <code>false</code>)	1 byte (vectorized)	<code>bool b = true;</code>

```
short | signed short integer | 2 bytes | short s = 123; |
int | standard signed integer | 4 bytes | int a = 42; |
long | signed integer (size varies by architecture) | 4 bytes (Windows), 8 (Linux) | long l = 100000; |
long long | long signed integer (guaranteed >= 64 bits) | 8 bytes | long long x = 1000000000000LL; |
unsigned | unsigned integer (>=0 only) | same size as signed | unsigned u = 42; |
float | single-precision floating-point number (IEEE754) | 4 bytes | float f = 3.14f; |
double | double-precision floating-point number | 8 bytes | double d = 2.718; |
long double | extended-precision floating-point (arch-dependent) | 8, 12 or 16 bytes | long double pi = 3.14159; |
size_t | unsigned integer for memory addressing | 8 bytes (64 bits) | size_t n = vec.size(); |
wchar_t | wide character (Unicode, platform-dependent) | 2 bytes (Windows), 4 (Linux) | wchar_t wc = 'é'; |
```

Note: The size may vary depending on the compiler and architecture, **except char which is always 1 byte**.

2.5 Getting the size with sizeof

In C and C++, the `sizeof` operator returns the size in bytes of a type or a variable.

Examples :

```
#include <stdio.h>

int main() {
    printf("sizeof(char) = %zu\n", sizeof(char));
    printf("sizeof(int) = %zu\n", sizeof(int));
    printf("sizeof(float) = %zu\n", sizeof(float));
    printf("sizeof(double)= %zu\n", sizeof(double));

    int a;
    double b;
    printf("sizeof(a) = %zu\n", sizeof(a));
    printf("sizeof(b) = %zu\n", sizeof(b));
    return 0;
}
```

Typical output on a 64-bit machine :

```
sizeof(char) = 1
sizeof(int) = 4
sizeof(float) = 4
sizeof(double)= 8
sizeof(a) = 4
sizeof(b) = 8
```

Rem. : the `%zu` specifier is the one provided by the standard to print a value of type `size_t` (e.g., the result of `sizeof`). It is also possible to convert to `unsigned long` and use `%lu`.

2.6 Important notes

- `sizeof(type)` is evaluated at compile time, without running the program.
- The size of a type can change depending on the architecture (32-bit vs 64-bit).
- Memory alignment can introduce padding in `structs`.
- To know the sizes with certainty on your machine, it's advisable to write a small program using `sizeof`.

2.7 Types with specific sizes

To obtain deterministic sizes (architecture-independent), the C/C++ standard defines types in the header (C++11 / C99). These types guarantee a precise number of bits, which is essential for serialization, binary formats, and network protocols.

Fixed-width types:

- `uint8_t` / `int8_t` : unsigned / signed 8-bit integers
- `uint16_t` / `int16_t` : unsigned / signed 16-bit integers
- `uint32_t` / `int32_t` : unsigned / signed 32-bit integers
- `uint64_t` / `int64_t` : unsigned / signed 64-bit integers

Useful additional examples:

- `int_fast32_t`, `uint_fast32_t` : integer types at least 32 bits, chosen for better performance on the platform
- `int_least16_t`, `uint_least16_t` : integer types of at least 16 bits (minimum guarantee)
- `intptr_t`, `uintptr_t` : signed/unsigned integers capable of holding a pointer value

Example usage :

```
#include <stdint>
#include <inttypes> // for the macros PRIu32, PRId64, ...
#include <stdio>
```

```

int main() {
    uint8_t a = 255;
    int16_t b = -12345;
    uint32_t c = 0xDEADBEEF;

    std::printf("sizeof(uint8_t) = %zu\n", sizeof(uint8_t));
    std::printf("sizeof(int16_t) = %zu\n", sizeof(int16_t));
    std::printf("sizeof(uint32_t) = %zu\n", sizeof(uint32_t));

    // safe usage with printf :
    std::printf("c = %" PRIu32 "\n", c);
    return 0;
}

```

2.8 Bitwise operations

Bitwise operations allow direct manipulation of the bits of an integer. They are very useful for working with flags, masks, optimizing simple calculations, or for low-level data processing (compression, binary formats, etc.).

Main operations in C/C++ :

- `&` : bitwise AND
- `|` : bitwise OR
- `^` : XOR (exclusive OR) bitwise
- `~` : NOT (negation) bitwise
- `<<` : left shift (shift left)
- `>>` : right shift (shift right)

Simple examples :

```

unsigned a = 0b1100; // the 0bxxxx notation allows defining a value in binary, here 1100 in binary => 12 in decimal
unsigned b = 0b1010; // 1010 in binary => 10 in decimal

unsigned and_ab = a & b; // 1000 (8)
unsigned or_ab = a | b; // 1110 (14)
unsigned xor_ab = a ^ b; // 0110 (6)
unsigned not_a = ~a; // invert all bits

// shifts
unsigned left = a << 1; // 11000 (24) : left shift (multiplication by 2)
unsigned right = a >> 2; // 0011 (3) : right shift (division by 2)

// print in hex / decimal as needed

```

Masks and bit tests

Masks are used to isolate, set, or clear bits :

```

unsigned flags = 0;
const unsigned FLAG_A = 1u << 0; // bit 0 -> 0b0001
const unsigned FLAG_B = 1u << 1; // bit 1 -> 0b0010
const unsigned FLAG_C = 1u << 2; // bit 2 -> 0b0100

// enable a flag
flags |= FLAG_B; // flags = 0b0010

// test if a flag is set
bool hasB = (flags & FLAG_B) != 0;

// disable a flag
flags &= ~FLAG_B; // clears bit 1

// toggle a flag
flags ^= FLAG_C; // inverts the state of bit 2

```

Conseils importants

- Use unsigned types (`unsigned`, `uint32_t`, `uint64_t`) for bitwise operations: the behavior of shifts on negative signed integers can be undefined or depend on the implementation.
- Left shift $x \ll n$ multiplies by 2^n as long as it does not overflow. Right shift $x \gg n$ divides by 2^n for unsigned types.
- To isolate a byte in a word (useful for endianness or extraction) :

```
uint32_t w = 0x12345678;
uint8_t byte0 = (w >> 0) & 0xFF; // 0x78 (LSB)
uint8_t byte1 = (w >> 8) & 0xFF; // 0x56
uint8_t byte2 = (w >> 16) & 0xFF; // 0x34
uint8_t byte3 = (w >> 24) & 0xFF; // 0x12 (MSB)
```

Use `std::bitset` to display/manipulate bits in a safe and readable way:

```
#include <bitset>
#include <iostream>

std::bitset<8> bs(0b10110010);
std::cout << bs << "\n"; // prints 10110010
bs.flip(0); // toggle bit 0
bs.set(3); // set bit 3 to 1
bs.reset(7); // set bit 7 to 0
```

2.9 Résumé

- Common types cover signed/unsigned integers, floating-point numbers, and characters.
- Their size is not always fixed (except `char` = 1 byte guaranteed).
- `sizeof` lets you know precisely the size of a type or a variable on a given architecture.

3 Pointers

3.1 Concept of storage and memory addressing

The memory of a computer can be seen as a **large linear array** of cells.

- Each cell contains a **byte** (i.e., 8 bits).
- Each cell has a **unique address**, which is a number that allows access to it.

We can thus imagine memory as a sequence of numbered cells:

Address	Content
1000	10101010
1001	00001111
1002	11110000
1003	01010101
...	

Here:

- each line represents a **byte** of memory,
- the **address** (1000, 1001, ...) is an integer managed by the processor,
- the **content** is a set of 8 bits (0 or 1).

Addresses and variables

When we declare a variable in C++:

```
int a = 42;
```

- The compiler reserves **4 consecutive bytes** (on a 32-bit or 64-bit architecture).
- Suppose the variable starts at address 1000. The memory might look like this:

Address	Content
1000	00101010 (0x2A)
1001	00000000
1002	00000000
1003	00000000

Thus:

- the variable `a` is seen as a **whole** (42),
- but in reality, it is stored as **four consecutive bytes** in memory.

Size and alignment

- **char**: 1 byte
- **short**: 2 bytes
- **int**: 4 bytes (most of the time)
- **long long**: 8 bytes
- **float**: 4 bytes
- **double**: 8 bytes

Note: The size can vary depending on the architecture, but **1 byte = 8 bits is guaranteed**.

For performance reasons, the compiler may introduce padding (filling with zeros) so that some variables start at addresses multiple of 2, 4, or 8. This eases memory access for the processor.

Importance of the address

The memory address is what allows:

- to precisely identify where a variable is located,
- to access its bytes,
- to manipulate complex data structures.

Analogy example

We can compare memory:

- to a **library** where each memory cell would be a book,
- the **address** is the shelf number + the book number,
- the **content** is the information written in that book (the bits).

To access a datum, the processor must know the **exact address**.

Summary

- Memory is organized into 1-byte cells (8 bits).
- Each cell has a **unique address**.
- Variables occupy one or more **consecutive** cells.
- Addresses allow the processor to locate and manipulate these values.
- This view is essential to understand how **pointers** and **dynamic memory allocation** work.

3.2 Address of a variable

Each variable in memory has an **address**, that is, the position of its first byte in the large memory array. In the C language (and thus also in C++), you can access this address using the **&** operator (the *address of*).

Simple example

```
#include <stdio.h>

int main() {
    int a = 42;

    printf("Valeur de a : %d\n", a);
    printf("Adresse de a : %p\n", &a);

    return 0;
}
```

Possible output (the address depends on the run and the machine) :

```
Valeur de a : 42
Adresse de a : 0x7ffee3b5a9c
```

- %d prints the integer value (42 here).
- %p prints a memory address (pointer format).
- &a means "the address of the variable a."

Reading and writing via the C function scanf

When using `scanf`, you must provide the **address of the variable** in which to store the result.

```
#include <stdio.h>

int main() {
    int age;
```

```

printf("Entrez votre age : ");
scanf("%d", &age); // &age = address of age

printf("Vous avez %d ans.\n", age);

return 0;
}

```

- Here `scanf("%d", &age)` places the value read directly into the memory cell of `age`.
- If we had written `scanf("%d", age)` (without `&`), the program would crash, because `scanf` needs the **address** to modify the variable.

Observation of the address

We can see that two successive variables in memory have different addresses, separated by their size in bytes.

```

#include <stdio.h>

int main() {
    int x = 10;
    int y = 20;

    printf("Adresse de x : %p\n", &x);
    printf("Adresse de y : %p\n", &y);

    return 0;
}

```

Example output:

```

Adresse de x : 0x7ffee3b5a98
Adresse de y : 0x7ffee3b5a94

```

Note: The addresses are close but not necessarily in increasing order, because the compiler and the system may arrange the variables differently (stack, memory alignment, etc.).

3.3 Argument passing

Pass by value (default behavior)

In C and C++, function arguments are **passed by value**:

- When you call a function, the program creates a **copy of the variable** in the function's memory.
- The function therefore works on its own copy.

Example :

```

#include <stdio.h>

void increment(int x) {
    x = x + 1; // modifies only the local copy
}

int main() {
    int a = 5;
    increment(a);
    printf("a = %d\n", a); // prints still 5
    return 0;
}

```

Memory explanation :

- `a` in `main` occupies a memory region.
- When calling `increment(a)`, the value `5` is copied into a new local variable `x` inside the function.
- Modifying `x` does not change `a`, because they are two independent variables.

Passing by address with a pointer

If we want a function to **be able to modify the original variable**, we must pass it not the value, but **the address of the variable**.

```
#include <stdio.h>

void increment(int* p) {
    *p = *p + 1; // modifies the value at the pointed address
}

int main() {
    int a = 5;
    increment(&a); // we pass the address of a
    printf("a = %d\n", a); // prints 6
    return 0;
}
```

Detailed explanation :

1. In `main`, we have the variable `a` (value 5) stored at a certain memory address (for example, 1000).
2. The expression `&a` yields this address (1000).
3. When calling `increment(&a)`, it's not **a that is copied**, but its **address** (1000).
 - The function thus receives a **pointer** `p`, which is a copy of the address.
4. Inside `increment`, `*p` means “the value stored at the address `p`”.
 - So `*p = *p + 1;` will fetch the value 5 at address 1000, increment it, and store 6 at the same location.
5. Since `p` designates the memory of `a`, the variable `a` is actually modified.

Summary of mechanisms

- **Pass by value:** the value is copied into a new local variable. The original variable is never modified.
- **Pass by address (pointer):** the **address** is copied into a pointer. The function thus has access to the same memory area, and can modify the original variable via `*p`.

Diagram (ASCII simplified) :

```
main:
  a = 5      (address 1000)

Call to increment(&a) :
  copy of address 1000 into p

increment:
  p = 1000
  *p = value stored at address 1000 = 5
  *p = 6    (modifies memory shared with a)
```

Best practices with pointers

A pointer is a variable that contains a memory address. However, if a pointer is not initialized, it may contain a **random address**, which leads to unpredictable behavior (segmentation fault, memory corruption).

Essential rule: always initialize pointers.

In modern C++, we use `nullptr` to indicate that a pointer points to nothing :

```
#include <iostream>

int main() {
    int* p = nullptr; // initialized pointer, but does not point to anything

    if(p == nullptr) {
```

```

    std::cout << "The pointer is empty, no dangerous access." << std::endl;
}

return 0;
}

```

Example of bad practice

```

int* p;    // uninitialized pointer (dangerous!)
*p = 10;   // undefined behavior -> likely crash

```

Here, `p` contains an indeterminate value: accessing `*p` is **dangerous**.

Correct example

```

int* p = nullptr; // safe pointer, but null
if(p != nullptr) {
    *p = 10;       // we access only if p points to a valid variable
}

```

Summary

- Always initialize your pointers (with `nullptr` by default).
- Always check that a pointer is not null before using it.
- Prefer references (&) or modern containers (`std::vector`, `std::unique_ptr`, `std::shared_ptr`) when possible, to avoid memory management errors.

3.4 Case of contiguous arrays

C Arrays

In C and C++, an **array** is always stored in memory as a sequence of **contiguous bytes**. This means that the elements follow each other, with no gaps between them.

Example :

```

#include <stdio.h>

int main() {
    int tab[3] = {10, 20, 30};

    printf("Address of tab[0] : %p\n", &tab[0]);
    printf("Address of tab[1] : %p\n", &tab[1]);
    printf("Address of tab[2] : %p\n", &tab[2]);

    return 0;
}

```

Possible output :

```

Address of tab[0] : 0x7ffee6c4a90
Address of tab[1] : 0x7ffee6c4a94
Address of tab[2] : 0x7ffee6c4a98

```

We note that the addresses are spaced by 4 bytes (the size of an `int`), which confirms the **memory contiguity**.

Pointer arithmetic

The name of an array (`tab`) is automatically converted to a **pointer to its first element** (`&tab[0]`). We can then use the **pointer arithmetic**:

```
* `p + N` : shifts the pointer by `N` elements.  
* `*(p + N)` : accesses the value of the `N`-th element.
```

This is exactly equivalent to writing `tab[N]`.

Example :

```
``c  
#include <stdio.h>  
  
int main() {  
    int tab[3] = {10, 20, 30};  
    int* p = tab; // equivalent to &tab[0]  
  
    printf("%d\n", *(p + 0)); // 10  
    printf("%d\n", *(p + 1)); // 20  
    printf("%d\n", *(p + 2)); // 30  
  
    return 0;  
}
```

These two notations are equivalent :

```
tab[i]  <=>  *(tab + i)
```

Memory diagram (example with tab[3])

```
Address : 1000  1004  1008  
Contents: 10   20   30  
Index   : tab[0] tab[1] tab[2]
```

```
p = 1000  
*(p+0) -> value at 1000 -> 10  
*(p+1) -> value at 1004 -> 20  
*(p+2) -> value at 1008 -> 30
```

Adapting to the element size

Memory contiguity applies to any array type, not just integers. If we define an array of larger objects (for example doubles or structs), the elements remain stored one after another.

Example with double

```
#include <stdio.h>  
  
int main() {  
    double tab[3] = {1.1, 2.2, 3.3};  
  
    printf("Address of tab[0] : %p\n", &tab[0]);  
    printf("Address of tab[1] : %p\n", &tab[1]);  
    printf("Address of tab[2] : %p\n", &tab[2]);  
  
    return 0;  
}
```

Possible output (each `double` = 8 bytes) :

```
Address of tab[0] : 0x7ffee6c4a90  
Address of tab[1] : 0x7ffee6c4a98  
Address of tab[2] : 0x7ffee6c4aa0
```

We can see that the addresses are spaced by 8, because a `double` occupies 8 bytes.

In C/C++, the expression $p + N$ **does not mean** "add N bytes", but "go to the N-th element from p".

- If p is of type `int*` and `sizeof(int) == 4`, then :

$p + 1$ -> advances by 4 bytes $p + 2$ -> advances by 8 bytes * If p is of type `double*` and `sizeof(double) == 8`, then :
 $p + 1$ -> advances by 8 bytes $p + 2$ -> advances by 16 bytes * Generally :

$\text{Address}(p + N) = \text{Address}(p) + N * \text{sizeof}(\text{type})$

It is the compiler that translates the operation into address calculation, and it is the processor that performs the addition during execution.

Dynamic arrays in C++: `std::vector`

In modern C++, we use `std::vector` rather than static arrays, because it offers :

- a **dynamic size** (we can add elements with `push_back`),
- automatic memory management,
- and it preserves the **memory contiguity**.

Example :

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {10, 20, 30};

    std::cout << "Address of v[0] : " << &v[0] << std::endl;
    std::cout << "Address of v[1] : " << &v[1] << std::endl;
    std::cout << "Address of v[2] : " << &v[2] << std::endl;
}
```

Typical output :

```
Address of v[0] : 0x7ffee6c4a90
Address of v[1] : 0x7ffee6c4a94
Address of v[2] : 0x7ffee6c4a98
```

We observe the same contiguity as with classic arrays.

Pointer arithmetic on `std::vector`

We can obtain a pointer to the internal data thanks to `v.data()` or `&v[0]`, then use the same logic as for C arrays.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {10, 20, 30};
    int* p = v.data(); // pointer to the first element

    std::cout << *(p+0) << std::endl; // 10
    std::cout << *(p+1) << std::endl; // 20
    std::cout << *(p+2) << std::endl; // 30
}
```

Résumé

- C arrays and `std::vector` store their elements in a **contiguous** manner.
- This enables fast indexed access (`tab[i]`) or via pointer arithmetic (`*(p+i)`).
- The `std::vector` also offers a **dynamic size** and safer memory management, but retains the same fundamental contiguity properties.

3.5 Contiguity in classes and structs

In C and C++, the **structures** (`struct`) and **classes** group several variables (members) into a single block of memory. By default, the fields are laid out one after another, which guarantees a **memory contiguity**.

Simple example

```
#include <stdio.h>

struct Point2D {
    int x;
    int y;
};

int main() {
    struct Point2D p = {1, 2};

    printf("Address of p.x : %p\n", &p.x);
    printf("Address of p.y : %p\n", &p.y);

    return 0;
}
```

Possible output :

```
Address of p.x : 0x7ffee3b5a90
Address of p.y : 0x7ffee3b5a94
```

Here, the two integers `x` and `y` (4 bytes each) are stored one after the other contiguously.

Padding and alignment

For performance reasons, the compiler may insert **padding bytes** between members to maintain optimal memory alignment.

```
struct Test {
    char a; // 1 byte
    int b; // 4 bytes
};
```

Memory layout :

```
Address  Content
1000     a (1 byte)
1001-1003 padding (3 unused bytes)
1004-1007 b (4 bytes)
```

Example with multiple fields

```
struct Mixed {
    char c; // 1 byte
    double d; // 8 bytes
    int i; // 4 bytes
};
```

Typical layout on a 64-bit machine :

```
Address  Field
1000     c (1 byte)
```

```
1001-1007 padding (7 bytes)
1008-1015 d (8 bytes)
1016-1019 i (4 bytes)
1020-1023 padding (4 bytes for overall alignment)
```

Total size: 24 bytes.

Contiguity in classes

In C++, a `class` behaves like a `struct` from the memory perspective:

- Data members are placed contiguously, with the same padding and alignment rules.
- The difference between `struct` and `class` is only in default visibility (`public` vs `private`).

`std::vector` of structures

In modern C++, you can store several `struct` or `class` objects in a `std::vector`. The vector guarantees that the elements are placed contiguously in memory, exactly like a C array.

Example:

```
#include <iostream>
#include <vector>

struct Point2D {
    int x;
    int y;
};

int main() {
    std::vector<Point2D> points = {{1,2}, {3,4}, {5,6}};

    std::cout << "Adresse du premier Point2D : " << &points[0] << std::endl;
    std::cout << "Adresse du deuxième Point2D : " << &points[1] << std::endl;
    std::cout << "Adresse du troisième Point2D : " << &points[2] << std::endl;
}
```

ASCII Diagram of a `std::vector<Point2D>`

Each `Point2D` occupies `sizeof(Point2D)` bytes (here, 8 bytes: 2 integers of 4 bytes). The elements of the `std::vector` are stored **back-to-back** in memory:

```
Memory of a std::vector<Point2D> with 3 elements

Address : 2000      2008      2016
Content  : [x=1, y=2] [x=3, y=4] [x=5, y=6]
Size     : 8 bytes  8 bytes  8 bytes
```

We can see that each element is a **structured block**, but the blocks remain **contiguous**.

Summary

- The fields of a `struct` or `class` are stored contiguously, with potential padding to respect alignment.
- The actual size can be larger than the sum of the fields.
- A `std::vector<struct>` lets you create a dynamic array of structures that is also contiguous in memory.
- This contiguity enables fast in-memory traversal and compatibility with C functions via `points.data()`.

3.6 Memory Organization AoS vs SoA

When dealing with structured data in large quantities (for example, 3D coordinates, particles, vertices in graphics), there are two classic ways to organize data in memory:

Array of Structs (AoS)

This is the classic representation with a `std::vector<struct>`. Each element of the array is a complete structure.

Example:

```
struct Point3D {
    float x, y, z;
};

std::vector<Point3D> points = {
    {1.0f, 2.0f, 3.0f},
    {4.0f, 5.0f, 6.0f},
    {7.0f, 8.0f, 9.0f}
};
```

Memory (each `Point3D` = contiguous block of 12 bytes) :

```
[x=1, y=2, z=3] [x=4, y=5, z=6] [x=7, y=8, z=9]
```

Here, contiguity applies **at the level of the structures**:

- The `Point3D` are laid out back-to-back.
- Each `Point3D` itself contains its contiguous `x`, `y`, `z` fields.

Advantage: convenient for manipulating a complete point. **Disadvantage:** if you only want to process the `x` values, you have to unnecessarily traverse the `y` and `z`.

Struct of Arrays (SoA)

Here, we invert the organization: instead of storing an array of structures, we store a structure that contains an array per field.

Example:

```
struct PointsSoA {
    std::vector<float> x;
    std::vector<float> y;
    std::vector<float> z;
};
```

Memory (each field is contiguous separately) :

```
x : [1, 4, 7]
y : [2, 5, 8]
z : [3, 6, 9]
```

Here, contiguity applies at the level of fields:

- All `x` are stored one after another.
- All `y` are contiguous, and likewise for the `z`.

Advantage: very efficient if one does heavy processing on a single field (e.g., applying a transformation to all `x` coordinates). **Disadvantage:** less natural if you want to work on a complete point (`x,y,z` grouped).

Contiguity: two complementary views

- **AoS** : contiguity *by object*. Each element of the array is a structured block (`{x,y,z}`), and the blocks follow one after another.
- **SoA** : contiguity *by field*. Each field is grouped in its own array, and the values follow by dimension.

The two approaches therefore use memory contiguity, but not at the same level of structuring.

Practical choice

- **AoS** : often preferred when data are manipulated as independent entities (e.g., list of particles, game objects, 3D vectors in a physics engine).
- **SoA** : used in **high-performance simulation**, **scientific computing**, **GPU** or **vectorized data processing**, because it favors optimized sequential accesses (cache, SIMD).

3.7 Dynamic allocation

So far, we have seen **automatic variables** (declared in a function), stored on the **stack** and **destroyed automatically** at the end of the block.

But in some cases, we need data whose lifetime extends beyond the end of a block (for example: keep an array created in a function, manage large structures, or build dynamic graphs). In this case, we use **dynamic memory**, allocated on the **heap**.

Stack vs heap

Characteristic	Stack	Heap
Allocation	Automatic	Manual (or controlled by objects)
Lifetime	Limited to the current block	Until explicit release
Maximum size	Limited (a few MB)	Very large (several GB)
Management	By the compiler	By the programmer
Example	<code>int a;</code> OR <code>int tab[10];</code>	<code>new int;</code> OR <code>new int[n];</code>

On most systems, the stack has a limited size (~8 MB by default), whereas the heap can use several gigabytes. Dynamic allocation thus allows you to **create large structures** or **variable-sized** ones at runtime.

Problem: lifetime of local variables

```
#include <iostream>

int* createValue() {
    int a = 42; // local variable on the stack
    return &a; // Dangerous: a is destroyed at the end of the function
}

int main() {
    int* p = createValue();

    std::cout << *p << std::endl; // undefined behavior !
}
```

`a` is destroyed on exit from `createValue()`. The returned pointer becomes **dangling** (dangerous).

Solution: heap allocation

```
#include <iostream>

int* createValue() {
    int* p = new int(42); // allocated on the heap
    return p; // valid even after the function ends
}
```

```
int main() {
    int* q = createValue();
    std::cout << *q << std::endl; // 42
    delete q; // deallocation mandatory
}
```

Here, the variable `*q` persists after the end of `createValue()`. But **the programmer must free the memory** with `delete`.

Dynamic allocation in C: `malloc` and `free`

In C, we use the standard library functions `<stdlib.h>`.

```
#include <stdlib.h>

int* p = (int*)malloc(sizeof(int));
```

Here:

- `malloc` reserves a block of memory of `sizeof(int)` bytes,
- it returns a pointer of type `void*`,
- this pointer is explicitly converted to `int*`.

Usage :

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* p = (int*)malloc(sizeof(int));
    if (p == NULL) {
        return 1; // allocation failure
    }

    *p = 42;
    printf("%d\n", *p);

    free(p); // deallocation
    return 0;
}
```

Important points:

- `malloc` **does not initialize** the memory,
- `free` must be called **exactly once** for each successful allocation.

Dynamic array allocation in C

```
int* tab = (int*)malloc(10 * sizeof(int));
```

Access:

```
tab[0] = 1;
tab[1] = 2;
```

Deallocation:

```
free(tab);
```

Dynamic allocation in C++: `new` and `delete`

In C++, we have the operators `new` and `delete`, which are **type-aware** and call constructors and destructors.

Allocation of an object:

```
int* p = new int(42);
```

Deallocation:

```
delete p;
```

For an array:

```
int* tab = new int[10];
```

Corresponding deallocation:

```
delete[] tab;
```

Fundamental rule:

- new ↔ delete
- new[] ↔ delete[]

Mixing them leads to undefined behavior.

Allocation of objects and constructor calls

```
struct Point {  
    float x, y;  
    Point(float a, float b) : x(a), y(b) {}  
};  
  
int main() {  
    Point* p = new Point(1.0f, 2.0f); // constructor called  
    delete p;                          // destructor called  
}
```

Dynamic allocation of an array (complete example)

```
#include <iostream>  
  
int* createArray(int n) {  
    int* arr = new int[n]; // allocation of n integers  
    for(int i=0; i<n; ++i)  
        arr[i] = i * 10;  
    return arr;  
}  
  
int main() {  
    int n = 5;  
    int* arr = createArray(n);  
  
    for(int i=0; i<n; ++i)  
        std::cout << arr[i] << " ";  
  
    delete[] arr; // deallocation mandatory  
}
```

Utility: n is known only at runtime -> impossible to use a static array.

Memory diagram

Stack (pile)	Heap (heap)
----- int main() {	new int[3]

```

int n = 3;
int* arr = new int[n]; --> | 0 | 1 | 2 | ...
}

```

- The stack contains the local variables (`n`, `arr`).
- The heap contains the dynamically allocated data.
- Heap memory is not automatically freed -> `delete[] arr;` is mandatory.

Common Problems

1. Memory leak:

```
void f() { int* p = new int(10); // forgetting to delete -> memory leak }
-> the memory remains allocated as long as the program runs.
```

2. Double free:

```
int* p = new int(5); delete p; delete p; // error: double free
This causes undefined behavior.
```

3. Use after free:

```
int* p = new int(5); delete p; std::cout << *p; // undefined behavior
```

Null pointer after deallocation

Best practice:

```

int* p = new int(5);
delete p;
p = nullptr;

```

This avoids accessing a freed pointer (*dangling pointer*).

Resizing (principle)

When manually resizing a dynamic array, you must:

1. Allocate a new space.
2. Copy the old data.
3. Free the old space.

```

Old array (@100) : [10 20 30]
New array (@320) : [10 20 30 40]
delete[] @100

```

Note: Expanding an array always requires a **new allocation + copying**, hence the cost.
Modern containers (`std::vector`) automate this process efficiently.

Dynamic structures: lists and graphs

Dynamic allocation also allows creating structures **linked** or **hierarchical**, where each element contains pointers to others.

Example: minimal linked list

```
struct Node {
    int value;
    Node* next;
};

int main() {
    Node* n1 = new Node{5, nullptr};
    Node* n2 = new Node{8, nullptr};
    // Note: the `->` operator allows accessing a member via a pointer.
    // `p->member` is equivalent to `(*p).member`.
    n1->next = n2;

    // traversal
    for(Node* p = n1; p != nullptr; p = p->next)
        std::cout << p->value << " ";

    // deallocation
    delete n2;
    delete n1;
}
```

Each element (Node) is allocated separately on the heap. **[Attention]:** You must remember to free each element to avoid leaks.

Summary

- Dynamic allocation happens on the **heap**.
- In C: malloc / free (raw memory, void*).
- In C++: new / delete (types + constructors).
- Every allocation must be paired with a deallocation.
- Common errors are: memory leaks, double frees, dangling pointers.
- In modern C++, prefer containers and safe abstractions.

Manual memory management is powerful but dangerous.

In C++, it must be limited to the necessary cases and replaced as much as possible by safe abstractions.

Modern best practices

In C++, today we avoid direct new / delete.

We prefer:

1. std::vector for dynamic arrays

Example:

```
#include <vector>
#include <iostream>

std::vector<int> createVector(int n) {
    std::vector<int> v(n);
    for(int i=0; i<n; ++i)
        v[i] = i * 10;
    return v; // automatic management
}

int main() {
    auto v = createVector(5);
    for(int x : v)
        std::cout << x << " ";
}
```

-> The memory is managed automatically (constructor / destructor).

2. Smart pointers (`std::unique_ptr`, `std::shared_ptr`)

Smart pointers are classes from the C++ standard library (`<memory>`) that encapsulate a raw pointer (`T*`) and **automatically manage the lifetime of the pointed-to resource**.

They follow the RAII principle: the resource is released automatically when the pointer goes out of scope (destruction of the object). Thus, there is no longer any need to call `delete` manually: memory is released as soon as the object is no longer used.

Example with `std::unique_ptr` Example:

```
#include <memory>
#include <iostream>

int main() {
    std::unique_ptr<int> p = std::make_unique<int>(42);
    std::cout << *p << std::endl;
} // automatic deletion here
```

Explanation:

- `std::unique_ptr<int>` owns the resource exclusively: a single pointer manages the allocated object.
- `std::make_unique<int>(42)` dynamically creates an `int` containing 42 and returns a `unique_ptr` that becomes its owner.
- When `p` goes out of scope (end of `main`), its destructor automatically calls `delete` on the object it manages.
- The memory is thus properly released, even in case of an exception or premature exit from the function.

Characteristics of `std::unique_ptr`:

- Ownership is unique (non-copyable).
- Lightweight, safe and very efficient.
- Ideal for representing exclusive ownership of a resource.

Example with `std::shared_ptr` Example:

```
#include <memory>
#include <iostream>

int main() {
    auto p1 = std::make_shared<int>(10);
    auto p2 = p1; // share the resource
    std::cout << *p2 << std::endl;
} // memory is freed when the last shared_ptr disappears
```

Explanation in detail:

- `std::shared_ptr` allows **multiple pointers** to share the same resource.
- Each copy (`p2 = p1;`) **increments an internal reference count**.
- When a `shared_ptr` is destroyed, the counter is decremented.
- When this counter reaches zero (no owners left), the destructor calls `delete` **automatically** on the resource.

Thus, memory is released exactly when it is no longer used by anyone.

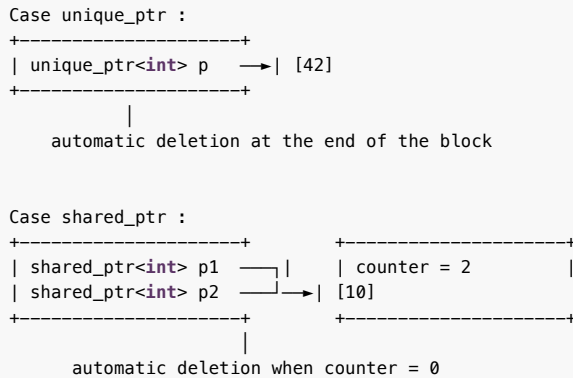
Characteristics of `std::shared_ptr`:

- Copyable: multiple instances can point to the same data.
- Reference counted: automatic destruction when the last owner disappears.
- Slightly more expensive than a `unique_ptr` (internal atomic counter).
- Ideal for shared structures or non-hierarchical graphs.

Comparison of the two smart pointer types

Type	Resource Copyable	Sharing	Destruction	Typical use case
<code>std::unique_ptr<T></code>	No	No	Automatic, as soon as the pointer goes out of scope	Exclusive ownership (e.g., internal component of an object)
<code>std::shared_ptr<T></code>	Yes	Yes (reference count)	Automatic, when the last pointer is destroyed	Resources shared between several objects or functions

Memory illustration



Why smart pointers replace `new` and `delete`

- They **avoid memory leaks** by automatically managing release.
- They **preserve safety** (no double free or dangling pointers).
- They **simplify code**: no need to call `delete`.
- They integrate naturally with the other classes of modern C++ (`std::vector`, `std::map`, `std::thread`, etc.).

3.8 Memory copy: `memcpy`

In C and C++, we often need to **copy a block of bytes** (array, struct, buffer received from the network/file, etc.). The standard function for that is `memcpy`, in `<string.h>` (C) or `<cstring>` (C++).

Prototype

```

#include <string.h>

void* memcpy(void* dest, const void* src, size_t n);

```

- `src` : source address
- `dest` : destination address
- `n` : number of bytes copied
- return : `dest`

Simple example: copy an array of integers

```

#include <stdio.h>
#include <string.h>

int main() {
    int a[3] = {10, 20, 30};
}

```

```

int b[3] = {0, 0, 0};

memcpy(b, a, 3 * sizeof(int));

for(int i=0; i<3; ++i)
    printf("%d ", b[i]); // 10 20 30
return 0;
}

```

Here, `memcpy` copies exactly `3 * sizeof(int)` bytes.

Example: copying a simple structure

```

#include <stdio.h>
#include <string.h>

typedef struct {
    int x;
    int y;
} Point2D;

int main() {
    Point2D p1 = {1, 2};
    Point2D p2;

    memcpy(&p2, &p1, sizeof(Point2D));

    printf("%d %d\n", p2.x, p2.y); // 1 2

    return 0;
}

```

Read a "raw buffer" and reconstruct types with `memcpy`

Typical case: we receive a byte array (network, binary file, sensor...) and we want to extract typed values from it. Suppose a binary message in the following format:

- `uint32_t` id
- `float` temperature
- `uint16_t` count

So: $4 + 4 + 2 = 10$ bytes.

```

#include <stdint.h>
#include <stdio.h>
#include <string.h>

int main() {
    // Simulated raw buffer (e.g., received from the network)
    uint8_t buf[10] = {
        0xD2, 0x04, 0x00, 0x00, // id = 1234 in little-endian
        0x00, 0x00, 0x48, 0x42, // float 50.0f in IEEE-754 (little-endian)
        0x07, 0x00 // count = 7 in little-endian
    };

    size_t offset = 0;

    uint32_t id;
    float temp;
    uint16_t count;

    memcpy(&id, buf + offset, sizeof(uint32_t));
    offset += sizeof(uint32_t);

    memcpy(&temp, buf + offset, sizeof(float));
    offset += sizeof(float);

    memcpy(&count, buf + offset, sizeof(uint16_t));
}

```

```
offset += sizeof(uint16_t);

printf("id=%u, temp=%.2f, count=%u\n", id, temp, count);
return 0;
}
```

3.9 The generic pointer `void*`

In C and C++, there exists a special pointer type: `void*`, called the **generic pointer**. A `void*` can contain the address of any data type, without knowing its nature.

It therefore represents a **raw address**, without any associated type information.

Declaration and principle

```
void* p;
```

Here:

- `p` can store the address of an `int`, a `float`, a struct, etc.
- the compiler **does not know** what `p` points to.

This means that:

- you can **store an address** in `p`,
- but you **cannot access the value pointed to directly**.

Simple example

```
#include <stdio.h>

int main() {
    int a = 42;
    float b = 3.14f;

    void* p;

    p = &a; // p points to an int
    p = &b; // p now points to a float

    return 0;
}
```

In this example:

- `p` can successively hold the address of `a` then that of `b`,
- but **no type information is preserved**.

Inability to dereference directly

It is **forbidden** to do:

```
void* p = &a;
printf("%d\n", *p); // ERROR
```

Why?

- `*p` means “access the value pointed to”,
- but the compiler does not know **either the size or the nature of the pointed-to type**.

The type `void` literally means: **absence of type information**.

Explicit conversion (cast)

To access the pointed-to value, you must **explicitly convert** the `void*` to the appropriate pointer type.

```
#include <stdio.h>

int main() {
    int a = 42;
    void* p = &a;

    int* pi = (int*)p;    // explicit cast
    printf("%d\n", *pi); // OK

    return 0;
}
```

Steps:

1. `p` contains the address of `a`,
2. we explicitly tell the compiler: "treat this address as an `int*`",
3. and we can then dereference correctly.

Example with several types

```
#include <stdio.h>

void print_value(void* data, char type)
{
    if (type == 'i') {
        printf("int : %d\n", *(int*)data);
    }
    else if (type == 'f') {
        printf("float : %f\n", *(float*)data);
    }
}

int main() {
    int a = 10;
    float b = 2.5f;

    print_value(&a, 'i');
    print_value(&b, 'f');

    return 0;
}
```

Here:

- the `void*` allows passing **any type**,
- but one must manually handle the correct interpretation.

Link with pointer arithmetic

Unlike other pointers (`int*`, `double*`, etc.), **pointer arithmetic is forbidden on `void*` in C++**.

```
void* p;
p + 1; // ERREUR en C++
```

Reason:

- `p + 1` requires knowing `sizeof(type)`,
- whereas `void` has **no size**.

In C (but not in C++), some compilers allow `void*` as a non-standard extension, treating it as a `char*`.

void* and arrays / raw memory

The `void*` is often used to manipulate **raw memory**, for example with `malloc`, `memcpy`, or low-level APIs.

Example :

```
#include <stdlib.h>

int main() {
    void* buffer = malloc(100); // 100 octets de mémoire brute

    // interprétation explicite
    int* tab = (int*)buffer;
    tab[0] = 42;

    free(buffer);
    return 0;
}
```

Here:

- `malloc` returns a `void*`,
- the programmer then decides **how to interpret** this memory.

A more complete example of using void*

Here's a typical example of using `void*` : we receive a block of raw bytes (network, file, sensor frame, image, ...), stored in a `void*`, then we reconstruct an "interpretable" structure.

Let's imagine a server that sends a binary message composed of :

1. a header with :

- `uint32_t id`
- `uint16_t width`
- `uint16_t height`

2. then data (payload) : here, for example, a grayscale image of size `width * height` bytes.

We receive the information as a **raw buffer** (typically `void* + size`) that we must "restructure".

- The general principle is as follows:
 1. **recognize** the structure (the header),
 2. **calculate** where the useful data begins,
 3. make **casts** (often via `uint8_t*` to do byte arithmetic),
 4. check the **sizes** (otherwise crash / vulnerability).

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#pragma pack(push, 1) // to avoid padding (compiler/ABI dependent)
typedef struct {
    uint32_t id;
    uint16_t width;
    uint16_t height;
} Header;
#pragma pack(pop)

int main() {
    // --- Simulation : "réception réseau" d'un bloc brut ---
    // On fabrique un buffer qui contient : Header + pixels
    Header h = { .id = 1234, .width = 4, .height = 3 };
    uint8_t pixels[12] = {
        10, 20, 30, 40,
        50, 60, 70, 80,
        90, 100, 110, 120
    };
}
```

```

};

size_t total = sizeof(Header) + sizeof(pixels);
void* buffer = malloc(total);

memcpy(buffer, &h, sizeof(Header));
memcpy((uint8_t*)buffer + sizeof(Header), pixels, sizeof(pixels));

// --- Reconstruction / interpretation ---
// 1) Read the header
Header header;
memcpy(&header, buffer, sizeof(Header));

printf("id=%u, width=%u, height=%u\n",
       header.id, header.width, header.height);

// 2) Access the payload (image) after the header
size_t image_size = (size_t)header.width * (size_t)header.height;

// Minimal coherence check
if (sizeof(Header) + image_size > total) {
    printf("Incomplete or corrupted buffer!\n");
    free(buffer);
    return 1;
}

uint8_t* image = (uint8_t*)buffer + sizeof(Header);

// Example: print the pixels (line by line)
for (uint16_t y = 0; y < header.height; ++y) {
    for (uint16_t x = 0; x < header.width; ++x) {
        printf("%3u ", image[y * header.width + x]);
    }
    printf("\n");
}

free(buffer);
return 0;
}

```

Practical usage

The `void*` is mainly used:

- in pure C (generic interfaces, system libraries),
- for low-level APIs,
- for handling raw memory,
- in historical generic functions (`qsort`, `bsearch`).

In modern C++, one prefers:

- templates,
- typed pointers,
- containers (`std::vector`, `std::array`),
- smart pointers (`std::unique_ptr`, `std::shared_ptr`).

Key takeaway

`void*` is a pointer without type information: it offers great flexibility, but **no safety**. Any correct usage relies on explicit conversions and the programmer's rigor.

3.10 References

In C++, references are introduced as a simpler and safer alternative to pointers. They can be seen as an **alias** to an existing variable, and above all as a **syntactic sugar** over the notion of a pointer:

- Like a pointer, a reference allows you to work directly on an original variable without making a copy.
- Unlike a pointer, you do not need to write * or -> : the reference is manipulated as the variable itself.

Argument passing: value, pointer, reference

Pass by value (default in C/C++)

```
#include <iostream>

int my_function(int b) {
    b = b + 2; // modifies local copy
    return b;
}

int main() {
    int a = 5;
    int c = my_function(a);
    std::cout << a << ", " << c << std::endl; // a=5, c=7
}
```

Here:

- b is a **copy** of a.
- Modifying b does not affect a.

Pass by address with pointer (C style)

```
#include <iostream>

void my_function(int* b) {
    *b = *b + 2; // modifies the value pointed to
}

int main() {
    int a = 5;
    my_function(&a); // we pass the address of a

    std::cout << a << std::endl; // prints 7
}
```

Here:

- b is a copy of the pointer to a.
- We must use *b to access/modify the value.
- Heavier syntax, with a risk of errors (null pointer, forgetting the *).

Pass by reference (C++ style)

```
#include <iostream>

void ma_fonction(int& b) {
    b = b + 2; // you feel like you're manipulating b as a variable
}

int main() {
    int a = 5;
    ma_fonction(a); // no &
    std::cout << a << std::endl; // prints 7
}
```

Here:

- `b` is a **reference** alias of `a`.
- No special syntax, we manipulate `b` as if it were a local variable.
- It's a *syntactic sugar*: behind the scenes, the compiler generates pass-by-address, but the syntax is simplified.

Initialization of references

A reference must always be **initialized** at the time of declaration:

```
int main() {
    int a = 5;
    int& ref_a = a; // OK: ref_a is an alias of a
    ref_a = 9;     // modifies a

    int& ref_b;   // ERROR: a reference must be initialized
}
```

Unlike a pointer, a reference:

- cannot be null,
- cannot be reassigned to another variable after initialization.

Constant references

A **constant reference** (`const &`) allows you to:

- avoid a costly copy,
- while guaranteeing that the object will not be modified.

```
#include <iostream>
#include <string>

void printMessage(const std::string& msg) {
    std::cout << msg << std::endl;
}

int main() {
    std::string text = "Hello";
    printMessage(text); // no copy, and safety guaranteed
}
```

Constant references are widely used to pass large objects (vectors, strings, structures) **without copying**.

Concrete example: vectors and structures

```
#include <iostream>

struct vec4 {
    double x, y, z, w;
};

// pass by reference to modify
void multiply(vec4& v, double s) {
    v.x *= s; v.y *= s; v.z *= s; v.w *= s;
}

// pass by constant reference to avoid a copy
void print(const vec4& v) {
    std::cout << v.x << " " << v.y << " " << v.z << " " << v.w << std::endl;
}

int main() {
    vec4 v = {1.1, 2.2, 3.3, 4.4};
    multiply(v, 2.0); // modifies v
    print(v);       // prints without copying
}
```

Accessors by reference

In C++, references are very handy for writing accessors:

```
class Vec50 {
private:
    float T[50];
public:
    void init() {
        for(int k=0; k<50; ++k)
            T[k] = static_cast<float>(k);
    }

    // read-only accessor
    float value(unsigned int i) const {
        return T[i];
    }

    // read/write accessor: returns a reference
    float& value(unsigned int i) {
        return T[i];
    }
};

int main() {
    Vec50 v;
    v.init();

    std::cout << v.value(10) << std::endl; // read
    v.value(10) = 42;                       // write via reference
}

std::cout << v.value(10) << std::endl;
}
```

Best practices

To do

- Use references to simplify code compared to pointers.
- Use `const &` to pass heavy objects (vectors, strings, classes).
- Return a reference if the goal is to allow modification (setter `set`).

To avoid

- Do not overuse non-const references in function parameters -> the reader should understand immediately whether a variable is modified.
- Never return a reference to a local variable (it no longer exists after the function exits).

Summary

- A **reference** is an alias of a variable.
- It is implemented like a pointer, but with a simplified syntax (*syntactic sugar*).
- Constant references (`const &`) are fundamental for writing safe and efficient code.
- When used well, references combine the power of pointers and the readability of clean code.

4 Classes

4.1 Introduction

In C++, a **class** allows grouping in a single entity of **data** (called *attributes*) and **functions** (called *methods*) that manipulate these data. An instance of a class is called an **object**. This organization facilitates code structuring, readability, and maintenance.

Grouping data: first example with struct

We often start with a `struct` to represent a simple object:

```
struct vec3 {
    float x;
    float y;
    float z;
};
```

Here, `vec3` groups three values representing a 3D vector. The members are **public by default**, which means they are directly accessible:

```
vec3 v;
v.x = 1.0f;
v.y = 2.0f;
v.z = 3.0f;
```

This type of structure is well suited for **simple data aggregates**, very common in computer graphics.

Add behavior: methods

A class or a struct can also contain **member functions**:

```
#include <cmath>

struct vec3 {
    float x, y, z;

    float norm() const {
        return std::sqrt(x*x + y*y + z*z);
    }
};
```

The method `norm()` operates directly on the attributes `x`, `y` and `z` of the object:

```
vec3 v{1.0f, 2.0f, 2.0f};
float n = v.norm(); // n = 3
```

Note: the `const` placed after the signature of a method (here `norm() const`) indicates that the method **does not modify the object's state**. A `const` method can be called on a `const` object, and the compiler forbids any modification of non-mutable members inside this method.

The implicit `this` pointer

In the methods of a class, the compiler provides **implicitly** a pointer named `this` that points to the current object. It is useful for explicitly accessing members, disambiguating parameters, and returning a reference to the object.

Example :

```
struct S {
    int x;
    void set(int x) { this->x = x; } // disambiguates the field x
    int get() const { return this->x; } // this is const
};
```

This notion is basic but important: `this` allows manipulating the current object inside methods and makes explicit certain operations (transfer of *ownership*, return of `*this`, ...).

struct vs class

The keyword `class` works exactly like `struct`, with the difference that: the members are **private by default**.

```
class vec3 {
    float x, y, z; // private by default
};
```

This code **does not compile** :

```
vec3 v;
v.x = 1.0f; // ERROR: x is private
```

To make certain members accessible, you must specify the access levels.

Public and private attributes

The keywords `public` and `private` are used to control access to the members :

```
class vec3 {
public:
    vec3(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}

    float norm() const {
        return std::sqrt(x*x + y*y + z*z);
    }

private:
    float x, y, z;
};
```

Usage:

```
vec3 v(1.0f, 2.0f, 2.0f);

float n = v.norm(); // OK
// v.x = 3.0f;      // ERROR: x is private
```

Here:

- the **attributes are private** → protected against uncontrolled modifications,
- the **methods are public** → interface accessible to the user.

Encapsulation and Security

Thanks to this encapsulation, the object guarantees its internal coherence. For example, we can enforce certain rules:

```
class Circle {
public:
    Circle(float radius) {
        set_radius(radius);
    }

    float area() const {
        return 3.14159f * r * r;
    }

    void set_radius(float radius) {
        if (radius > 0.0f)
            r = radius;
    }

private:
```

```
float r;  
};
```

Here, the radius can never become negative, because direct access to `r` is forbidden.

Best practices

- Use `struct` for:
 - simple objects,
 - primarily data carriers,
 - without complex invariants.
- Use `class` for:
 - encapsulating data,
 - controlling access,
 - guaranteeing internal invariants.

4.2 Initialization, Constructors

In C++, object initialization is handled by the **constructors**. A constructor is a special function (same name as the class, no return type) called automatically when the object is created. Its purpose is to guarantee that the object is in a **valid state** from the start.

Classic problem: uninitialized attributes

If a class/struct contains fundamental types (`int`, `float`, etc.), they are not necessarily initialized automatically.

```
#include <iostream>  
  
struct vec3 {  
    float x, y, z;  
};  
  
int main() {  
    vec3 v; // x,y,z undefined !  
    std::cout << v.x << std::endl; // indeterminate behavior  
}
```

In the case of an aggregate struct, you can force zero initialization with `{}` :

```
vec3 v{}; // x=y=z=0
```

But as soon as we want to precisely control the state of the object, we use constructors.

Default constructor

The default constructor takes no arguments. It is often used to set coherent values.

```
struct vec3 {  
    float x, y, z;  
  
    vec3() : x(0.0f), y(0.0f), z(0.0f) {}  
};  
  
int main() {  
    vec3 v; // calls vec3()  
}
```

Here, `v` is guaranteed valid: its fields are 0.

Initialization list

The expression : $x(\dots)$, $y(\dots)$, $z(\dots)$ is the initialization list. It initializes the attributes before entering the constructor body.

```
struct vec3 {
    float x, y, z;

    vec3(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}
};
```

Usage :

```
vec3 v(1.0f, 2.0f, 3.0f);
vec3 w{1.0f, 2.0f, 3.0f}; // uniform (often recommended)
```

This list is preferable to an assignment in the constructor body, because it avoids a “double step” (construction followed by reassignment) and it is required for certain members.

Overloaded constructors

We can define several constructors to offer different ways of creating an object.

```
struct vec3 {
    float x, y, z;

    vec3() : x(0), y(0), z(0) {}
    vec3(float v) : x(v), y(v), z(v) {}
    vec3(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}
};

int main() {
    vec3 a; // (0,0,0)
    vec3 b(1.0f); // (1,1,1)
    vec3 c(1.0f,2.0f,3.0f); // (1,2,3)
}
```

Single-argument constructor and explicit

A single-argument constructor can serve as an implicit conversion, which can cause side effects. The keyword `explicit` prevents these automatic conversions.

```
struct vec3 {
    float x, y, z;

    explicit vec3(float v) : x(v), y(v), z(v) {}
};
```

```
vec3 a(1.0f); // OK
// vec3 b = 1.0f; // forbidden thanks to explicit
```

This makes the code safer and more readable.

const Members and References: constructor required

The `const` attributes and references must be initialized via the initializer list.

```
struct sample {
    int const id;
    float& ref;

    sample(int id_, float& ref_) : id(id_), ref(ref_) {}
};
```

Without an initializer list, this code won't compile, because `id` and `ref` cannot be “assigned” after the fact: they must be initialized immediately.

Destructor (reminder)

The destructor is automatically called when the object is destroyed (end of scope, `delete`, etc.). It is mainly used to release resources (files, memory, GPU...).

```
#include <iostream>

struct tracer {
    tracer() { std::cout << "Constructed\n"; }
    ~tracer() { std::cout << "Destroyed\n"; }
};

int main() {
    tracer t; // "Constructed"
} // "Destroyed"
```

Best practices

- Always initialize the attributes (via constructor or `{}`).
- Prefer the initializer list `:` to initialize members.
- Use `explicit` for single-argument constructors, unless the implicit conversion is desired.
- Design constructors to guarantee that objects are always valid.

4.3 Operators

In C++, it is possible to **overload operators** for classes and structures to make their use more natural and expressive. This feature is particularly useful in computer graphics, where one frequently manipulates vectors, matrices, colors, or transformations, and where expressions such as `v1 + v2` or `2.0f * v` are much more readable than an explicit function call.

General principle

Operator overloading consists of defining a **special function** named `operator<symbol>`. From the compiler's point of view, an expression like:

```
a + b
```

is translated to:

```
operator+(a, b);
```

or, in the case of a member operator:

```
a.operator+(b);
```

Overloading does not create a new operator: it simply redefines the behavior of an existing operator for a given type.

Member and non-member operators

An operator can be defined:

- as **member method** of the class,
- or as **non-member function** (often preferable for symmetric operators).

Common rule:

- operators that **modify the object** (`+=`, `*=`, `[]`, etc.) are often member methods;
- symmetric binary operators (`+`, `-`, `*`) are often non-member functions.

Example: arithmetic operators for a 3D vector

```
struct vec3 {
    float x, y, z;

    vec3() : x(0), y(0), z(0) {}
    vec3(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}

    vec3& operator+=(vec3 const& v) {
        x += v.x;
        y += v.y;
        z += v.z;
        return *this;
    }
};
```

The += operator modifies the current object and returns a reference to it.
We then define + as a non-member operator by reusing +=:

```
vec3 operator+(vec3 a, vec3 const& b) {
    a += b;
    return a;
}
```

Usage:

```
vec3 a{1,2,3};
vec3 b{4,5,6};

vec3 c = a + b; // (5,7,9)
a += b;        // a becomes (5,7,9)
```

Operators with different types

We can define operators between different types, for example scalar multiplication:

```
vec3 operator*(float s, vec3 const& v) {
    return vec3{s*v.x, s*v.y, s*v.z};
}

vec3 operator*(vec3 const& v, float s) {
    return s * v;
}
```

This allows natural notation:

```
vec3 v{1,2,3};
vec3 w = 2.0f * v;
```

Comparison operators

Comparison operators allow comparing objects:

```
bool operator==(vec3 const& a, vec3 const& b) {
    return a.x == b.x && a.y == b.y && a.z == b.z;
}

bool operator!=(vec3 const& a, vec3 const& b) {
    return !(a == b);
}
```

Since C++20, there is also the <=> (three-way comparison) operator, but its use goes beyond the scope of this introduction.

The [] access operator

The [] operator is often used to provide indexed access to internal data:

```
struct vec3 {
    float x, y, z;

    float& operator[](int i) {
        return (&x)[i]; // contiguous access
    }

    float const& operator[](int i) const {
        return (&x)[i];
    }
};
```

Usage:

```
vec3 v{1,2,3};
v[0] = 4.0f;
float y = v[1];
```

The const version is essential to allow read access on a constant object.

The << display operator

To facilitate debugging, we often overload the << operator with std::ostream:

```
#include <iostream>

std::ostream& operator<<(std::ostream& out, vec3 const& v) {
    out << "(" << v.x << ", " << v.y << ", " << v.z << ")";
    return out;
}
```

Usage:

```
vec3 v{1,2,3};
std::cout << v << std::endl;
```

Best practices

- Always use **const references** for read-only parameters.
- Return `*this` by reference for modifying operators (`+=`, `*=`, etc.).
- Avoid overloads that make the code ambiguous or counterintuitive.
- Do not overload an operator if its mathematical or logical meaning is not clear.

Operator overloading allows you to write code that is more readable and more expressive, but it must remain **simple, coherent and predictable**.

4.4 Inheritance

The **inheritance** is a central mechanism of object-oriented programming that allows you to **define a new class from an existing one**. The derived class inherits the attributes and methods of the base class, which fosters the **code reuse** and the hierarchical structuring of concepts. In C++, inheritance is often used to factor out common behaviors while allowing specializations.

General principle

We define a derived class by indicating the base class after :

```
class Derived : public Base {
    // contenu spécifique à Derived
};
```

The keyword `public` indicates that the public interface of the base class remains public in the derived class. This is the most common case and the one used in the majority of object-oriented designs.

Simple example of inheritance

Consider a base class representing a geometric shape :

```
class Shape {
public:
    float x, y;

    Shape(float x_, float y_) : x(x_), y(y_) {}

    void translate(float dx, float dy) {
        x += dx;
        y += dy;
    }
};
```

We can define a derived class that specializes this behavior :

```
class Circle : public Shape {
public:
    float radius;

    Circle(float x_, float y_, float r_)
        : Shape(x_, y_), radius(r_) {}
};
```

Usage :

```
Circle c(0.0f, 0.0f, 1.0f);
c.translate(1.0f, 2.0f); // method inherited from Shape
```

The `Circle` class automatically inherits `x`, `y` and the `translate` method.

Constructors and inheritance

The constructor of the derived class **must explicitly call** the base class constructor in its initializer list.

```
Circle(float x_, float y_, float r_)
    : Shape(x_, y_), radius(r_) {}
```

If the base class constructor is not called explicitly, the compiler will attempt to call the default constructor, which can lead to an error if it does not exist.

Access to members: public, protected, private

The access level of the base class members determines their visibility in the derived class:

- `public` : accessible everywhere, including in derived classes.
- `protected` : accessible only within the class and its derivatives.
- `private` : accessible only within the base class.

```
class Shape {
protected:
    float x, y;
```

```

public:
    Shape(float x_, float y_) : x(x_), y(y_) {}
};

```

```

class Circle : public Shape {
public:
    float radius;

    Circle(float x_, float y_, float r_)
        : Shape(x_, y_), radius(r_) {}

    float center_x() const {
        return x; // allowed because x is protected
    }
};

```

Method overriding

An derived class can **redefine** a method of the base class to provide a specific behavior.

```

class Shape {
public:
    float x, y;

    Shape(float x_, float y_) : x(x_), y(y_) {}

    float area() const {
        return 0.0f;
    }
};

```

```

class Rectangle : public Shape {
public:
    float w, h;

    Rectangle(float x_, float y_, float w_, float h_)
        : Shape(x_, y_), w(w_), h(h_) {}

    float area() const {
        return w * h;
    }
};

```

Here, `Rectangle::area` masks the version defined in `Shape`. This mechanism naturally prepares the introduction of **polymorphism**, which will be studied in the next chapter.

Inheritance and code factoring

Inheritance helps avoid duplication:

```

class Vehicle {
public:
    float speed;

    void accelerate(float dv) {
        speed += dv;
    }
};

class Car : public Vehicle {
    // specific behavior
};

class Plane : public Vehicle {
    // specific behavior
};

```

The classes `Car` and `Plane` share the same base behavior without duplication.

Best practices

- Use inheritance to express an **is-a** relationship (*is-a*).
- Prefer base classes that are **simple and stable**.

4.5 Polymorphism

The **polymorphism** allows you to manipulate objects of different types **through a common interface**, while automatically calling the correct implementation according to the **real type** of the object. In C++, it relies on inheritance, the **virtual functions** and the use of **pointers or references** to a base class. It is particularly useful when you want to **store heterogeneous objects in a single container** and treat them in a uniform way.

The problem: store different objects in a single container

Suppose we want to represent different geometric shapes and calculate their total area.

```
struct Circle {
    float r;
    float area() const {
        return 3.14159f * r * r;
    }
};

struct Rectangle {
    float w, h;
    float area() const {
        return w * h;
    }
};
```

These two types have an `area()` method, but **they have no type relationship**. Therefore it is impossible to write:

```
std::vector<Circle> shapes; // only circles
std::vector<Rectangle> shapes; // only rectangles
```

and especially impossible to do:

```
std::vector</* Circle and Rectangle */> shapes; // impossible
```

Without polymorphism, we are constrained either:

- to duplicate the code,
- to use type tests,
- or to design an artificial structure grouping all possible cases.

Polymorphism provides an elegant solution to this problem.

Common interface via a base class

We begin by defining a **base class** representing the general concept of “shape”:

```
class Shape {
public:
    virtual float area() const = 0; // pure virtual function
    virtual ~Shape() = default;
};
```

This class is **abstract**:

- it defines an interface,
- it cannot be instantiated.

Specialized Derived Classes

Each concrete shape inherits from Shape and implements area():

```
// Remarque : le mot-clé 'override' (C++11) indique au compilateur  
// que la méthode redéfinit une méthode virtuelle de la classe de base.  
// Il provoquera une erreur de compilation si la signature ne correspond pas.  
class Circle : public Shape {  
public:  
    float r;  
  
    explicit Circle(float r_) : r(r_) {}  
  
    float area() const override {  
        return 3.14159f * r * r;  
    }  
};
```

```
class Rectangle : public Shape {  
public:  
    float w, h;  
  
    Rectangle(float w_, float h_) : w(w_), h(h_) {}  
  
    float area() const override {  
        return w * h;  
    }  
};
```

Polymorphic storage in a container

Thanks to inheritance and virtual functions, we can now store **pointers to the base class** in the same container:

```
#include <vector>  
#include <memory>  
  
int main() {  
    std::vector<std::unique_ptr<Shape>> shapes;  
  
    shapes.push_back(std::make_unique<Circle>(2.0f));  
    shapes.push_back(std::make_unique<Rectangle>(3.0f, 4.0f));  
  
    float total_area = 0.0f;  
    for (auto const& s : shapes) {  
        total_area += s->area(); // polymorphic call  
    }  
}
```

Here:

- the container only knows the type Shape,
- each element points to an object of a different concrete type,
- the call to area() is resolved **dynamically** according to the actual type (Circle OR Rectangle).

Role of virtual and dynamic dispatch

The call:

```
s->area();
```

is resolved at runtime thanks to the **virtual table**:

- if s points to a Circle, Circle::area() is called,
- if s points to a Rectangle, Rectangle::area() is called.

This is the heart of dynamic polymorphism.

Importance of the virtual destructor

Objects are destroyed via a pointer to the base class. The destructor must therefore be virtual:

```
class Shape {
public:
    virtual ~Shape() = default;
};
```

Without this, the destructor of the derived class would not be called, which could lead to resource leaks.

Why pointers and not objects?

You cannot directly store derived objects in a container of type `std::vector<Shape>` because that would cause a **slicing** (loss of the derived part). Pointers (often smart pointers) avoid this problem and enable dynamic binding.

Cost and alternatives

Dynamic polymorphism implies:

- an indirection,
- a call overhead slightly higher than a non-virtual function.

In very performance-critical loops, static polymorphism via templates may sometimes be preferred, discussed later.

Use of raw pointers (*)

In the previous examples, we used **smart pointers** (`std::unique_ptr`) to automatically manage the lifetime of objects. It is, however, important to understand that polymorphism in C++ historically works with **raw pointers** (`Shape*`). These offer more freedom, but require manual memory management, which greatly increases the risk of errors.

Example with raw pointers

```
#include <vector>

int main() {
    std::vector<Shape*> shapes;

    shapes.push_back(new Circle(2.0f));
    shapes.push_back(new Rectangle(3.0f, 4.0f));

    float total_area = 0.0f;
    for (Shape* s : shapes) {
        total_area += s->area(); // polymorphic call
    }

    // Manual release of memory
    for (Shape* s : shapes) {
        delete s;
    }
}
```

Here:

- the objects are dynamically allocated with `new`,
- the container stores pointers to the base class `Shape`,
- the calls to `area()` are resolved dynamically,
- **the programmer must explicitly free the memory** with `delete`.

Critical role of the virtual destructor

With raw pointers, the virtual destructor is absolutely indispensable:

```
class Shape {
public:
    virtual ~Shape() = default;
};
```

Without a virtual destructor, the call:

```
delete s;
```

would destroy only the `Shape` part of the object, and not the derived part (`Circle`, `Rectangle`), leading to resource leaks and undefined behavior.

Common problems with raw pointers

Using raw pointers exposes several classic mistakes:

- forgetting to `delete` → **memory leak** ;
- double `delete` → **undefined behavior** ;
- deletion in the wrong order ;
- exception or early return preventing release ;
- confusion about who is responsible for destruction.

These problems are difficult to detect and fix, especially in large-scale projects.

Best practices

- Use polymorphism to solve problems of **uniform treatment of heterogeneous objects**.
- Define abstract base classes as interfaces.
- Always declare a virtual destructor in a polymorphic hierarchy.
- Use `override` to ensure safe overrides.
- Combine polymorphism with smart pointers (`std::unique_ptr`).

Polymorphism thus enables designing extensible systems where new types can be added without modifying existing code, especially when dealing with collections of varied objects.

4.6 Access control: `const`

In C++, the keyword `const` applied to **class methods** plays a central role in access control and in code safety. It is not merely a documentary indicator: a `const` method and a non-`const` method are considered by the compiler as two distinct methods, capable of coexisting in the same class with the same name.

Meaning of a `const` method

A method declared with `const` after its signature guarantees that it **does not modify the object's state**.

```
class vec3 {
public:
    float x, y, z;

    float norm() const {
        return std::sqrt(x*x + y*y + z*z);
    }
};
```

The `const` here means that the method cannot modify `x`, `y` or `z`. Any attempt to modify would cause a compilation error.

```
float norm() const {
    x = 0.0f; // ERREUR : modification interdite
    return 0.0f;
}
```

Constant objects and accessible methods

A const object can call only **const methods**.

```
const vec3 v{1.0f, 2.0f, 3.0f};

v.norm(); // OK
// v.normalize(); // ERREUR if normalize() is not const
```

This naturally imposes a clear separation between:

- the **read** methods (access, calculations),
- the **modification** methods (updating the state).

const and non-const methods: two distinct signatures

A const method and a non-const method with the same name are not the same function. They can be defined simultaneously in a class.

```
class vec3 {
public:
    float x, y, z;

    float& operator[](int i) {
        return (&x)[i];
    }

    float const& operator[](int i) const {
        return (&x)[i];
    }
};
```

Here:

- the **non const** version is called on a modifiable object,
- the **const** version is called on a constant object.

Usage:

```
vec3 a{1,2,3};
a[0] = 5.0f; // calls the non-const version

const vec3 b{1,2,3};
float x = b[0]; // calls the const version
```

The compiler automatically selects the appropriate version based on the **const-ness of the object**.

Classic example: read/write accessor

```
class Buffer {
public:
    float& value() {
        return data;
    }

    float value() const {
        return data;
    }
};
```

```
private:
    float data;
};
```

Here:

- `value()` (non-const) allows modifying the data,
- `value() const` allows only reading it.

```
Buffer b;
b.value() = 3.0f; // non-const version

const Buffer c;
// c.value() = 3.0f; // ERREUR
float v = c.value(); // const version
```

Conceptual Interest

This distinction allows:

- to clearly express the intentions of the code,
- to guarantee that certain operations have no side effects,
- to detect errors at compile time,
- to write more robust interfaces.

In a well-structured design, the majority of methods should be `const`.
Non-const methods correspond to **explicit modification operations**.

Best practices

- Mark every method that does not modify the object as `const`.
- Always provide a `const` and non-`const` version when access may be read or written.
- Consider a `const` method and a non-`const` method as **two distinct contracts**.
- Use `const` as a design tool, not merely as a syntactic constraint.

4.7 Access control: the `static` keyword in classes

The keyword `static`, applied to the members of a class, profoundly changes their **nature** and their **lifetime**. A static member does not belong to an **object**, but to the **class itself**. It is therefore **shared by all instances** of this class. This mechanism is essential for representing global data or behaviors related to a concept, rather than to a particular object.

Static attributes

A **static attribute** is unique for the entire class, regardless of how many objects are created.

```
class Counter {
public:
    Counter() {
        ++count;
    }

    static int get_count() {
        return count;
    }

private:
    static int count;
};
```

The declaration in the class **is not enough**. The static attribute must be **defined once** in a `.cpp` file:

```
int Counter::count = 0;
```

Usage:

```
Counter a;  
Counter b;  
Counter c;  
  
int n = Counter::get_count(); // n = 3
```

All Counter objects share the **same variable** count.

Access to static attributes

A static attribute:

- can be accessed **without an object**, via the class name,
- can also be accessed from an object, but this is not recommended.

```
Counter::get_count(); // form recommended
```

This underscores that the data belongs to the class, not to a particular instance.

Static methods

A **static method** is a function associated with the class, but independent of any instance.

```
class MathUtils {  
public:  
    static float square(float x) {  
        return x * x;  
    }  
};
```

Usage:

```
float y = MathUtils::square(3.0f);
```

Constraints of static methods

A static method:

- **does not have a this pointer**,
- can access only the **static members** of the class,
- cannot directly access non-static attributes.

```
class Example {  
public:  
    static void f() {  
        // x = 3; // ERREUR : x n'est pas statique  
        y = 4;    // OK  
    }  
  
private:  
    int x;  
    static int y;  
};
```

static and initialization

Since C++17, it is possible to initialize directly certain static attributes in the class if they are `constexpr` or of literal type.

```
class Physics {
public:
    static constexpr float gravity = 9.81f;
};
```

Usage:

```
float g = Physics::gravity;
```

In this case, no additional definition in a `.cpp` is necessary.

Common use cases

The keyword `static` is used for:

- count the number of instances of a class,
- store global constants related to a concept,
- share common resources,
- group utility functions related to a class,
- implement factories (*factory methods*).

Example: unique identifier per object

```
class Object {
public:
    Object() : id(next_id++) {}

    int get_id() const {
        return id;
    }

private:
    int id;
    static int next_id;
};

int Object::next_id = 0;
```

Each object receives a unique identifier, generated from a shared counter.

Best practices

- Use `static` to express a **class-level** belonging, not to the object.
- Access static members via `NomClasse::membre`.
- Limit the use of mutable static attributes to avoid hidden dependencies.
- Prefer `static constexpr` for constants known at compile time.

Key idea to remember

A static member is **unique and shared**, it belongs to the **class**, not to the objects.

4.8 Namespaces (namespace)

When a project grows, it becomes common to have **identical names** in different parts of the code: `vec3`, `add`, `normalize`, `load`, etc. In C++, a **namespace** allows you to **group** functions, types and constants under a common prefix, in order to:

- **avoid name conflicts** between modules/libraries,

- **structure** the code by domains (math, io, gpu, ...),
- make the API more **clear** and more **predictable**.

The most famous example is the standard library: `std::vector`, `std::string`, `std::cout`.

Declaration and usage

A namespace creates a “box” logically:

```
namespace math {
    struct vec3 {
        float x, y, z;
    };

    float dot(vec3 const& a, vec3 const& b)
    {
        return a.x*b.x + a.y*b.y + a.z*b.z;
    }
} // namespace math
```

Usage:

```
math::vec3 a{1,2,3};
math::vec3 b{4,5,6};

float p = math::dot(a, b);
```

Here, `math::` is the **qualifier**: it disambiguates symbols.

Example: avoid a name conflict

Two libraries may offer a `load()` function but for different purposes. Without a namespace, this becomes ambiguous.

```
namespace io {
    int load(char const* filename) { /* ... */ return 0; }
}

namespace gpu {
    int load(char const* shader_file) { /* ... */ return 1; }
}
```

Explicit and unambiguous usage:

```
int a = io::load("mesh.obj");
int b = gpu::load("shader.vert");
```

using : importing names (with caution)

There are two syntaxes:

1) Import a specific name (recommended)

```
using math::vec3;

vec3 v{1,2,3}; // equivalent to math::vec3
```

2) Import an entire namespace (to be avoided in a header)

```
using namespace std;
```

This allows you to write `vector` instead of `std::vector`, but it can cause conflicts.

Best practice:

- using `namespace ...;` is acceptable in a small local `.cpp`,
- **to avoid in a `.hpp`**, as it pollutes all files that include this header.

Nested namespaces

We can structure by modules:

```
namespace engine {
namespace math {
    struct vec2 { float x, y; };
}
namespace io {
    void save();
}
}
```

Since C++17, you can write more simply:

```
namespace engine::math {
    struct vec2 { float x, y; };
}
```

Anonymous namespaces (local visibility)

An anonymous namespace makes symbols **visible only in the current file** (equivalent to `static` for global functions, but more general).

```
namespace {
    int helper(int x) { return 2*x; }
}

int f(int a)
{
    return helper(a);
}
```

Benefits:

- avoid exposing internal functions to the rest of the project,
- limit the surface of the public API.

Namespace aliases

Useful if a name is long:

```
namespace em = engine::math;

em::vec2 v{1,2};
```

Best practices

- Use namespaces to structure a project (e.g., `engine::math`, `engine::io`, `engine::render`).
- Keep `using namespace ...;` out of headers.
- Prefer `using name::symbol;` rather than importing the entire namespace.
- Use an anonymous namespace for implementation details in a `.cpp`.
- Design a stable public API via a clear namespace (e.g., `myproject::`).

5 Threads and Parallelism

Parallelism refers to the ability of a program to execute **multiple tasks simultaneously**. In C++, this concept is directly linked to **threads**, which allow exploiting the **multiple cores** of modern processors. Understanding threads is essential to write fast, but also safe and correct programs.

5.1 Concept of a thread

A **thread** is an **execution thread** independent inside the same program.

- A classic program has **one thread** (sequential execution).
- A multithreaded program has **several threads**, executed in parallel or near-parallel.

All the threads in the same program:

- share the **same memory space** (heap, global variables),
- each have their own **execution stack** (local variables, function calls).

(Quick reminder: in C++ we often manipulate threads via the `std::thread` class provided in `<thread>`.)

5.2 Creating a thread in C++

Since C++11, the standard library provides `std::thread`.

(`std::thread` : class that represents a thread of execution and allows launching a function in a separate thread; defined in `<thread>`.)

Simple example:

```
#include <iostream>
#include <thread>

void task() {
    std::cout << "Hello from a thread" << std::endl;
}

int main() {
    std::thread t(task); // create the thread
    t.join();           // wait for the thread to finish
    return 0;
}
```

Key points:

- the thread starts **immediately** upon its creation,
- `join()` blocks the main thread until the end of thread `t`,
- `detach()` detaches the thread from the calling thread: it becomes independent and is no longer joinable,
- not calling `join()` or `detach()` before the destruction of a `std::thread` object causes `std::terminate()` to be called at runtime.

In this example:

- `task()` runs in a **separate thread**,
- the main thread waits for the end of `t` thanks to `join()`.

5.3 Example of parallel execution

Now consider two threads executing a time-visible task.

```

#include <iostream>
#include <thread>
#include <chrono>

void task(int id) {
    for(int i = 0; i < 5; ++i) {
        std::cout << "Thread " << id << " : step " << i << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

int main() {
    std::thread t1(task, 1);
    std::thread t2(task, 2);

    t1.join();
    t2.join();

    return 0;
}

```

(Reminder: `std::chrono` (in `<chrono>`) provides types for durations and clocks, e.g. `milliseconds`.)
 Typical output (the exact order may vary) :

```

Thread 1: step 0
Thread 2: step 0
Thread 1: step 1
Thread 2: step 1
Thread 2: step 2
Thread 1: step 2
Thread 1: step 3
Thread 2: step 3
Thread 2: step 4
Thread 1: step 4

```

What we observe:

- both threads **progress simultaneously**,
- their outputs are **interleaved**,
- the order is **not deterministic**.

5.4 Argument passing to threads

Arguments are copied by default.

```

void print(int x) {
    std::cout << x << std::endl;
}

std::thread t(print, 42);
t.join();

```

Following the generic argument-passing format.

```
std::thread t(fonction, arg1, arg2, arg3, ...);
```

To pass a reference:

```

#include <functional>

void increment(int& x) {
    x++;
}

int main() {
    int a = 5;
    std::thread t(increment, std::ref(a));
}

```

```
t.join();
}
```

5.5 Multiple threads and real parallelism

Example with multiple threads:

```
#include <thread>
#include <vector>

void work(int id) {
    // independent calculation
}

int main() {
    std::vector<std::thread> threads;

    for(int i = 0; i < 4; ++i)
        threads.emplace_back(work, i);

    for(auto& t : threads)
        t.join();
}
```

Each thread can be executed on a different core.

5.6 Shared memory

Threads share memory, which introduces **major risks**:

- race conditions (*race conditions*),
- data inconsistencies,
- nondeterministic behaviors.

Dangerous example:

```
int counter = 0;

void increment() {
    counter++; // non-atomic
}
```

If several threads execute `increment()`, the result is unpredictable.

5.7 Synchronization and critical sections

A **critical section** is a region of code that must be executed by only **one thread at a time**.

In C++, we use `std::mutex`.

(`std::mutex`: a mutex (lock) defined in `<mutex>` used to protect a critical section.)

```
#include <mutex>

int counter = 0;
std::mutex m;

void increment() {
    std::lock_guard<std::mutex> lock(m);
    counter++;
}
```

- the mutex prevents concurrent access,
- `lock_guard` guarantees automatic unlock.

5.8 Atomic variables

For simple operations, one can use `std::atomic`.

```
#include <atomic>

std::atomic<int> counter(0);

void increment() {
    counter++;
}
```

Advantages:

- faster than a mutex,
- safe for elementary operations.

Limit:

- unsuitable for complex structures.

Cost and limits of multithreading

Creating threads has a cost:

- creation,
- synchronization,
- memory contention.

Too many threads can:

- degrade performance,
- increase latency,
- complicate reasoning.

Good practice:

- use a number of threads close to the number of cores,
- favor coarser tasks rather than very fine-grained ones.

6 Generic programming, templates

Generic programming allows writing type-independent code, while preserving the performance of compiled C++. In C++, this paradigm relies mainly on templates, which enable defining functions and classes parameterized by types (or values). Templates are ubiquitous in the standard library (STL) and constitute a fundamental tool for writing reusable, expressive, and efficient code.

6.1 General principle of templates

A **template** is a code pattern that is not directly compiled. The compiler automatically generates a specialized version of the code for each type used.

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

The keyword `typename` (or alternatively `class` in this context) introduces a **type parameter** in the declaration `template <typename T>`.

Usage :

```
int a = add(2, 3);           // T = int
float b = add(1.5f, 2.5f); // T = float
```

For each type (`int`, `float`), the compiler generates a different function, with the same performance as hand-written code.

Function templates

Function templates allow writing generic algorithms without duplicating code.

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}
```

This function works for any type that supports the `>` operator:

```
maximum(3, 5);           // int
maximum(2.0f, 1.5f);    // float
```

If the type does not support the required operator, the error is detected at compile time.

Class templates

Templates can also be used to define generic classes.

```
template <typename T>
struct Box {
    T value;

    explicit Box(T v) : value(v) {}
};
```

Usage :

```
Box<int> a(3);
Box<float> b(2.5f);
```

Here, `Box<int>` and `Box<float>` are two distinct types generated by the compiler.

Examples for vectors

In computer graphics, templates are very widely used for:

- vectors and matrices of varying dimensions or types,
- typed CPU/GPU buffers,
- algorithms independent of precision (float, double).

Example of a generic vector :

```
template <typename T>
struct vec3 {
    T x, y, z;

    vec3(T x_, T y_, T z_) : x(x_), y(y_), z(z_) {}

    T norm2() const {
        return x*x + y*y + z*z;
    }
};
```

Usage :

```
vec3<float> vf(1.0f, 2.0f, 3.0f);
vec3<double> vd(1.0, 2.0, 3.0);
```

Non-type template parameters

A template can also take **non-type parameters**, known at compile time.

```
template <typename T, int N>
struct Array {
    T data[N];

    T& operator[](int i) { return data[i]; }
    T const& operator[](int i) const { return data[i]; }
};
```

Usage :

```
Array<float, 3> v; // size known at compile time
```

This principle is used in `std::array<T, N>`.

Template specialization

It is possible to provide a specific implementation for a given type.

```
template <typename T>
struct Printer {
    static void print(T const& v) {
        std::cout << v << std::endl;
    }
};

// spécialisation pour bool
template <>
struct Printer<bool> {
    static void print(bool v) {
        std::cout << (v ? "true" : "false") << std::endl;
    }
};
```

Specialization allows adapting the behavior without modifying the generic code.

6.2 Compilation Principles: duck typing, instantiation, and header files

Compilation of templates in C++ follows specific rules, different from those of ordinary code. Understanding these principles is essential for interpreting compiler error messages and organizing code properly.

Static duck typing

The templates rely on a principle called **static duck typing**.

The principle is as follows:

A type is valid if it provides all the operations used in the template.

For example:

```
template <typename T>
T square(T x) {
    return x * x;
}
```

This template imposes **no explicit constraint** on T . However, during instantiation, the compiler requires that the type used possesses the `*` operator.

```
square(3);           // OK: int supports *
square(2.5f);       // OK: float supports *
```

On the other hand:

```
struct A {};
square(A{}); // ERREUR de compilation
```

The error occurs **at the moment the template is instantiated**, and not at its definition. This is a key feature of templates:

- the generic code can be syntactically correct,
- but invalid for some concrete types.

This mechanism explains why template-related errors can be long and complex: the compiler tries to instantiate the code with a given type and fails when a required operation does not exist.

Template instantiation

A template is **not compiled until it is used**. Actual compilation occurs during **instantiation**, i.e., when the compiler encounters a concrete usage:

```
add<int>(2, 3);
add<float>(1.5f, 2.5f);
```

Each instantiation generates:

- a different function for each type,
- or a different type for each combination of template parameters.

Thus:

```
Box<int>
Box<float>
```

are two **distinct types**, with no inheritance relationship between them.

Important consequence: code visible at compile time

For the compiler to instantiate a template, it must have access to the **complete implementation** of the template at compile time.

This has a major consequence for how files are organized.

Templates and header files (.hpp)

Unlike regular functions and classes, **the body of templates must be visible everywhere they are used**. That's why:

- templates are defined in **header files** (.hpp),
- they are generally **not separated** into .hpp / .cpp.

Correct example:

```
// vec.hpp
#pragma once

template <typename T>
T add(T a, T b) {
    return a + b;
}
```

```
// main.cpp
#include "vec.hpp"

int main() {
    int a = add(2, 3);
}
```

If the body of the template were placed in a .cpp, the compiler could not generate the specialized versions, because the implementation would not be visible at instantiation time.

Why templates cannot be compiled separately

In standard code:

- the compiler produces an object file (.o) from a .cpp,
- the linker then assembles the symbols.

With templates:

- the generated code depends on the **types used**,
- these types are known only at the point of use.

The compiler cannot therefore pre-produce a single generic version of the template. It must see **both**:

- the definition of the template,
- and the concrete type used.

Exceptions and special cases

There are advanced techniques (explicit instantiation) that allow partially separating the implementation, but they remain complex; in practice, the simple rule is:

All templates must be fully defined in a header file.

Summary of key principles

- Templates use a **static duck typing**: constraints on types are implicit.
- Errors are detected **at instantiation**, not at definition.
- Each combination of template parameters generates a specific code.
- The compiler must see **the complete implementation** of the template.
- Templates are therefore defined in `.hpp` files, not `.cpp`.

These rules explain both the **power** and the **complexity** of templates in C++.

6.3 Static metaprogramming

Static metaprogramming refers to the set of techniques that allow performing **calculations at compile time**, even before the program runs. In C++, templates and the `constexpr` expressions enable moving part of the program's logic to the compiler. The result is code **faster at runtime**, since some decisions and calculations are already resolved.

General principle

The central idea is the following:

use the compiler as a **calculation engine**.

The values produced by metaprogramming:

- are known at compile time,
- cost no computation time at runtime,
- can be used as **template parameters**, array sizes, or constants.

Metaprogramming with integer template parameters

Non-type template parameters (integers) are the first tool of metaprogramming.

```
template <int N>
int static_square()
{
    return N * N;
}
```

Usage:

```
int main()
{
    const int a = static_square<5>();    // evaluated at compile time

    float buffer[static_square<3>()];   // taille connue statiquement

    std::cout << a << std::endl;
    std::cout << sizeof(buffer) / sizeof(float) << std::endl;
}
```

Here:

- `static_square<5>()` is calculated by the compiler,
- no multiplication is executed at run time.

constexpr : computations evaluated by the compiler

Since C++11, the keyword `constexpr` allows explicitly requesting a **compile-time evaluation**, if the arguments are constant.

```
constexpr int square(int N)
{
    return N * N;
}
```

The compiler:

- checks that the expression can be evaluated statically,
- generates a constant if that's the case.

Comparison with a classic function:

```
int runtime_square(int N)
{
    return N * N;
}
```

Usage in a template parameter:

```
template <int N>
void print_value()
{
    std::cout << N << std::endl;
}

int main()
{
    print_value<square(5)>(); // OK: constant expression
    // print_value<runtime_square(5)>(); // ERROR: not constant
}
```

Recursive calculations at compile time

Templates and `constexpr` allow writing **recursive** calculations evaluated at compile time.

Example: factorial calculation.

```
constexpr int factorial(int N)
{
    return (N <= 1) ? 1 : N * factorial(N - 1);
}
```

Use as a template parameter:

```
template <typename T, int N>
struct vecN
{
    T data[N];
};

int main()
{
    vecN<float, factorial(4)> v;

    for (int k = 0; k < factorial(4); ++k)
        v.data[k] = static_cast<float>(k);
}
```

The calculation of $4!$ is performed **entirely at compile time**.

Template metaprogramming (historical form)

Before `constexpr`, metaprogramming relied exclusively on **recursive templates**.

```
template <int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static constexpr int value = 1;
};
```

Usage :

```
int size = Factorial<5>::value; // evaluated at the compilation
```

This technique is more complex and less readable, but it is historically important and still present in some libraries.

Typical use cases

Static metaprogramming is used for:

- sizes of arrays known at compile time,
- algorithms specialized according to constant parameters,
- conditional code selection (`if constexpr` in C++17),
- aggressive optimization with no runtime cost,
- generic mathematical structures (vectors, matrices).

Example with `if constexpr` :

```
template <typename T>
void process(T v)
{
    if constexpr (std::is_integral_v<T>)
        std::cout << "Integer" << std::endl;
    else
        std::cout << "Non-integer" << std::endl;
}
```

Note: `std::is_integral_v` is provided by the header `<type_traits>`.

The non-relevant branch is **removed at compile time**.

Limitations and precautions

- It increases the **compilation time**.
- Can make errors harder to understand.
- Code can become less readable if metaprogramming is excessive.

6.4 Type deduction in templates

One of the major goals of generic programming is to make code **both generic and readable**. In C++, the compiler is able to **automatically deduce template parameters** in many cases, from the arguments provided at the call. Understanding when this deduction works — and when it fails — is essential to write effective generic interfaces.

General principle of deduction

When a template is used **without explicitly specifying its parameters**, the compiler tries to deduce them from the argument types.

```
template <typename T>
T add(T a, T b)
{
    return a + b;
}
```

Usage:

```
int a = add(2, 3); // T deduced as int
float b = add(1.2f, 3.4f); // T deduced as float
```

Here, the compiler deduces `T` automatically from the arguments passed to the function.

Limitations of automatic deduction

Type deduction works **only** from the **function parameters**. It does not work from the return type.

```
template <typename T>
T identity();
```

This template **cannot be called** without specifying `T`, because the compiler has no information to deduce it.

```
// identity(); // ERROR
identity<int>(); // OK
```

Problematic example: generic dot product

Consider a generic dot product function:

```
template <typename TYPE_INPUT, typename TYPE_OUTPUT, int SIZE>
TYPE_OUTPUT dot(TYPE_INPUT const& a, TYPE_INPUT const& b)
{
    TYPE_OUTPUT val = 0;
    for (int k = 0; k < SIZE; ++k)
        val += a[k] * b[k];
    return val;
}
```

Usage:

```
vecN<float,3> v0, v1;

// Heavy and hard-to-read call
float p = dot<vecN<float,3>, float, 3>(v0, v1);
```

In this case:

- `TYPE_INPUT`, `TYPE_OUTPUT` and `SIZE` **cannot be deduced automatically**,
- the call becomes verbose and hard to read.

Why deduction fails here

Deduction fails because:

- `TYPE_OUTPUT` appears only in the **return type**,
- `SIZE` appears only as a **template parameter**, not in the function arguments.

The compiler can deduce a template parameter only if it is **directly tied to the argument types**.

Exposing template parameters in the types

One solution is to explicitly expose the template parameters in the generic class.

```
template <typename TYPE, int SIZE>
class vecN
{
public:
    using value_type = TYPE;
    static constexpr int size() { return SIZE; }

    TYPE& operator[](int index);
    TYPE const& operator[](int index) const;

private:
    TYPE data[SIZE];
};
```

We can then write a much more readable function:

```

template <typename V>
typename V::value_type dot(V const& a, V const& b)
{
    typename V::value_type val = 0;
    for (int k = 0; k < V::size(); ++k)
        val += a[k] * b[k];
    return val;
}

```

Usage :

```
float p = dot(v0, v1); // types et taille déduits automatiquement
```

Here :

- v is deduced as `vecN<float,3>`,
- the return type is extracted via `V::value_type`,
- the size is known at compile-time via `V::size()`.

Access to internal types: `typename`

When a type depends on a template parameter, it must be preceded by `typename` to indicate to the compiler that it is indeed a type.

```
typename V::value_type
```

Without `typename`, the compiler cannot tell whether `value_type` is a type or a static value.

Partial deduction and default parameters

Templates can also use **default parameters** to reduce verbosity:

```

template <typename T, int N = 3>
struct vecN;

```

This mechanism helps simplify certain usages, but does not replace good interface design.

Deduction with `auto` and C++17+

Since C++17, `auto` can be used to deduce the return type of a template function :

```

template <typename V>
auto norm2(V const& v)
{
    auto val = typename V::value_type{};
    for (int k = 0; k < V::size(); ++k)
        val += v[k] * v[k];
    return val;
}

```

This improves readability while preserving genericity.

6.5 Template specialization

Template specialization allows adapting the behavior of a generic template to a **particular case**, without changing the general implementation. It is used when, for a given type or parameter, the default behavior is not suitable, inefficient, or incorrect.

Specialization is a compile-time resolved mechanism, and an integral part of generic programming in C++.

General principle

We start by defining a **generic template** (the general case), then provide a **specialized** implementation for a given type or value.

```
template <typename T>
struct Printer
{
    static void print(T const& v)
    {
        std::cout << v << std::endl;
    }
};
```

This template works for any type compatible with operator<<.

Full specialization of a template

A **full specialization** completely replaces the template implementation for a specific type.

```
template <>
struct Printer<bool>
{
    static void print(bool v)
    {
        std::cout << (v ? "true" : "false") << std::endl;
    }
};
```

Usage :

```
Printer<int>::print(5); // uses the generic version
Printer<bool>::print(true); // uses the specialization
```

The compiler automatically selects the most specific version available.

Specialization of function templates

Function templates can also be specialized, but their use is more delicate.

```
template <typename T>
void display(T v)
{
    std::cout << v << std::endl;
}

template <>
void display<bool>(bool v)
{
    std::cout << (v ? "true" : "false") << std::endl;
}
```

Here as well, the specialized version is used when $\tau = \text{bool}$.

Partial Specialization (class templates)

The **Partial specialization** allows specializing a template for a **family of types**, but it is allowed **only for class templates**, not for functions.

Example: specialization according to an integer parameter.

```
template <typename T, int N>
struct Array
{
    T data[N];
};
```

Partial specialization for $N = 0$:

```

template <typename T>
struct Array<T, 0>
{
    // empty array
};

```

Here, all types `Array<T,0>` use this specific version.

Partial Specialization with pointer types

Another classic example :

```

template <typename T>
struct is_pointer
{
    static constexpr bool value = false;
};

template <typename T>
struct is_pointer<T*>
{
    static constexpr bool value = true;
};

```

Usage :

```

is_pointer<int>::value;    // false
is_pointer<int*>::value;  // true

```

This type of specialization is widely used in the STL (`std::is_pointer`, `std::is_integral`, etc.).

Full (or complete) Specialization

The **Full specialization** consists of providing a specific implementation for **an entirely fixed combination of template parameters** (types and/or values). For this exact combination, the generic template **is not used at all**: the specialization replaces it entirely.

In the context of **generic vectors**, this allows for example:

- to optimize a particular case (current dimension),
- to define a different behavior for a given size,
- or to adapt an internal representation.

Example: fixed-size generic vector

We first define a generic template for a vector of arbitrary size known at compile time.

```

template <typename T, int N>
struct vec
{
    T data[N];

    T& operator[](int i) { return data[i]; }
    T const& operator[](int i) const { return data[i]; }
};

```

This template works for **any type** `T` and **any size** `N`.

Full Specialization for a 2D vector

Suppose we want a special treatment for 2D vectors, for example:

- direct access via `x` and `y`,
- more readable code,
- possibly more optimizable.

We then define a **full specialization** :

```
template <typename T>
struct vec<T, 2>
{
    T x, y;

    vec() : x(0), y(0) {}
    vec(T x_, T y_) : x(x_), y(y_) {}

    T& operator[](int i)
    {
        return (i == 0) ? x : y;
    }

    T const& operator[](int i) const
    {
        return (i == 0) ? x : y;
    }
};
```

Here:

- `vec<T,2>` is a **completely different type** from `vec<T,N>`,
- the array `data[N]` no longer exists,
- the behavior is completely redefined for `N = 2`.

Usage

```
vec<float, 3> v3;
v3[0] = 1.0f;
v3[1] = 2.0f;
v3[2] = 3.0f;

vec<float, 2> v2(1.0f, 4.0f);
std::cout << v2[0] << " " << v2[1] << std::endl;
```

- `vec<float,3>` uses the **generic template**,
- `vec<float,2>` uses the **total specialization**.

The choice is made **at compile time**, with no runtime test.

Total specialization for a specific type and size

It is also possible to specialize for a **specific type and size**.

```
template <>
struct vec<float, 3>
{
    float x, y, z;

    vec() : x(0.f), y(0.f), z(0.f) {}
    vec(float x_, float y_, float z_) : x(x_), y(y_), z(z_) {}

    float norm2() const
    {
        return x*x + y*y + z*z;
    }
};
```

Usage:

```
vec<float,3> v(1.f, 2.f, 3.f);
std::cout << v.norm2() << std::endl;
```

Here:

- this version is used **only** for `vec<float,3>`,
- all other combinations (`vec<double,3>`, `vec<float,4>`, etc.) use the generic template.

Comparison with partial specialization

- **Total specialization** All the template parameters are fixed (`vec<float,3>`). → a single case, completely redefined behavior.
- **Partial specialization** Only a part of the parameters is fixed (`vec<T,2>`). → a family of types sharing a specific behavior.

6.6 Priority between specialization and overloading

It is common to confuse **overloading** and **template specialization**, but they are two distinct mechanisms that come into play **at different times** in compilation. Understanding their **order of priority** is essential to avoid surprising behaviors.

The key idea is the following:

Overloading is resolved before template specialization.

In other words, the compiler first chooses **which function to call**, and only then **which version of the template to instantiate**.

Step 1: overloading resolution

When several functions bear the same name, the compiler starts by applying the classical overloading rules:

- exact type matching,
- implicit conversions,
- templates vs non-template functions.

Example:

```
void display(int x)
{
    std::cout << "normal int function\n";
}

template <typename T>
void display(T x)
{
    std::cout << "generic template\n";
}
```

Call:

```
display(3);
```

Result:

```
normal int function
```

A non-template function is always prioritized over a function template if it matches exactly.

Step 2: template selection

If no non-template function matches, the compiler considers the **template functions** and attempts to deduce the parameters.

```
template <typename T>
void display(T x)
{
    std::cout << "generic template\n";
}
```

```
}  
display(3.5); // T = double
```

Here, the template is selected because no classic function matches.

Step 3: template specialization

Once a **template has been chosen**, the compiler searches whether there exists a **more specific specialization** for the deduced parameters.

```
template <typename T>  
void display(T x)  
{  
    std::cout << "generic template\n";  
}  
  
template <>  
void display<bool>(bool x)  
{  
    std::cout << "bool specialization\n";  
}
```

Calls:

```
display(5); // generic template  
display(true); // bool specialization
```

Result:

```
generic template  
bool specialization
```

The specialization **does not participate in overloading**. It is selected **after** the generic template has been chosen.

Subtle case: specialization vs overloading

Consider now:

```
template <typename T>  
void display(T x)  
{  
    std::cout << "generic template\n";  
}  
  
template <>  
void display<int>(int x)  
{  
    std::cout << "int specialization\n";  
}  
  
void display(int x)  
{  
    std::cout << "normal int function\n";  
}
```

Call:

```
display(3);
```

Result:

```
normal int function
```

Explanation:

1. the compiler sees a non-template function `display(int)` → **takes priority**,
2. the template is not even considered,
3. the template specialization is ignored.

A specialization can never beat a non-template overload.

Why this behavior?

Because:

- overloading is a decision **syntactic and local**,
- specialization is a decision **internal to the template**,
- mixing the two levels would render compilation ambiguous.

C++ thus imposes a strict hierarchy.

Priority summary (exact order)

During a function call:

1. Selection of candidate functions (name, scope).
2. Overload resolution:
 - non-template functions,
 - then template functions.
3. If a template is chosen:
 - selection of the most specific specialization.
4. Instantiation of the corresponding code.

Practical rule to remember

Overloading chooses the function. Specialization chooses the template implementation.

Best practices

- Use the **overloading** to provide different interfaces.
- Use **specialization** to tailor internal behavior for a template.
- Avoid mixing overloading and specialization on the same name without a clear reason.

6.7 Aliases

Type aliases in templates (`typedef` and `using`)

Type aliases allow giving a **more readable** or more **expressive** name to a type, often complex. They play a central role in generic programming, as they facilitate **type deduction**, the **writing of generic functions**, and the **readability of interfaces**.

In C++, two equivalent mechanisms exist:

- `typedef` (historical),
- `using` (modern, recommended).

Alias with `typedef` (historical form)

```
typedef unsigned int uint;
```

This mechanism works, but quickly becomes hard to read with complex types, especially in the presence of templates.

Alias with using (modern form)

Since C++11, we prefer to use `using`, clearer and more powerful.

```
using uint = unsigned int;
```

This syntax is equivalent to `typedef`, but much more readable, especially with templates.

Alias inside a template class

Aliases are very often used **inside template classes** to expose their internal parameters.

Example with a generic vector:

```
template <typename T, int N>
class vec
{
public:
    using value_type = T;
    static constexpr int size() { return N; }

    T& operator[](int i) { return data[i]; }
    T const& operator[](int i) const { return data[i]; }

private:
    T data[N];
};
```

Here:

- `vec<T,N>::value_type` gives access to the stored type,
- `vec<T,N>::size()` gives access to the size known at compile time.

These aliases make the class **auto-descriptive** and facilitate its use in generic code.

Using aliases in template functions

Thanks to aliases, one can write generic functions **without explicitly knowing the template parameters**.

```
template <typename V>
typename V::value_type sum(V const& v)
{
    typename V::value_type s = 0;
    for (int i = 0; i < V::size(); ++i)
        s += v[i];
    return s;
}
```

Usage:

```
vec<float,3> v;
v[0] = 1.0f; v[1] = 2.0f; v[2] = 3.0f;

float s = sum(v);
```

Here:

- the return type is automatically deduced via `value_type`,
- the function works for **any type of vector compatible**.

Dependent aliases (typename)

When accessing a dependent alias from a template parameter, it is necessary to use the keyword `typename` to indicate that it is indeed a **type**.

```
typename V::value_type
```

Without `typename`, the compiler cannot tell whether `value_type` is a type or a static value.

Alias templates (templated aliases)

Aliases themselves can be **templates**, which allows simplifying very complex types.

```
template <typename T>
using vec3 = vec<T, 3>;
```

Usage:

```
vec3<float> a;
vec3<double> b;
```

Here:

- `vec3<float>` is equivalent to `vec<float,3>`,
- the alias greatly improves readability.

Aliases and coherence of generic interfaces

Aliases are widely used in the STL:

- `value_type`,
- `iterator`,
- `reference`,
- `const_reference`.

Respecting these conventions helps to make your classes **compatible with generic algorithms**.

Example:

```
template <typename Container>
void print_container(Container const& c)
{
    for (typename Container::value_type const& v : c)
        std::cout << v << " ";
}
```

7 Development methodologies and best practices

This chapter presents the **fundamental methodological principles** for producing C++ code:

- readable,
- robust,
- testable,
- maintainable,

all while respecting the performance and low-level constraints inherent to the language.

These principles apply just as well to small programs as to complex projects (simulation, graphics engine, parallel computation).

7.1 Code quality: concrete objectives

Code quality is not measured by perceived elegance, but by practical criteria:

- **Readability**: the code is understandable without excessive effort.
- **Locality**: understanding a function does not require exploring the entire project.
- **Robustness**: errors are detected and handled explicitly.
- **Testability**: the code can be automatically validated.
- **Scalability**: future changes are possible without massive rewrites.
- **Controlled performance**: optimization guided by measurements, not by intuition.

Note that when working with others, code readability should be the priority. Readable code:

- facilitates reading and code reviews,
- reduces errors during modifications,
- speeds up the onboarding of new contributors,
- makes reasoning and testing easier.

In most cases, one should favor readability and simplicity over premature micro-optimizations. Efficiency can be pursued later, in a targeted and measured way, when a performance bottleneck is proven.

Best practices for readability: explicit names, short functions, comments when the code is not self-documenting, consistent formatting, and systematic code reviews.

7.2 General principles: KISS, DRY, YAGNI

KISS -- *Keep It Simple, Stupid*

Simple code is more reliable than complex code.

- Prefer a direct implementation over premature abstraction.
- Avoid "clever" constructions that are hard to explain.
- A function should ideally fit on one screen.

Example (KISS) :

```
// Condensed and less readable version: nested logic, index calculation
// difficult to follow, everything is condensed into a few lines.
int count_neighbors_ugly(const std::vector<int>& grid, size_t w, size_t h,
                       size_t x, size_t y)
{
    int c = 0;
    // sweep a 3x3 rectangle centered on (x,y) by adjusting the bounds
    size_t start = (y ? y - 1 : 0) * w + (x ? x - 1 : 0);
    size_t end_y = (y + 1 < h ? y + 1 : h - 1);
```

```

size_t end_x = (x + 1 < w ? x + 1 : w - 1);
for (size_t idx = start; ++idx) {
    size_t cx = idx % w;
    size_t cy = idx / w;
    if (!(cx == x && cy == y)) c += grid[idx];
    if (cy == end_y && cx == end_x) break; // subtle logic
}
return c;
}

// Clear and simple version: helper functions and explicit loops

inline bool in_bounds(size_t x, size_t y, size_t w, size_t h) { return x < w && y < h; }
inline int at(const std::vector<int>& g, size_t w, size_t x, size_t y) { return g[y * w + x]; }

int count_neighbors(const std::vector<int>& grid, size_t w, size_t h,
                  size_t x, size_t y)
{
    int c = 0;
    size_t y0 = (y > 0) ? y - 1 : 0;
    size_t y1 = (y + 1 < h) ? y + 1 : h - 1;
    size_t x0 = (x > 0) ? x - 1 : 0;
    size_t x1 = (x + 1 < w) ? x + 1 : w - 1;

    for (size_t yy = y0; yy <= y1; ++yy) {
        for (size_t xx = x0; xx <= x1; ++xx) {
            if (xx == x && yy == y) continue; // ignore the central cell
            c += at(grid, w, xx, yy);
        }
    }
    return c;
}

```

DRY -- *Don't Repeat Yourself*

One piece of logic should exist in only one place.

Attention:

eliminating any duplication can lead to unnecessary abstractions.

A local and simple duplication is sometimes preferable to a complex generalization.

Example (DRY) :

```

// Duplication (moins bon) : deux fonctions très similaires
double average_int(const std::vector<int>& v) {
    if (v.empty()) return 0.0;
    long sum = 0;
    for (int x : v) sum += x;
    return double(sum) / v.size();
}

double average_double(const std::vector<double>& v) {
    if (v.empty()) return 0.0;
    double sum = 0;
    for (double x : v) sum += x;
    return sum / v.size();
}

// Refactorisation (DRY) : une implémentation générique évite la duplication
template<typename T>
double average(const std::vector<T>& v) {
    if (v.empty()) return 0.0;
    long double sum = 0;
    for (T x : v) sum += x;
    return double(sum) / v.size();
}

// Usage :
// std::vector<int> vi = {1,2,3};
// std::vector<double> vd = {1.0,2.0,3.0};
// double a1 = average(vi); // fonctionne pour int
// double a2 = average(vd); // fonctionne pour double

```

YAGNI -- *You Aren't Gonna Need It*

Do not implement features "just in case" if they are not needed.

This principle is especially important in C++, where: - templates, - generics, - and metaprogramming can encourage excessive complexity too early.

Example (YAGNI) :

```
// Prematurely generalized (YAGNI)
template <typename T = float, int N = 3>
struct vec { T data[N]; };

// Version simple et suffisante pour l'usage courant
struct vec3 { float x, y, z; };
```

7.3 Invariants, assertions et contrat de fonction

A robust program does not just "work in normal cases": it **explicitly states its assumptions** and verifies that they are satisfied.

These assumptions constitute what is called the **contract** of the code.

Why talk about a contract ?

When a function is called, two viewpoints exist :

- **the caller's viewpoint** "What am I allowed to pass to this function ?"
- **the function's viewpoint** "What do I guarantee in return ?"

If these rules are implicit or only "in the developer's head", the code becomes fragile:

- silent errors,
- indeterminate behaviors,
- bugs difficult to diagnose.

The contract allows formalizing these rules. The set of these rules is what we call design by contract.

The three key notions of the contract

We distinguish three complementary types of rules.

1. Preconditions

A **precondition** is a condition that **must be true before** calling a function.

- It describes what the function **expects**.
- It is the caller's responsibility.

Examples:

- an index must be valid,
- a pointer must not be null,
- a divisor must be non-zero.

2. Postconditions

A **postcondition** is a condition that **must be true after the function executes**.

- It describes what the function **guarantees**.
- It is the function's responsibility.

Examples:

- the size of a container has increased,
- a returned value lies within a range,
- an internal state has been updated correctly.

3. Invariants

An **invariant** is a property that must be **always true** for a valid object.

- It is established by the constructor.
- It must be preserved by **all public methods**.

Examples:

- $0 \leq \text{size} \leq \text{capacity}$,
- a radius is always strictly positive,
- two member pointers are either both valid or both null.

Conceptual illustration: stack

Before looking at C++, here is a conceptual view of a stack's contract.

```
Entity: Stack (Stack)

Invariant :
    0 <= size <= capacity

Constructor(capacity):
    establishes the invariant
    size := 0
    capacity := capacity

push(value):
    precondition: size < capacity
    postcondition: top == value, size increased by 1

pop():
    precondition: size > 0
    postcondition: size decreased by 1
```

The invariant must be true **after every public call**, regardless of the sequence of operations.

Assertions at runtime (assert)

Assertions allow to verify these rules **during execution**, mainly in the development phase.

In C++, we use `assert` to detect **programming errors**.

```
#include <cassert>

float safe_div(float a, float b)
{
    assert(b != 0.0f && "Division by zero");
    return a / b;
}
```

Here:

- `b != 0.0f` is a **precondition**,
- the assertion documents and checks this hypothesis.

What are asserts for?

Assertions allow you to:

- document the internal assumptions of the code,
- quickly detect logic errors,
- stop the program **at the exact point of the problem** in debug.

They are therefore a **development tool**, not a mechanism for handling user errors.

Best practices with assert

- use `assert` for **programming errors**. The asserts are theoretically 'useless' for the correct operation of the program; they are only there to help programming by detecting unexpected/unforeseen cases that should never happen.
- do not use `assert` for:
 - missing files,
 - invalid user input,
 - recoverable errors
- never write side effects:

```
assert(++i < 10); // forbidden
// Here the value of i is modified after the execution of the assert.
// When compiling in "release" mode, the assertion is not executed, and the value of i will be different in
// the program.
```

- provide an explicit message:

```
assert(ptr && "ptr must not be null");
```

Debug vs Release mode

- In **debug**: the asserts are active
- In **release**: they are removed (`NDEBUG`)

Note: The program should **never rely** on assertions to function correctly.

Compile-time Assertions (`static_assert`)

Some rules can be checked **before even running**, at compile time.

That's the role of `static_assert`.

```
#include <type_traits>

template <typename T>
T square(T x)
{
    static_assert(std::is_arithmetic_v<T>,
                  "square expects an arithmetic type");
    return x * x;
}
```

Here:

- the constraint is checked **at compile time**,
- a misuse prevents the generation of the executable.

When to use `static_assert`?

- sizes known at compile time,
- constraints on template types,
- structural hypotheses impossible to verify at runtime.

General rule: prefer compile-time checks when possible.

Complete example: stack with invariant and assertions

```
#include <cassert>
#include <vector>

struct Stack {
    std::vector<int> data;
    size_t capacity;

    // Invariant :
    // 0 <= data.size() <= capacity

    explicit Stack(size_t cap) : capacity(cap)
    {
        assert(capacity > 0 && "capacity must be positive");
    }

    void push(int v)
    {
        // precondition
        assert(data.size() < capacity && "push: stack full");

        data.push_back(v);

        // postcondition
        assert(data.back() == v && "push: top incorrect");
    }

    int pop()
    {
        // precondition
        assert(!data.empty() && "pop: stack empty");

        int v = data.back();
        data.pop_back();

        // invariant always valid
        assert(data.size() <= capacity && "invariant violated");

        return v;
    }
};
```

Summary

- A **contract** describes what the code expects and guarantees.
- The **preconditions** are the caller's responsibility.
- The **postconditions** are the function's responsibility.
- The **invariants** define the valid states of an object.
- `assert` checks the contract at runtime (debug).
- `static_assert` checks the contract at compile time.
- When used correctly, they make the code:
 - safer,
 - more readable,
 - and easier to maintain.

Alternatives to asserts

The `assert` function remains fairly limited in terms of functionality. Alternative tools can help express and verify contracts in a more readable, safer, and maintainable way for large-scale codebases:

- **GSL (Guideline Support Library)**: provides `Expects()` / `Ensures()` (macros or functions) to document pre/postconditions, as well as `not_null<T>` and `span<T>` for safe pointers and views.
- **Result types (expected/Outcome)**: use `tl::expected` / `Outcome` OR `std::expected` when available to explicitly represent recoverable errors instead of exceptions or magic codes.
- **Concepts & `static_assert` / `constexpr`**: move the checks to compile time when possible (templates, type constraints), reducing the need for runtime assertions.
- **Contract libraries**: `Boost.Contract` and other frameworks offer richer `require/ensure/invariant` annotations (activatable/deactivatable contracts, centralized diagnostics).
- **Lightweight annotations (Expects/Ensures)**: define wrappers `Expects(condition)` to standardize messages and enable different behaviors depending on configuration (throw, abort, log).
- **Complementary tools**: sanitizers (ASan/UBSan/TSan) and static analysis (clang-tidy, cppcheck) detect classes of errors that assertions alone do not cover.

7.4 Tests and Test-Driven Development (TDD)

A program may seem correct on a few simple examples and yet be wrong in edge cases or after a later modification. Tests allow automatic verification that the code respects its expected behavior, and especially that this behavior remains correct over time.

Testing is not about proving that the program is perfect, but about **reducing the risk of error** and detecting problems as early as possible.

Why write tests?

Tests are useful when they help to:

- detect an error before the end user,
- avoid regressions when making a change or refactoring,
- document the expected behavior of the code in an executable form,
- facilitate evolving the code with confidence.

In a real project, tests are often run automatically with every modification (continuous integration).

What makes a good test?

A good test is:

- **deterministic**: it always produces the same result under the same conditions,
- **fast**: it should be able to be run frequently,
- **isolated**: it does not depend on hidden global state,
- **clear**: one easily understands what is being tested and why,
- **localized**: in case of failure, the cause is quickly identifiable.

Large categories of tests

Unit tests

A **unit test** verifies a function or a class in isolation.

- without I/O,
- without network access,
- without hardware dependencies.

They are fast and very precise.

They are ideal for testing: - mathematical functions, - algorithms, - data structures.

Integration tests

An **integration test** verifies the interaction between multiple components:

- reading files,
- loading resources,
- threads,
- communication between modules.

They are slower but closer to real-world behavior.

Regression tests

A **regression test** is added after fixing a bug.

- it reproduces a case that has already failed,
- it guarantees that this bug will not reappear.

These tests are extremely valuable in the long term.

Structure of a test : Arrange / Act / Assert

A readable test generally follows the following structure :

1. **Arrange** : preparation of data,
2. **Act** : call to the code under test,
3. **Assert** : verification of the result.

Example :

```
// Arrange
float x = -1.0f;

// Act
float y = clamp(x, 0.0f, 1.0f);

// Assert
assert(y == 0.0f);
```

This structure improves readability and maintainability of tests.

Which cases should be tested ?

For a given function, it is recommended to test :

1. the **nominal case** (normal usage),
2. the **edge cases** (bounds, sizes 0 or 1, extreme values),
3. the **error cases** (violated preconditions, invalid inputs).

Testing only the nominal case is rarely sufficient.

Minimalist test tool (no framework)

You can write tests with `assert`, but it is often useful to have more explicit messages, especially for floating-point numbers.

```
#include <iostream>
#include <cmath>
#include <cstdlib>

inline void check(bool cond, const char* msg)
{
    if (!cond) {
        std::cerr << "[TEST FAILED] " << msg << std::endl;
    }
}
```

```

    std::exit(1);
}
}

inline void check_near(float a, float b, float eps, const char* msg)
{
    if (std::abs(a - b) > eps) {
        std::cerr << "[TEST FAILED] " << msg
            << " (a=" << a << ", b=" << b << ")" << std::endl;
        std::exit(1);
    }
}
}

```

7.5 Guided example: unit tests for clamp

Expected specification

The function `clamp(x, a, b)` :

- returns `a` if `x < a`,
- returns `b` if `x > b`,
- returns `x` otherwise.

Precondition: `a <= b`.

Tests

```

#include <cassert>

float clamp(float x, float a, float b);

int main()
{
    // nominal case
    assert(clamp(0.5f, 0.0f, 1.0f) == 0.5f);

    // edge cases
    assert(clamp(0.0f, 0.0f, 1.0f) == 0.0f);
    assert(clamp(1.0f, 0.0f, 1.0f) == 1.0f);

    // saturation
    assert(clamp(-1.0f, 0.0f, 1.0f) == 0.0f);
    assert(clamp( 2.0f, 0.0f, 1.0f) == 1.0f);

    // precondition violation (should fail in debug)
    // clamp(0.0f, 1.0f, 0.0f);
}

```

Implementation :

```

#include <cassert>

float clamp(float x, float a, float b)
{
    assert(a <= b && "clamp: intervalle invalide");
    if (x < a) return a;
    if (x > b) return b;
    return x;
}

```

The precondition here is part of the **contract**: its violation is a programming error.

7.6 Test-Driven Development (TDD)

The **TDD** is a methodology in which code is written **in response to tests**. It aims to translate the functional requirement into verifiable behavior.

TDD Loop: Red -> Green -> Refactor

1. **Red**: write a test that fails,
2. **Green**: write the minimal code to get the test to pass,
3. **Refactor**: improve the code without breaking the tests.

This loop is repeated frequently.

Benefits of TDD

TDD:

- forces clarification of the API from the outset,
- encourages short, testable functions,
- limits over-engineering (YAGNI),
- makes refactorings much safer.

7.7 TDD Example: normalization of a 3D vector

Specification

- if v is non-zero, `normalize(v)` returns a vector of norm 1,
- the direction is preserved,
- precondition: $\text{norm}(v) > 0$.

Step 1: Test (Red)

```
#include <cassert>
#include <cmath>

struct vec3 { float x, y, z; };

float norm(vec3 const& v)
{
    return std::sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}

vec3 normalize(vec3 const& v);

int main()
{
    vec3 v{3.0f, 0.0f, 4.0f};
    vec3 u = normalize(v);

    assert(std::abs(norm(u) - 1.0f) < 1e-6f);

    float dot = v.x*u.x + v.y*u.y + v.z*u.z;
    assert(dot > 0.0f);
}
```

Step 2: Minimal Implementation (Green)

```
#include <cassert>
#include <cmath>

vec3 normalize(vec3 const& v)
```

```

{
    float n = norm(v);
    assert(n > 0.0f && "normalize: vecteur nul");
    return {v.x / n, v.y / n, v.z / n};
}

```

Step 3: Refactor (Refactor)

Then, one can:

- factorize `norm2`,
- improve performance,
- add non-regression tests.

Conclusion on tests and TDD

Tests provide an automatic verification of a function's contract. TDD offers a simple methodology for writing code:
define the behavior -> verify it automatically -> improve the implementation with confidence.

When used properly, tests make code more reliable, more readable, and easier to evolve.

Testing invalid cases

Testing only valid cases is not enough: robust code must also **properly detect invalid usages**. It is therefore essential to write tests that verify that:

- violated preconditions are properly detected (assertion, exception, error returned),
- invalid inputs do not produce silently incorrect results,
- the program fails in a controlled and explicit way, rather than continuing in an incoherent state.

These negative tests help ensure that the code contract is really respected, and not only in ideal cases. They are particularly important during refactorings: an internal change must never transform a detected error into silent behavior.

According to the chosen error handling policy, a test can verify:

- that an assertion fails in debug mode,
- that an exception is thrown,
- or that a result type explicitly signals an error.

In practice, **testing invalid cases is often as important as testing valid cases**, because it is precisely in these situations that the most costly bugs appear.

Example: test an invalid case detected by assert

We take back the function `normalize(v)` seen earlier. Its **precondition** is that the vector is not null.

```

vec3 normalize(vec3 const& v)
{
    float n = norm(v);
    assert(n > 0.0f && "normalize: vecteur nul");
    return {v.x / n, v.y / n, v.z / n};
}

```

It is important to verify that this precondition is **indeed detected**.

```

// Negative test: precondition violation (should fail in debug)
int main()
{
    vec3 zero{0.0f, 0.0f, 0.0f};

    // This test is not intended to "pass" :
    // in debug mode, the assertion should trigger.
    // normalize(zero);
}

```

Note:

- this test is deliberately **commented** out in a standard test binary,
- it is often enabled separately or checked manually,
- its role is to explicitly document the **expected behavior in case of invalid usage**.

Example: test an invalid case with explicit error handling

If one wishes to handle invalid inputs without crashing the program, one can use a result type.

```
#include <optional>

std::optional<vec3> normalize_safe(vec3 const& v)
{
    float n = norm(v);
    if (n <= 0.0f)
        return std::nullopt;

    return vec3{v.x / n, v.y / n, v.z / n};
}
```

Test corresponding :

```
#include <cassert>

int main()
{
    vec3 zero{0.0f, 0.0f, 0.0f};

    auto r = normalize_safe(zero);
    assert(!r.has_value()); // the invalid case is indeed detected
}
```

Here, the test explicitly checks that:

- the invalid input is recognized,
- no incorrect value is produced.

Creating tests

Creating exhaustive tests is often a **repetitive** and **time-consuming** task. For a non-trivial function or API, one should generally cover:

- the nominal cases,
- the boundary cases,
- the invalid inputs,
- and sometimes many parameter combinations.

Moreover, when the code evolves (refactoring, API changes, added parameters), the tests must be **updated** to remain consistent with the new contract. This maintenance phase can represent a significant portion of development time.

In this context, AI-assisted code generation tools can be used to accelerate and facilitate the setup of test batteries. They are particularly useful for:

- quickly generate systematic unit tests from a clear specification,
- propose boundary or negative tests often overlooked,
- help adapt or regenerate tests after a code change,
- automatically explore different input combinations.

7.8 Error Handling: Principles and Methodology

A robust program does not merely detect errors: it must **classify them**, **report them correctly**, and **allow the caller to react** appropriately.

Error handling is an integral part of the **design** of the code and its **API**.

Why explicit error handling?

Without a clear error-handling strategy, one obtains:

- silent errors,
- undefined behavior,
- inconsistent internal states,
- bugs difficult to reproduce.

Good error handling allows:

- to make failures **visible and understandable**,
- to separate the nominal code from the error-handling code,
- to explicitly test invalid behaviors,
- to strengthen the contract between caller and function.

Two major categories of errors

The first step is to **distinguish the nature of the error**.

1. Programming errors (bugs)

These are situations that **should never happen** if the code is used correctly.

Examples:

- violation of an invariant,
- out-of-bounds index,
- unexpected null pointer,
- precondition not met.

These errors indicate a **bug**.

Recommended handling:

- `assert`,
- `static_assert`,
- or immediate termination of the program.

```
assert(index < data.size() && "index out of bounds");
```

These errors are generally **not recoverable**.

2. Usage or environmental errors

These are **predictable** situations, even if the code is correct.

Examples:

- missing file,
- malformed data,
- invalid user input,
- unavailable hardware resource.

These errors must be signaled to the caller.

Recommended handling:

- exceptions,
- return codes,
- result types (`optional`, `expected`, `Result`).

Error-handling strategies in C++

The choice of a strategy depends:

- on the type of error,
- on the context (library, application, real-time),
- on performance and readability constraints.

1. Exceptions

Exceptions allow a clear separation between the nominal code and the error-handling code.

```
float parse_float(std::string const& s)
{
    return std::stof(s); // may throw an exception
}
```

Advantages:

- readable nominal code,
- automatic propagation of the error,
- suited for rare errors.

Disadvantages:

- potential cost (depending on context),
- less explicit control flow,
- sometimes forbidden at low-level / real-time.

To be used with **discipline**, and to be clearly documented.

2. Return codes

Historical and explicit approach.

```
bool read_file(std::string const& name, Data& out);
```

Advantages :

- simple,
- no exception,
- explicit control.

Disadvantages :

- easy to forget to check,
- not very expressive without an associated structure.

3. Result types (optional, expected, Result)

Modern and expressive approach.

```
std::optional<float> parse_float_safe(std::string const& s);
```

Or with error information :

```
std::expected<float, ParseError> parse_float(std::string const& s);
```

Advantages :

- makes the error explicit in the type,
- forces the caller to handle it,
- very testable.

Often the best compromise for modern APIs.

Complete example: Robust API with result type

```
#include <fstream>
#include <optional>
#include <string>
#include <vector>

struct ReadError {
    enum class Code { FileNotFound, ParseError };
    Code code;
    std::string message;
    int line = -1;
};

template <typename T>
struct Result {
    std::optional<T> value;
    std::optional<ReadError> error;

    static Result ok(T v) { return {std::move(v), std::nullopt}; }
    static Result fail(ReadError e) { return {std::nullopt, std::move(e)}; }
};
```

Lecture d'un fichier contenant un flottant par ligne :

```
Result<std::vector<float>> read_floats(std::string const& filename)
{
    std::ifstream file(filename);
    if (!file.is_open()) {
        return Result<std::vector<float>>::fail(
            {ReadError::Code::FileNotFound, "Impossible d'ouvrir le fichier"});
    }

    std::vector<float> values;
    std::string line;
    int line_id = 0;

    while (std::getline(file, line)) {
        ++line_id;
        try {
            values.push_back(std::stof(line));
        } catch (...) {
            return Result<std::vector<float>>::fail(
                {ReadError::Code::ParseError, "Erreur de parsing", line_id});
        }
    }

    return Result<std::vector<float>>::ok(std::move(values));
}
```

Test minimal :

```
auto r = read_floats("data.txt");
assert(r.value.has_value() || r.error.has_value());
```

Link to the contract and tests

- the **assertions** verify programming errors,
- the **result types / exceptions** manage recoverable errors,
- the **negative tests** verify that errors are properly detected,
- the **contract** documents what falls under one or the other.

7.9 Best practices for API design

An **API** (*Application Programming Interface*) is the **communication interface** between a piece of code and its users (other functions, other modules, or other developers). It describes **how to use the code**, which operations are

available, which parameters are expected, and which results or errors can be produced.

In C++, an API most often corresponds to the set of declarations visible in header files (.hpp). These files describe what the code allows you to do, without exposing how it does it.

Concretely, a C++ API consists of: - functions and their signatures, - classes and their public methods, - types (structures, enumerations, aliases), - constants and exposed namespaces.

The API user only needs to read the header files to understand: - how to call a function, - which parameters to provide, - which values or errors to expect, - and which rules (preconditions) must be respected.

Source files (.cpp) contain the internal implementation and can evolve freely as long as the API, defined by the headers, remains unchanged.

Thus, in C++, **designing a good API essentially comes down to designing good header files**: clear, coherent, and hard to misuse.

Objectives of a good API

A well-designed API must be:

- **clear** : hard to misuse,
- **predictable** : coherent behaviors in similar situations,
- **documented by the type** : the types express constraints,
- **testable** : easy to use in unit tests,
- **stable** : changes do not unnecessarily break existing code.

Making errors explicit in the API

An API should clearly indicate **how errors are signaled**.

Bad example (silent error)

```
float normalize(vec3 const& v); // what happens if v is zero?
```

Here:

- the contract is implicit,
- the user can call the function without knowing it is invalid,
- the error behavior is ambiguous.

Example with explicit result type

```
std::optional<vec3> normalize(vec3 const& v);
```

Usage :

```
auto r = normalize(v);
if (!r) {
    // invalid case: v is zero
}
```

The error is part of the API: **it cannot be accidentally ignored**.

Example with explicit precondition (programming error)

```
vec3 normalize(vec3 const& v); // precondition: norm(v) > 0
```

Here:

- the caller is responsible,
- the violation is a **programming error**,
- it can be detected via `assert`.

Choose explicitly whether the error is recoverable or not.

Prefer expressive types

Types should carry meaning, not just values.

To avoid: ambiguous parameters

```
void load(int mode); // what does mode mean?
```

The API allows invalid values (`mode = 42`).

Prefer: strong and explicit types

```
enum class LoadMode { Fast, Safe };  
void load(LoadMode mode);
```

Usage :

```
load(LoadMode::Fast);
```

Advantages :

- impossible to pass an invalid value,
- the intent is clear,
- errors are detected at compile time.

Another example: ambiguous bool vs dedicated type

```
void draw(bool wireframe); // what does true mean?
```

Meilleur design :

```
enum class RenderMode { Solid, Wireframe };  
void draw(RenderMode mode);
```

Limiting invalid states

Une bonne API rend les états invalides **impossible or difficult to represent**.

Problematic example: partially valid state

```
struct Image {  
    unsigned char* data;  
    int width;  
    int height;  
};
```

Here, nothing prevents:

- `data == nullptr`,
- `width <= 0`,
- internal inconsistencies.

Better example: invariant established by the constructor

```

class Image {
public:
    Image(int w, int h)
        : width(w), height(h), data(w*h*4)
    {
        assert(w > 0 && h > 0);
    }

    unsigned char* pixels() { return data.data(); }

private:
    int width, height;
    std::vector<unsigned char> data;
};

```

Advantages:

- the object is always valid after construction,
- invariants are centralized,
- the user cannot create an incoherent state.

Separating interface and implementation

The API should expose **what the code does**, not **how it does it**.

Header (.hpp) : interface

```

// image.hpp
class Image {
public:
    Image(int w, int h);
    void clear();
    void save(const std::string& filename) const;
};

```

Source (.cpp) : implementation

```

// image.cpp
#include "image.hpp"

void Image::clear()
{
    // internal details invisible to the user
}

```

Advantages:

- freedom to change the implementation,
- faster compilation,
- API is more stable.

Avoiding hidden side effects

A function should not modify global states in unexpected ways.

Bad example

```

void render()
{
    global_state.counter++; // hidden side effect
}

```

Better example

```
void render(RenderContext& ctx)
{
    ctx.counter++;
}
```

Dependencies are explicit and testable.

Practical API design guidelines

- clearly document the **preconditions** and **postconditions**,
- make errors visible in the type or behavior,
- avoid ambiguous parameters (`bool`, `int` undocumented),
- prefer small and orthogonal functions,
- test the API as if you were an **external user**,
- consider that the API is harder to modify than the implementation.

Key idea to remember

A good API prevents errors even before the program runs.

It guides the user toward the right usage, makes errors explicit, and facilitates testing, maintenance, and evolution of the code.