C++ Programming

Applications to Computer Graphics

[CSC-43043-EP]

2026

Damien ROHMER - damien.rohmer@polytechnique.edu

Contents

1 Introduction to C++

1.1 Preface

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

C++ distinguishes itself from other programming languages by its unique ability to combine low-level performance with high-level abstraction. Direct heir to C, it allows precise control of memory and hardware, indispensable in domains 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 an indirection step 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 specificity of C++ is its simultaneous support for several 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 design of complex software. - **Generic**, thanks to **templates**, which allow writing reusable code independent of types. - **Functional**, increasingly present since C++11 with the **lambdas** and the algorithms of the standard library.

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

Evolutions of C++

The C++ language continues to evolve with regular updates

- **C++98** and **C++03** standardized the language and its standard libraries.
- **C++11**, called “Modern C++”, marked an important turning point with, notably, the introduction of range-based loops, easier initialization of structures, the auto keyword, smart pointers, and lambda functions.
- **C++14** and **C++17** enriched the syntax and the standard library (structured bindings, filesystem, parallelism).
- **C++20** introduced the concepts, coroutines, and ranges.
- **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 stringent performance, real-time, or compute-intensive requirements.

Application domains

- **Scientific and real-time applications**: physical simulations, numerical computations, embedded systems.
- **Game engines** (Game Engines): Unity, Unreal Engine, Godot, as well as practically all AAA games heavily use C++.
- **2D/3D software**: Maya, Blender, Photoshop, Premiere Pro, Catia, SolidWorks rely largely on C++.
- **Parallel computing and GPUs**: 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 written largely in C and C++.
- **Web and massive-scale services**: browsers (Chrome, Firefox) and critical infrastructures (AWS, Facebook, etc.) use C++ for the core performance-critical parts.

Strengths (+)

- **Performance**: direct compilation to native machine code, allowing very fine-grained optimizations.
- **Robustness**: a mature language, used and tested at very large scales.
- **High and low level**: rare combination that allows both writing near-hardware code and using modern high-level abstractions.
- **Specificity**: this duality is present only 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 major source of complexity and programming errors.
- **Build 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 philosophy differs.
  - C++ is compiled to native machine code, which makes it very performant and suitable for systems where every computation 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++ gives the programmer fine-grained control over allocation and deallocation.
- **C++ vs Python**
  Python is renowned for its writing simplicity and development speed, but remains an interpreted language, thus much slower in execution.
  - C++ requires more rigor and syntax, but enables achieving maximum performance.
  - In practice, Python is often used for prototyping, scripting and data analysis, while C++ is favored for performance-critical parts (3D engines, scientific computing, 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 memory leaks or illegal memory access thanks to its system of **borrowing and ownership**.
  - C++ offers more flexibility and has a huge existing software ecosystem, but at the cost of the rigor needed to avoid mistakes and security vulnerabilities.
  - Rust is seen as a modern and secure alternative, but C++ remains today largely dominant in industry and in the available libraries.

1.2 First C++ program

We consider the following C++ program:

```cpp
// standard library for input/output
#include <iostream>

int main() {
    // display a message on the command line
    std::cout << "Hello, world!" << std::endl;

    // end of program
    return 0;
}
```

Line-by-line explanations

1. `#include <iostream>`
   - This directive tells the compiler to include the standard library **iostream**, which allows using input and output streams (`std::cin`, `std::cout`, etc.).
2. `int main()`
   - This is the main function of the program.
   - Every C++ program must have a `main` function.
   - Its execution always starts 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 output.
4. `return 0;`
   - Indicates that the program terminated successfully.
   - The returned value is passed to the system.

Note: Each statement 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, we use a **C++ compiler**. On Linux or macOS, the most common compilers are:

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

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

```
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`).

The 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 memory area 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

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

int main() {
    int age = 20;                 // integer
    float taille = 1.75f;         // floating-point number (single precision)
    double pi = 3.14159;          // floating-point number (double precision)
    std::string nom = "Alice";    // string

    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**.

  ```cpp
  int entier = 325;
  ```

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

  ```cpp
  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.

  ```cpp
  bool estEtudiant = true;
  ```

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

  ```cpp
  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`s 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';
  ```

  A `char` can also be used to manipulate memory directly at the byte level.

Important notes

1. **Integer division vs floating-point division**

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

   "‘cpp

```
    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:

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

2. **The keyword `auto`**

   It allows the compiler to automatically deduce the type:

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

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

3. **Uninitialized variables**

   In C++, built-in variables are not initialized by default.

   ```cpp
   int a; // contains an undefined value
   ```

   ☐ To avoid indeterminate behaviors, it is advisable to always initialize your variables:

   ```cpp
   int a = 0;
   ```

Declaration without initialization (example)

```
int compteur;    // uninitialized
compteur = 10;  // assignment of a value later
```

[Attention]: an uninitialized variable contains an undefined value and must 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 moment of its declaration** and **cannot be modified** afterwards.

```
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;
}
```

Why this matters

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

1.4 Formatted output and input with printf, scanf

`printf` and `scanf` (C heritage)

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

- **`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 <cstdio>

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

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

Output :

```
Age: 20 years, height: 1.75 m
```

- `%d` : integer (**`int`**)
- `%f` : floating point (**`float`** or **`double`**)
- `%.2f` : floating point displayed with two decimals

Example of reading with scanf

```
#include <cstdio>

int main() {
    int age;
    printf("Enter your age: ");
```

```
scanf("%d", &age);   // & = memory address
printf("You are %d years old.\n", age);
return 0;
}
```

In `scanf`, it is necessary to provide the **address** of the variable (here `&age`), because the function directly modifies its value.

Perfect □ Here is a concise recap table of the main formats usable with `printf` and `scanf`, written in **LaTeX-friendly** style (no non-representable special characters).

Main format specifiers (`printf` / `scanf`)

Specifier	Expected type	Example usage	Displayed result
`%d`	signed integer (`int`)	`printf("%d", 42);`	`42`
`%u`	unsigned integer (`unsigned`)	`printf("%u", 42u);`	`42`
`%f`	floating-point (`float` or `double`)	`printf("%f", 3.14);`	`3.140000`
`%.nf`	floating-point with `n` decimals	`printf("%.2f", 3.14159);`	`3.14`
`%e`	floating-point in scientific notation	`printf("%e", 12345.0);`	`1.234500e+04`
`%c`	character (`char`)	`printf("%c", 'A');`	`A`
`%s`	string (`char*`)	`printf("%s", "Bonjour");`	`Bonjour`
`%x`	integer in hexadecimal (lowercase)	`printf("%x", 255);`	`ff`
`%X`	integer in hexadecimal (uppercase)	`printf("%X", 255);`	`FF`
`%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 on 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 on the **heap memory**: slightly more costly to allocate, 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.).
- Not widely used in modern C++, except to interact with C code or for very low-level needs.

Simple example with `std::vector`

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

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

    // Adding 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;

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

    // Modification of an element
    vec[1] = 12;

    // Traversing 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 an **undefined behavior**, which can cause the program to crash.

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

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

Resizing

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

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

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

```cpp
#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.
- **`std::vector<T>`**: dynamic array, size modifiable, stored on the heap.
- 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 may 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:

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

// instructions if the condition is false }

~~~
**[Warning]** The braces `{}` are **optional** if only a single statement is present:
```cpp
if (x > 0)
    std::cout << "x is positive" << std::endl;
~~~

Example:

```cpp
int age = 20;

if (age >= 18) {
    std::cout << "You are an adult." << std::endl;
} else {
    std::cout << "You are a minor." << 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 << "Very good!" << std::endl;
else if (note >= 10)
    std::cout << "Passing." << std::endl;
else
    std::cout << "Fail." << std::endl;
```

The loops

The while loop

General structure:

```
while (condition) {
    // repeated instructions as long as 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; continuation-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 : container) {
    // instructions using the variable
}
```

Example:

```
#include <vector>

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

    for (int v : values)
        std::cout << v << std::endl;
}
```

Extension : switch / case

The `switch` allows testing several values of the same integer or character variable.
General structure:

```
switch (variable) {
    case value1:
        // instructions
        break;
    case value2:
        // instructions
        break;
    default:
        // default instructions
}
```

[Warning] It only works with integer or character types.
The keyword `break` prevents executing 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 key. Each key is unique and allows efficient access to the corresponding value (lookup in O(log n)).

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

Simple example: counting word frequency

```
#include <iostream>
#include <map>
```

~~~cpp
#include <string>

int main() {

```cpp
std::map<std::string, int> counts;

// Insertion / increment
counts["pomme"] = 5;
counts["banane"] = 4;
counts["avocat"] = 8;
counts["pomme"]++;

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

// Lookup 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: variable local to a block

```cpp
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

```cpp
int main()
{
    int x = 5; // x is defined in the block of the function main()
    if (true) {
        std::cout << x << std::endl; // x can be used in this sub-block
    }
    // x 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 multiple variables with the same name in a single block.
  - This is possible in **sub-blocks**:

```cpp
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 particular task.
The general syntax is as follows:

```cpp
typeRetour nomFonction(type nomArgument1, type nomArgument2, ...)
{
    // corps de la fonction
    return valeur;
}
```

Simple example

```cpp
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 function's name and parameter types is called the function's signature or header.
- The remainder 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)

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

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

Correct example (declaration followed by definition)

```cpp
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

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

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)`:

```cpp
#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);`

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

Function Overloading

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

Example

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

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

// Solves ax^2 + bx + c = 0 (one root)
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);       // Calls the 1st version
    float y = solve(1.0f, 2.0f, 1.0f); // Calls the 2nd version

    std::cout << "Linear solution : " << x << std::endl;
    std::cout << "Quadratic solution : " << 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:
- The modifications made inside the function stay local.

- For large objects (vectors, arrays, structures), copying can be **costly** in terms of performance.

Example with pass-by-copy

```cpp
#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

We can use the symbol `&` in the signature to pass an argument **by reference**.
This allows directly modifying the original variable :

```cpp
#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 the main: 1
```

Here, `vec` is passed **by copy** 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 the main: 2
```

Constant references

If we want to avoid copying **without modifying** the vector, we can use a **constant 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 :

- of **attributes** (data members),
- and of **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>

// Déclaration 'dune structure
struct vec3 {
    float x, y, z;
};

int main()
{
    // Création 'dun vec3 non initialisé
    vec3 p1;

    // Création et initialisation 'dun vec3
    vec3 p2 = {1.0f, 2.0f, 5.0f};

    // Accès et modification des attributs
    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; // Par défaut : public
};

class vec3 {
  public:
    float x, y, z; // Doit être indiqué explicitement
};
```

Main difference :

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

In practice :

- We often use `struct` for simple objects that aggregate 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 manipulate directly its attributes.

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

struct vec3 {
    float x, y, z;

    float norm() const;    // method that does not modify the object
    void display() const;  // same
    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 directly the object's attributes without using `this->`, although it is possible.
- We generally separate the **declaration** (in the struct/class) and the **implementation** (with `ClassName::MethodName`).
- 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.

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

struct vec3 {
    float x, y, z;

    // Empty constructor
    vec3();

    // Custom constructor
    vec3(float v);

    // Destructor
    ~vec3();
};

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

```cpp
// Initialisation avec une valeur commune
vec3::vec3(float v) : x(v), y(v), z(v) { }

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

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

    return 0; // appelle ~vec3()
}
```

Default constructor or destructor (= `default`)

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

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

    // Génère automatiquement un constructeur par défaut
    vec3() = default;

    // Génère automatiquement un destructeur par défaut
    ~vec3() = default;
};
```

This is equivalent to writing nothing, but has two advantages:

- Readability: this makes explicit that a constructor or destructor exists and should be the one provided by the compiler.
- Robustness: helps avoid certain implicit suppressions 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 developer. For example, the standard can also be defined as an independent function:

```cpp
#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 unnecessary copying of the object.

1.12 External file I/O

In C++, the library `<fstream>` allows you 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`** : for combining reading and writing.

Example: writing a vec3 to a file

We want to save the coordinates of a `vec3` to 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"); // ouverture en écriture
    if (!file.is_open()) {
        std::cerr << "Erreur : impossible 'douvrir le fichier !" << std::endl;
        return 1;
    }

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


    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 read this `vec3` back from the file:

```
#include <iostream>
```

#include #include
struct vec3 { float x, y, z; };
int main() { vec3 p;

```
std::ifstream file("vec3.txt"); // ouverture en lecture
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; // lecture des trois valeurs
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` : read (default for `ifstream`).
* `std::ios::out` : write (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:

```cpp
std::ofstream file("log.txt", std::ios::app); // ouverture en ajout
file << "Nouvelle entrée" << std::endl;
~~~

1.13 Code file organization

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

A typical organization with C++ classes 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 must know to use the class.
2. **Implementation file (.cpp)**
    - Contains the **code of the methods** and functions declared in the `.hpp`.
    - Performs 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`

```cpp
#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`

```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`

```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;
}
```

Important notes

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

About `#pragma once`

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

With `#pragma once`, the compiler guarantees that the file's content 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 (`.cpp` and `.hpp` files) into an executable program understandable by the computer. This transformation happens in several steps. The compiler starts by analyzing the code and translating it into **assembly code**.

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

Then, 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** assembles the various 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)
   Executable (binary program)
```

Diagram with multiple source files

```
 main.cpp   vec3.cpp   utils.cpp
     ↓         ↓          ↓
  (compiler) (compiler) (compiler)
     ↓         ↓          ↓
 main.o    vec3.o     utils.o
     ↓         ↓          ↓
  [linker]
            ↓
```

Executable program

Example of assembly code

C++ Example

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

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

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

Explanations

- **`edi` and `esi`**: registers used to pass the 1st and 2nd 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 an 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 (a single 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 multiple files, it becomes tedious to compile everything by hand. We then use a **Makefile** with the **make** tool, which describes dependencies and the compilation rules.

Minimal Makefile example:

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

```
# Default target (here: "main")
all: main
# General syntax :
# target: dependencies
#     command(s) to execute

# Build of the executable "main"
main: main.o vec3.o
    g++ main.o vec3.o -o main
# General syntax :
# executable: object_files
#     compiler object_files -o executable

# Rule to generate the object main.o
main.o: main.cpp vec3.hpp
```

```
    g++ -c main.cpp
# General syntax :
# file.o: file.cpp header.hpp
#     compiler -c file.cpp

# Rule to generate the object vec3.o
vec3.o: vec3.cpp vec3.hpp
    g++ -c vec3.cpp
# General syntax :
# file.o: file.cpp header.hpp
#     compiler -c file.cpp

# Cleaning intermediate files
clean:
    rm -f *.o main
# General syntax :
# clean:
#     command to remove generated files
```

Windows

On Windows, the compiler is provided directly by **Microsoft Visual Studio** (MSVC).

It does not rely on `make` or 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 handles launching the MSVC compiler automatically when you press "Build" or "Run". Thus, it is not necessary (and not practical) to manually invoke `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 tool for **project generation**.
- It reads a configuration file (`CMakeLists.txt`) and automatically generates the files adapted to your system :
  - **Linux/MacOS** → a **Makefile** usable with `make`.
  - **Windows** → a **Visual Studio project** (`.sln`).

Example usage on Linux/MacOS:

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

In summary

- Linux/MacOS: compilation via `g++` or `clang++`, automation via **Makefile**.
- Windows: compilation via MSVC through a Visual Studio project.
- CMake: a 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 for memory addressing (8 bytes on 64-bit machines)
```

Important notes:

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

2.1 Integer Encoding

Binary Representation

An integer is represented in **binary**:

- Each bit is 0 or 1.
- A set of bits is grouped into **bytes** (8 bits).
- 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 several bytes:

- 4 bytes (`int` classic) = 32 bits ⟶ up to `2^32` possible values.
- 8 bytes (`long long`) = 64 bits ⟶ up to `2^64` possible values.

Unsigned Integers

An **unsigned int** on 4 bytes (32 bits) encodes values from `0` to `2^32 - 1 = 4 294 967 295`.
Example in hexadecimal (practical representation of bytes):

- `00000000` ⟶ `0`
- `FFFFFFFF` ⟶ `4294967295`

Reminder:

- 1 byte (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
Inverted → 11111010
Add +1 → 11111011 = -5
```

Consequence:

- 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

Take the signed integer encoded on 2 bytes:

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

- Interpreted as **unsigned**: `50317`.
- Interpreted as **signed two's complement**:
    - Bit inversion → `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:

- The 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 what we call **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
   Content: 78     56     34     12
   ```

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

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

   ```
   Address: 1000   1001   1002   1003
   Content: 12     34     56     78
   ```

Why is this important?

- **Network compatibility** Protocols (TCP/IP, etc.) require Big Endian (*network byte order*). Classic 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 explicitly state the byte order.

2.4 Summary of fundamental types

Type	Description	Typical size (x86/64-bit)	Example declaration
`char`	ASCII character (or signed small integer)	1 byte	`char c = 'A';`
`bool`	boolean value (`true` or `false`)	1 byte (vector-optimized)	`bool b = true;`
`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` \| signed long integer (guaranteed to be at least 64 bits) \| 8 bytes \| `long long x = 1e12;` \|
`unsigned` \| unsigned integer (≥0 only) \| same size as signed \| `unsigned u = 42;` \|
`float` \| single-precision floating-point number (IEEE 754) \| 4 bytes \| `float f = 3.14f;` \|
`double` \| double-precision floating-point number \| 8 bytes \| `double d = 2.718;` \|
`long double` \| extended-precision floating-point (architecture-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';` \|

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

2.5 Getting the size with `sizeof`

In C and C++, the operator `sizeof` 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
```

2.6 Important notes

- `sizeof(type)` is evaluated **at compile time**, without executing 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 sizes for sure on your machine, it is advisable to write a small program using `sizeof`.

2.7 Fixed-size types

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

Main fixed-size 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 supplementary examples:

- `int_fast32_t`, `uint_fast32_t` : integer types at least 32 bits but 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 integer types capable of holding a pointer value

Example usage:

```
#include <cstdint>
#include <cinttypes> // for PRIu32, PRId64, ...
#include <cstdio>

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; // 12
unsigned b = 0b1010; // 10

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;    // inversion of all bits

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

// display in hex / decimal as needed
```

Masks and bit tests

We use masks 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; // flips the state of bit 2
```

Important tips

- Use unsigned types (`unsigned`, `uint32_t`, `uint64_t`) for bitwise operations: the behavior of shifts on negative signed integers can be undefined or implementation-defined.

- Left shift `x << n` multiplies by `2^n` provided it does not overflow. Right shift `x >> 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)

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

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

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

2.9 Summary

- The standard types cover the **signed/unsigned integers**, the **floating-point numbers** and the **characters**.
- Their size is not always fixed (except `char` = 1 byte guaranteed).
- `sizeof` lets you determine the exact 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 viewed as a **large linear array** of cells.

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

We can therefore imagine memory as a succession of numbered cells:

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

Here:

- each line represents **one memory byte**,
- 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 you declare a variable in C++:

```cpp
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 viewed 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 certain variables start at addresses that are multiples of 2, 4 or 8. This makes memory access easier for the processor.

Importance of the address

The memory address is what allows:

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

Example of an analogy

One can compare memory:

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

To access data, 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**, i.e., the position of its first byte in the big memory array. In the C language (and thus also in C++), you can access this address using the operator `&` (called *address of*).

Simple example

```c
#include <stdio.h>

int main() {
    int a = 42;
```

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

    return 0;
}
```

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

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

- `%d` displays the integer value (`42` here).
- `%p` displays 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.

```c
#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 read value 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.

Observing the address

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

```c
#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 variables differently (stack, memory alignment, etc.).

3.3 Passing arguments

Passing 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:

```c
#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 5
    return 0;
}
```

Memory explanation:

- `a` in `main` occupies a memory area.
- During the call `increment(a)`, the value `5` is copied into a new local variable `x` in the function.
- Modifying `x` does not change `a`, because these are two independent variables.

Passing by address with a pointer

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

```c
#include <stdio.h>

void increment(int* p) {
    *p = *p + 1; // modifies the value at the pointed-to 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 (e.g. 1000).
2. The expression `&a` yields this address (1000).
3. When calling the `increment(&a)`, it's not **the `a` that is copied**, but its **address** (1000).
   - The function therefore 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` in the same place.
5. Since `p` designates the memory of `a`, the variable `a` is actually modified.

Summary of mechanisms

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

Diagram (simplified ASCII):

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

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

increment:
 p = 1000
 *p → value stored at address 1000 → 5
 *p = 6   (modifies the 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 **an arbitrary 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; // pointer initialized, but points to nothing

    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 → probable crash
```

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

Correct example

```
int* p = nullptr;   // safe pointer, but empty
if(p != nullptr) {
    *p = 10;        // only access 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 contiguous sequence of bytes. This means that the elements follow one another, 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 notice that the addresses are spaced by 4 bytes (the size of an `int`), which confirms 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` : advances 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
```

Adaptation to the element size

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

Example with `double`

```c
#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 starting 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
```

- In general:

```
Address(p + N) = Address(p) + N * sizeof(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** (you 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.
```

~~~
```cpp
#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

- The C arrays and the `std::vector`s store their elements in a **contiguous** manner.
- This allows fast index-based access (`tab[i]`) or via pointer arithmetic (`*(p+i)`).
- The `std::vector`s also offer a **dynamic size** and safe memory management, but preserve the same fundamental properties of contiguity.

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

```c
#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 another contiguously.

Padding and alignment

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

Example:

```c
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 global alignment)
```

Total size: 24 bytes.

Contiguity in classes

In C++, a `class` behaves like a `struct` from a 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++, we can store several objects `struct` or `class` in a **`std::vector`**. The `vector` guarantees that the elements are placed **contiguously in memory**, exactly as for 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 << "Address of the first Point2D : " << &points[0] << std::endl;
    std::cout << "Address of the second Point2D : " << &points[1] << std::endl;
    std::cout << "Address of the third 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 arranged **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 may be larger than the sum of the fields.
- A **`std::vector<struct>`** allows creating a dynamic array of structures also contiguous in memory.
- This contiguity makes fast memory traversal possible and compatibility with C functions via `points.data()`.

3.6 Memory Organization AoS vs SoA

When manipulating 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 :

```cpp
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 one only wants to process the `x`, one must 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 :

```cpp
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 field level:

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

Advantage: very efficient if one performs bulk processing on a single field (e.g. applying a transformation on 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 another.
- **SoA**: contiguity *by field*. Each field is grouped in its own array, and the values follow per dimension.

Both approaches thus 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 Memory allocation and deallocation

Memory allocation consists of dynamically reserving a memory region during program execution, and deallocation consists of freeing it when it is no longer necessary. This dynamic management is indispensable when the size of the data is not known at compile time or when their lifetime exceeds a local block.

In C and C++, dynamic memory is stored in a region called the heap, as opposed to the stack used for local variables.

Stack vs heap

Variables on the stack:

- automatic allocation upon entering a block,
- automatic release upon exiting the block,
- very fast,
- size is limited.

```
void f() {
    int x = 10; // on the stack
}
```

Dynamic memory on the heap:

- explicit allocation by the programmer,
- lifetime independent of the blocks,
- must be freed explicitly.

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

In C, we use functions from the standard library `<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;
}
```

Points to note:

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

Dynamic allocation of arrays 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 a **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
}
```

Classic error example: memory leak

```
void f() {
    int* p = new int(10);
    // forgetting to delete
}
```

Each call to `f`, memory is allocated but never freed: **memory leak**.

Double free (dangerous)

```
int* p = new int(5);
delete p;
delete p; // ERROR: double free
```

This leads to undefined behavior.

Null pointer after release

Good practice:

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

This avoids accessing a freed pointer (dangling pointer).

Generic allocation with `void*`

In C, `malloc` returns a `void*`, which allows allocating **any type of structure**.

```
struct Point2D {
    float x, y;
};

struct Point2D* p = (struct Point2D*)malloc(sizeof(struct Point2D));
```

But:

- no constructors are called,
- no type checking is performed.

Recommended approach in modern C++

In modern C++, we **avoid manual memory management as much as possible**.
To prefer:

- `std::vector` for dynamic arrays,
- `std::unique_ptr` for unique ownership,
- `std::shared_ptr` for shared ownership,
- automatic allocations on the stack when possible.

Example with `std::vector`:

```cpp
#include <vector>

std::vector<int> v(10); // automatic allocation and deallocation
```

Summary

- Dynamic allocation is performed on the **heap**.
- In C: `malloc` / `free` (raw memory, `void*`).
- In C++: `new` / `delete` (types + constructors).
- Every allocation should 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 should be limited to necessary cases and replaced as much as possible by safe abstractions.

3.8 The generic pointer `void*`

In C and C++, there exists a peculiar 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 associated type information.

Declaration and principle

```cpp
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:

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

Simple example

```c
#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 contain the address of `a` then that of `b`,
- but **no type information is preserved**.

Impossibility of dereferencing directly

It is **forbidden** to do:

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

Why?

- `*p` means "access the value pointed to",
- but the compiler does not know **either the size** nor the nature of the pointed-to type.

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

Explicit conversion (cast)

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

```c
#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: "consider this address as an `int*`",
3. we can then dereference correctly.

Example with several types

```c
#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 bytes of raw memory

```
// explicit interpretation
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.



### Practical usage

The `void*` is mainly used :

* in **plain C** (generic interfaces, system libraries),
* for low-level APIs,
```

~~~~
* for manipulating raw memory,
* in historic generic functions (`qsort`, `bsearch`).

In **modern C++**, we prefer :

* 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.



## References



In C++, the **references** are introduced as a simpler and safer alternative to pointers.
They can be seen as an **alias** to an existing variable, and mainly as a **syntactic sugar** above 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 don't need to write `*` or `->` : the reference is handled as if it were the variable
     itself.

---

### Passing arguments: value, pointer, reference

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

```cpp
#include <iostream>

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

int main() {
    int a = 5;
    int c = ma_fonction(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)

```cpp
#include <iostream>

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

int main() {
    int a = 5;
    ma_fonction(&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.
- More verbose syntax, with risk of errors (null pointer, forgetting the `*`).

Pass-by-reference (style C++)

```cpp
#include <iostream>

void ma_fonction(int& b) {
    b = b + 2; // we have the impression of manipulating b as if it were a local variable
}

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

Here :

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

~~~

Initialization of references

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

```cpp
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 to:

- avoid an expensive copy,
- while guaranteeing that the object will not be modified.

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

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

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

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

Concrete example: vectors and structures

```cpp
#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 const 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:

```cpp
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; // reading
    v.value(10) = 42;                      // writing 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 accessor).

To avoid

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

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 with the readability of clear code.

3.9 Dynamic Allocation

Until now, we have seen **automatic variables** (declared inside 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: keeping an array created in a function, handling large structures, or building dynamic graphs). In this case, we use **dynamic memory**, allocated on the **heap**.

The stack vs the heap

Characteristic	Stack (stack)	Heap (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	`int a;` or `int tab[10];`	`new int;` or `new int[n];`

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

Example: limited lifetime with automatic variables

```cpp
#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 when exiting `createValue()`. The returned pointer becomes **dangling** (dangerous).

Example: extended lifetime with dynamic allocation

```cpp
#include <iostream>

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

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

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

Dynamic allocation of an array

```
#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 required
}
```

Utility: `n` is known only at runtime → impossible to use a static array.

Memory Diagram

```
Pile (stack)                 Tas (heap)
------------                 ------------
int main() {                 new int[3]
  int n = 3;                 ---------------
  int* arr = new int[n]; --> | 0 | 1 | 2 | ...
                             ---------------
}
```

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

Common Problems

1. **Memory leak:**

```
void f() {
    int* p = new int(10);
    // oubli de delete → fuite mémoire
}
```

→ the memory remains occupied as long as the program runs.

2. **Double free:**

```
int* p = new int(5);
delete p;
delete p; // error : double free
```

3. **Use after free:**

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

Example: resizing (principle)

When resizing a dynamic array manually, one 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: Reallocation of an array always requires a **new allocation + copy**, 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};
    n1->next = n2;

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

    // freeing
    delete n2;
    delete n1;
}
```

Each element (`Node`) is allocated separately on the heap. [**Attention**]: It is important to **free each element** to avoid leaks.

Modern Best Practices

In C++, we nowadays avoid direct `new` / `delete`. We prefer:

1. `std::vector` for dynamic arrays

```
#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 << " ";
}
```

→ Memory is managed automatically (constructor / destructor).

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

Smart pointers are classes in the C++ standard library () that encapsulate a raw pointer (T*) and **automatically manage the lifetime of the pointed resource**.

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

Example with `std::unique_ptr`

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

int main() {
    std::unique_ptr<int> p = std::make_unique<int>(42);
    std::cout << *p << std::endl;
} // delete automatique ici
```

Explanation:

- `std::unique_ptr<int>` holds exclusive ownership of the resource: 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 freed, even in the case of an exception or premature exit from the function.

Characteristics of `std::unique_ptr`:

- Ownership is *unique* (non-copyable).
- Transferable via `std::move()`.
- Lightweight, safe and very fast.
- Ideal for representing exclusive ownership of a resource.

Example of usage with ownership transfer:

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

void display(std::unique_ptr<int> p) {
    std::cout << *p << std::endl;
}

int main() {
    std::unique_ptr<int> a = std::make_unique<int>(7);
    display(std::move(a)); // ownership transfer
    // a no longer owns anything here
}
```

Example with `std::shared_ptr`

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

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

Detailed explanation:

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

Thus, the memory is freed 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	Copyable	Resource sharing	Destruction	Typical use case
`std::unique_ptr<T>`	No	No	Automatic, as soon as the pointer goes out of scope	Exclusive ownership (e.g., internal component of an object)
`std::shared_ptr<T>`	Yes	Yes (reference counter)	Automatic, when the last pointer is destroyed	Resources shared among multiple objects or functions

---

Memory Illustration

```
Unique_ptr case :
+----------------------+
| unique_ptr<int> p    ──►| [42]
+----------------------+
           |
    automatic deletion at the end of the block


Shared_ptr case :
+----------------------+        +----------------------+
| shared_ptr<int> p1   ──┐|     | counter = 2         |
| shared_ptr<int> p2   ──┘──►| [10]
+----------------------+        +----------------------+
                      |
      automatic deletion when counter = 0
```

Why smart pointers replace new and `delete`

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

In summary

- `std::unique_ptr` : exclusive, safe, ideal by default.
- `std::shared_ptr` : shared, useful when several entities need to access the same resource.
- Both rely on the **RAII**, guaranteeing automatic release and code safety.

4 Classes

4.1 Introduction

In C++, a **class** allows grouping, within 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 the structuring of code, its readability and maintenance.

Regrouper des données : premier exemple avec `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 accessible directly:

```
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.

Ajouter un comportement : méthodes

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
```

`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 some members accessible, you must specify the access levels.

Attributs publics et privés

We use the keywords `public` and `private` to control access to members:

```cpp
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 :

```cpp
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 et sécurité

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

```cpp
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.

Bonnes pratiques

- Use **`struct`** for:
  - simple objects,
  - primarily data carriers,
  - without complex invariants.
- Use **`class`** for :
- encapsulate data,
  - control access,
  - guarantee internal invariants.

4.2 Initialization, Constructors

In C++, the initialization of an object is handled by the **constructors**. A constructor is a special function (same name as the class, no return type) automatically called 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 primitive types (`int`, `float`, etc.), they are not necessarily initialized automatically.

```cpp
#include <iostream>

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

int main() {
    vec3 v; // x,y,z undefined !
    std::cout << v.x << std::endl; // undefined behavior
}
```

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

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

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

Default constructor

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

```cpp
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 syntax `: x(...), y(...), z(...)` is the initializer list. It initializes the attributes before entering the constructor body.

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

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

Usage:

```cpp
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 then reassignment) and it is required for certain members.

Overloaded constructors

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

```cpp
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)
}
```

One-argument constructor and `explicit`

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

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

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

```cpp
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` members and references must be initialized via the initializer list.

```cpp
struct sample {
```

```cpp
int const id;
float& ref;

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

};

Sans liste ’dinitialisation, ce code ne compile pas, car `id` et `ref` ne peuvent pas être “”assigned après coup :
     they must be initialized immediately. // Note: I kept the French quotes around assigned as in the original.

Destructor (reminder)

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

```cpp
#include <iostream>

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

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

Best practices

- Always initialize attributes (via constructor or `{}`).
- Prefer the initialization list `:` to initialize members.
- Use `explicit` for single-argument constructors, unless implicit conversion is desired.
- Design constructors to guarantee 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 like `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 a **member method** of the class,
- or as a **non-member function** (often preferable for symmetrical 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:

```cpp
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

The 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 `<=>` operator (three-way comparison), but its use goes beyond the scope of this introduction.

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.

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 **constant references** for read-only parameters.
- Return `*this` by reference for mutating 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 writing code that is more readable and expressive, but it must remain **simple, coherent and predictable**.

4.4 Inheritance

Inheritance is a central mechanism of object-oriented programming that allows you to **define a new class from an existing class**. The derived class inherits attributes and methods from the base class, which promotes **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 {
    // content specific to 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); // méthode héritée de 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 try to call the default constructor, which can cause 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 derived classes.
- `private`: accessible only within the base class.

Example:

```
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; // autorisé car x est protected
    }
};
```

Method overriding

A derived class can **override** a method of the base class to provide a specific behavior.

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

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

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

```cpp
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` hides 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:

```cpp
class Vehicle {
  public:
    float speed;

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

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

~~~

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

Best practices

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

Polymorphism

Polymorphism

The **Polymorphism** allows to manipulate objects of different types **through a common interface**, while
     automatically calling the correct implementation according to the **actual 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 one wants to **store heterogeneous objects in the same container** and handle
     them uniformly.

The problem: storing different objects in a single container

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

```cpp
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 relation**. 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 forced either to:

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

Polymorphism provides an elegant solution to this problem.

Common interface via a base class

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

```
class Shape {
  public:
    virtual float area() const = 0; // pure virtual method
    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()` :

```
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_) {}
```

```cpp
    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 a single container:

~~~cpp
#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 real type (`Circle` or `Rectangle`).

### Role of `virtual` and dynamic dispatch

The call:

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

is resolved at run time 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

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

```cpp
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?

We cannot store derived objects directly 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 issue and enable dynamic binding.

Cost and alternatives

Dynamic polymorphism involves:

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

In performance-critical loops, one may sometimes favor **static polymorphism** via templates, to be 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 allocated dynamically with `new`,
- the container stores pointers to the base class `Shape`,
- the calls to `area()` are resolved dynamically,
- **the programmer must explicitly release 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` portion of the object, and not the derived portion (`Circle`, `Rectangle`), leading to resource leaks and undefined behavior.

Common problems with raw pointers

The use of raw pointers exposes you to several classic mistakes:

- forgetting to call `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 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 safeguard overrides.
- Combine polymorphism and 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.

4.5 Access control: `const`

In C++, the keyword **`const`** applied to the **class methods** plays a central role in access control and in code safety. It is not merely a documentation hint: a `const` method and a non-`const` method are considered by the compiler as **two different methods**, able to **coexist 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 state of the object**.

```
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

An object declared `const` 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** (access, calculations) methods,
- the **modification** (state-updating) methods.

`const` and non-`const` methods: two distinct signatures

A `const` method and a non-const method bearing the same name are not the same function. They can be defined simultaneously in a class.

Here:

- the non-const version is called on a mutable 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 object's const-ness.

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; // ERROR
float v = c.value(); // const version
```

Conceptual significance

This distinction allows:

- to express clearly 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`.
- Systematically provide both a `const` and a 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.

4.6 Keyword: `static`

4.7 Access management: the `static` keyword in classes

The keyword **`static`**, applied to class members, deeply 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 to represent global data or behaviors tied 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.

```cpp
class Counter {
  public:
    Counter() {
        ++count;
    }

    static int get_count() {
        return count;
    }

  private:
    static int count;
};
```

The declaration inside the class is not enough. The static attribute must be defined once in a `.cpp` file:

```
```

int Counter::count = 0;

Usage :

```cpp
Counter a;
Counter b;
Counter c;

int n = Counter::get_count(); // n = 3
```

All the `Counter` objects share the **same variable** `count`.

Accessing static attributes

A static attribute:

- can be accessed **without an object**, via the class name,
- can also be accessed from an object, but that is not recommended.

```cpp
Counter::get_count(); // recommended form
```

This underlines the fact that the data belongs to the class, and not to a particular instance.

Static methods

A **static method** is a function associated with the class, but independent of any instance.

```cpp
class MathUtils {
  public:
    static float square(float x) {
        return x * x;
    }
};
```

Usage :

```cpp
float y = MathUtils::square(3.0f);
```

Constraints of static methods

A static method :

- has no **this** pointer,
- can access only the **static members** of the class,
- cannot directly access non-static attributes.

```cpp
class Example {
  public:
    static void f() {
        // x = 3; // ERROR: x is not static
        y = 4;    // OK
    }

  private:
    int x;
    static int y;
};
```

`static` and initialization

Since C++17, it is possible to initialize directly some static attributes in the class if they are `constexpr` or of literal type.

```cpp
class Physics {
  public:
    static constexpr float gravity = 9.81f;
};
```

Usage :

```cpp
float g = Physics::gravity;
```

In this case, no additional definition in a `.cpp` is necessary.

Common use cases

The keyword `static` is used for:

- counting the number of instances of a class,
- storing global constants related to a concept,
- sharing common resources,
- grouping utility functions related to a class,
- implementing factories (factory methods).

Example: unique identifier per object

```cpp
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 **belonging to the class**, not to the object.
- Access static members via `ClassName::member`.
- Limit the use of mutable static attributes to avoid hidden dependencies.
- Prefer `constexpr static` for constants known at compile time.

Key takeaway

A static member is **unique and shared**, it belongs to the **class**, not to the objects.

5 Threads and Parallelism

The **parallelism** designates the ability of a program to execute **multiple tasks simultaneously**. In C++, this notion is directly related to **threads**, which allow exploiting the **multiple cores** of modern processors. Understanding threads is essential for writing high-performance programs, but also safe and correct ones.

5.1 Concept of a thread

A **thread** is an independent **execution thread** within the same program.

- A classic program has **a single thread** (sequential execution).
- A multithreaded program has **multiple threads**, executed in parallel or near-parallel.

All the threads of the same program:

- share the **same memory space** (heap, global variables),
- each has its own **execution stack** (local variables, function calls).

(Petit 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`: a class that represents a thread of execution and allows launching a function in a separate thread; defined in `<thread>`.)
Example simple:

```cpp
#include <iostream>
#include <thread>

void task() {
    std::cout << "Hello from a thread" << std::endl;
}

int main() {
    std::thread t(task); // thread creation
    t.join();            // wait for the thread to finish
    return 0;
}
```

Important 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 leads to `std::terminate()` 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 performing a task visible over time.

```cpp
#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;
}
```

(Note: `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 at the same time**,
- their outputs are **interleaved**,
- the order is **not deterministic**.

5.4 Passing arguments 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 format for argument passing.

```
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 several 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,
- non-deterministic 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` : 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 unlocking.

5.8 Atomic variables

For simple operations, one can use `std::atomic`.

```cpp
#include <atomic>

std::atomic<int> counter(0);

void increment() {
    counter++;
}
```

Advantages :

- faster than a mutex,
- safe for elementary operations.

Limitation :

- 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.

Best practice :

- use a number of threads close to the number of cores,
- favor coarse-grained tasks over very fine-grained ones.

6 Generic Programming, Template

The **generic programming** allows writing code **type-independent**, while preserving the **performance of compiled C++**. In C++, this paradigm relies primarily on the **templates**, which allow 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 model 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 the code.

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}
```

This function works for any type supporting 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 heavily 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;
    }
};
```

The specialization allows adapting the behavior without modifying the generic code.

6.2 Compilation principles: static duck typing, instantiation, and header files

The compilation of templates in C++ follows specific rules, different from those of regular code. Understanding these principles is essential to interpret compiler error messages and organize your code correctly.

Static duck typing

Templates rely on a principle called **duck typing static**.

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 during its definition. This is a key feature of templates:

- the generic code can be syntactically correct,
- but invalid for certain 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.

Instantiation of templates

A template is **not compiled until it is used**. Actual compilation happens during the **instantiation**, that is, 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 the organization of files.

Templates and header files (`.hpp`)

Unlike regular functions and classes, **the body of templates must be visible wherever they are used**. That is why:

- templates are defined **in header files** (`.hpp`),
- they are typically **not separated** into `.hpp` / `.cpp`.

Correct example :

```cpp
// vec.hpp
#pragma once

template <typename T>
T add(T a, T b) {
    return a + b;
}
```

```cpp
// 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 would not be able to generate the specialized versions, because the implementation would not be visible at instantiation time.

Why templates cannot be compiled separately

In ordinary 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 therefore cannot produce in advance 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 exist advanced techniques (explicit instantiation) allowing partial separation of the implementation, but they remain complex; in practice, the simple rule is:

Every template must be fully defined in a header file.

Summary of key principles

- Templates use a **static duck typing**: the constraints on types are implicit.
- Errors are detected **at instantiation**, not at definition.
- Each combination of template parameters generates 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 `constexpr` expressions allow moving part of the program's logic to the compiler. The result is code **faster at runtime**, because some decisions and some calculations are already resolved.

General principle

The central idea is the following:

> use the compiler as a **calculation engine**.

Values produced by metaprogramming:

- are known at compile time,
- incur **no runtime calculation cost** at execution,
- can be used as **template parameters**, array sizes, or constants.

Metaprogramming with integral template parameters

Untyped 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>()]; // size known at compile time
std::cout « a « std::endl; std::cout « sizeof(buffer) / sizeof(float) « std::endl; }
Here:

- ’`static_square<5>()` is computed by the compiler,
- no multiplication is performed at run-time.

`constexpr` : calculations 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:

- verifies that the expression can be evaluated statically,
- generates a constant if that's the case.

Comparison with a classic function:

```cpp
int runtime_square(int N)
{
    return N * N;
}
```

Usage in a template parameter:

```cpp
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.

```cpp
constexpr int factorial(int N)
{
    return (N <= 1) ? 1 : N * factorial(N - 1);
}
```

Usage as a template parameter:

```cpp
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**.

```cpp
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 compile time
```

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 << "Not integer" << std::endl;
}

Note: `std::is_integral_v` is provided by the header `<type_traits>`.
```

The non-relevant branch is **removed at compile time**.

Limits and precautions

- Increases the **compile time**.
- It can make errors more difficult 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 the code **both generic and readable**. In C++, the compiler is capable of automatically **deducing template parameters** in many cases, from the arguments provided at call time. Understanding when this deduction works — and when it fails — is essential for writing efficient 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.

Limits 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 automatically deduced**,
- 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**.

Expose template parameters in the types

One solution is to expose explicitly 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:

```cpp
float p = dot(v0, v1); // types and size inferred automatically
~~~

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 know 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 allows simplifying 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 generality.

6.5 Template specialization

Template specialization allows adapting the behavior of a generic template to a **particular case**, without modifying the general implementation. It is used when, for a type or parameter in particular, the default behavior is not appropriate, inefficient, or incorrect.

Specialization is a mechanism **resolved at compile time**, and is 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<<`.

Complete specialization of a template

A **complete specialization** replaces entirely the template's 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 v) { std::cout « (v ? “true” : “false”) « std::endl; }

Here too, the specialized version is used when `T = bool`.

Partial specialization (class templates)

The **partial specialization** allows you to specialize a template for **a family of types**, but it is only
     allowed **for class templates**, not for functions.

Example: specialization according to an integer parameter.

```cpp
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 specialization (or complete)

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 enables, for example:

- to optimize a particular case (the current dimension),
- to define a different behavior for a given size,
- or to adapt an internal representation.

Example: generic fixed-size 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 special handling 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 **full specialization**.

The choice is made **at compile time**, with no runtime test.

Full specialization for a specific type and size

It is also possible to specialize for **a specific type and size**.

```cpp
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

- **Full specialization** All template parameters are fixed (`vec<float,3>`). → a unique case, behavior completely redefined.
- **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 these are two distinct mechanisms that come into play **at different times** during compilation. Understanding their **order of precedence** 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 template version to instantiate.

Step 1: overload resolution (overloading)

When several functions have the same name, the compiler starts by applying the classic overload rules:

- exact type matches,
- 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 tries 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 non-template function matches.

Step 3: template specialization

Once a **template has been chosen**, the compiler searches for whether there exists a **more specific specialization** for the deduced parameters.

```
template <typename T>
void display(T x)
{
    std::cout << "template generique\n";
}

template <>
void display<bool>(bool x)
{
    std::cout << "specialisation bool\n";
}
```

Calls :

```
display(5);     // template générique
display(true);  // spécialisation bool
```

Result :

```
template generique
specialisation bool
```

The specialization **does not participate in overload resolution**. It is selected **after** the generic template has been chosen.

Subtle case: specialization vs overloading

Now consider :

```
template <typename T>
void display(T x)
{
    std::cout << "template generique\n";
}

template <>
void display<int>(int x)
{
    std::cout << "specialisation int\n";
}

void display(int x)
{
    std::cout << "fonction normale int\n";
}
```

Call :

```
display(3);
```

Result :

```
fonction normale int
```

Explanation :

1. the compiler sees a non-template function `display(int)` → **priority**,

2. the template is not even considered,
3. the specialization of the template is ignored.

A specialization can never beat a non-template overload.

Why this behavior?

Because :

- overload resolution is a **syntactic and local** decision,
- specialization is an **internal to the template** decision,
- mixing the two levels would make compilation ambiguous.

C++ thus imposes a strict hierarchy.

Priority summary (exact order)

When calling a function :

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 implementation of the template.

Best practices

- Use the **overloading** to offer different interfaces.
- Use the **specialization** to tailor internal behavior to 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 the **type deduction**, the **writing of generic functions** and the **readability of interfaces**.

In C++, there are two equivalent mechanisms:

- `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.

Aliases in 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 **self-descriptive** and facilitate its use in generic code.

Using aliases in template functions

Thanks to aliases, we 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**.

Aliases and dependent types (`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 know whether `value_type` is a type or a static member.

Alias templates (templated aliases)

The aliases themselves can be **templates**, which helps simplify 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 consistency of generic interfaces

Aliases are widely used in the STL:

- `value_type`,
- `iterator`,
- `reference`,
- `const_reference`.

Adhering to these conventions makes 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 Hardware view

This chapter offers a **simplified yet coherent view of the hardware** underlying the execution of a C/C++ program. The objective is not to delve into fine electronics, but to understand **how the code is physically executed**, and why certain notions (memory, cache, alignment, performance) are crucial in computer graphics and scientific computing.

7.1 Transistor principle

The **transistor** is the fundamental element of any modern electronic circuit. A processor today contains **billions of transistors**, each behaving as an **electronically controllable switch**.

Fundamental role

A transistor can be seen as :

- an **open or closed switch**,
- controlled by an electrical signal.

Classically, we associate:

- off state → `0`
- on state → `1`

These two states allow representing the **binary information**.

Physical principle of the transistor

Physical principle of the transistor

The **transistor** is above all a **physical object**, whose operation rests on the electrical properties of matter at the microscopic scale. Understanding its physical principle helps grasp how a continuous phenomenon (voltages, electric fields, electrons) is exploited to produce a **discrete logic** (`0` / `1`).

Silicon and electrical conduction

The central material of modern electronics is the **silicon**, a crystal whose electrons are bound to atoms by covalent bonds. In its pure state:

- silicon conducts current very poorly,
- it does not behave neither as a perfect insulator nor as a good conductor.

Its conductivity can however be **controlled** thanks to **doping**.

Doping and charge carriers

Doping consists of introducing a very small amount of foreign atoms into the silicon crystal.

- **N-type doping** Atoms with an excess electron → appearance of free electrons (negative charges)

- **P-type doping** Atoms with a missing electron → appearance of holes (effective positive charges)

These charge carriers are mobile under the effect of an electric field, which allows the passage of current.

PN junction and current control

When a P-doped region is brought into contact with an N-doped region, a **PN junction** forms.
At the interface :

- electrons and holes recombine,
- a charge-depleted zone appears,
- this zone creates a **potential barrier**.

According to the applied voltage :

- the barrier is lowered → current allowed,
- the barrier is strengthened → current blocked.

This is the first physical building block of the **electrical control**.

The MOSFET transistor: electric field rather than current

Modern processors use almost exclusively **MOSFET transistors** – Metal Oxide Semiconductor Field Effect Transistor.

- **Metal-Oxide-Semiconductor (MOS)**: Describes the physical structure (an insulated metal gate from the semiconductor by an oxide layer).
- **Field-Effect (Effet de Champ)**: Describes the control principle. It is an electric field (created by a voltage) that controls the passage of current, and not a current (unlike the bipolar transistor).

Unlike older transistors, they are controlled by an electric field (a voltage), and not by a current, which significantly reduces their power consumption.
A MOSFET consists of four main terminals:

1. **Source (S):** The entry of electrons.
2. **Drain (D):** The exit of electrons.
3. **Gate (G):** The control electrode.
4. **Substrate (Body):** The body of the transistor.

Key Innovation: The Gate is electrically isolated from the channel by an extremely thin **oxide** layer.

3. The Faucet Analogy To understand how it works, visualize a water faucet:

MOSFET Component	Faucet Analogy	Role
Source	Water inlet	Provides the current.
Drain	Water outlet	Receives the current.
Gate	Handle	Controls the flow without touching the water.
Voltage	Force on the handle	The opening command.

4. The Switching Physics: Threshold Voltage () The passage of current is not instantaneous. It relies on a phenomenon called **inversion**.

- **At rest ():** The transistor is a barrier. No current flows between Source and Drain.
- **Under voltage ():** The positive voltage on the gate acts like a magnet. It attracts minority electrons from the substrate toward the surface, just under the oxide.
- **The Threshold ():** When the voltage exceeds a critical value called **Threshold Voltage** (), the electron concentration is sufficient to form a "bridge" conductive between the Source and the Drain: the **channel** is created.

5. Operating Regimes (Simplified Mathematics) The drain current (I_D) behavior follows three regimes depending on the applied voltages:

1. **Cut-off regime (Blocking):**

-
- The faucet is closed. .
- *Logical interpretation:* **State 0**.

2. **Linear (Ohmic) regime:**

- The channel is open, but the Drain-Source potential difference is small. The transistor acts as a simple resistor.
- I_D is proportional to the applied voltage.

3. **Saturation regime:**

- The channel is fully conducting and the current is maximal and constant for a given gate voltage.
- This is the regime used for amplification, or the "fully on" state in digital logic.
- *Logical interpretation:* **State 1**.

6. From the Physical Phenomenon to the Logical Bit In computing, these complex behaviors are abstracted to keep only two stable states:

- **Low voltage ($ < V_{th}$):** Blocked transistor Bit `0`
- **High voltage ($ > V_{th}$):** Saturated transistor Bit `1`

However, at the current nanometer scale (transistors of a few nanometers), physical constraints reappear:

- **Leakage currents:** Even blocked, the transistor lets through a tiny current (tunneling effect), which heats the processor.
- **Thermal dissipation:** This is the main limit to increasing clock frequencies (GHz).

Nanometer-scale and physical constraints

Current transistors measure a few **nanometers**. At this scale:

- quantum effects become significant,
- leakage currents appear,
- thermal dissipation becomes critical.

These constraints explain:

- the frequency limits of processors,
- the necessity of parallelism,
- the importance of energy efficiency.

From transistor to logic

By combining several transistors, we build:

- **logic gates** (AND, OR, NOT, XOR),
- then more complex circuits :
  - adders,
  - multiplexers,
  - registers,
  - calculation units.

Conceptual example :

- an integer addition is performed by a **cascade of logic gates**,
- each gate is itself made of transistors.

Thus, any C++ instruction (addition, comparison, conditional jump) ultimately translates to **transistor switching**.

7.2 Basic structure of memory and arithmetic operations

Principles of memory

Storing a memory element relies on a very precise organization of transistors, different depending on the type of memory. Here is a progressive explanation, starting from the **bit** up to memories used in a CPU.

7.3 Minimal organization: storing a bit

A bit must:

- represent `0` or `1`,
- remain stable over time,
- be readable and writable.

There are several ways to store information using transistors:

1. Static memory (SRAM)

The **SRAM** (Static Random Access Memory) is used for:

- the processor's **registers**,
- the **L1, L2 and L3 caches**.

Typical access time: 0.3 to 2 ns
It is:

- extremely fast,
- **non-refreshing** (as long as it is powered),
- very stable,
- but **area-intensive**, as each bit uses several transistors.

General principle

An SRAM bit is stored using a **bistable electronic latch**, implemented with transistors.
Classical organization:

- **6 transistors per bit** (6T cell) :
  - 4 transistors form two cross-coupled inverters,
  - 2 transistors serve for read/write access.

Operation:

- the inverters hold each other in a stable state,
- the state corresponds to `0` or `1`,
- as long as power is present, the state is preserved.

Read:

- non-destructive,
- very fast.

Write:

- temporarily forces a state onto the latch.

Thus SRAM stores information in the form of an **active electrical equilibrium** between transistors.

2. Dynamic RAM (DRAM)

Dynamic RAM (DRAM) constitutes the **central memory** of a computer (RAM).

Typical access time: 50 to 100 ns

It is:

- slower than SRAM,
- **volatile**,
- requires a **periodic refresh**,
- much denser (fewer transistors per bit).

General principle

A DRAM bit is stored as an **electrical charge**.

Classic organization:

- **1 transistor + 1 capacitor per bit** (1T1C cell).

Operation:

- the capacitor stores a charge (`1`) or is empty (`0`),
- the transistor controls access to the cell.

Read:

- the charge is measured,
- the read is **destructive** (the capacitor discharges),
- the value must be rewritten immediately.

Refresh:

- charges leak naturally,
- each cell must be read and rewritten periodically (about every 64 ms).

DRAM stores information in the form of a **passive charge**, hence the need for refresh.

3. Flash memory

Flash memory is a **non-volatile** memory used for:

- SSDs,
- USB flash drives,
- memory cards,
- firmware storage (BIOS, microcontrollers).

It is:

- persistent without power,
- slower to write than RAM,
- limited in the number of write cycles,
- very dense.

Typical access time: 50 to 100 µs (microseconds) for reads, 200 µs to a few ms for writes

General principle

A flash memory bit is stored using a **floating-gate transistor**.

Organization:

- the cell is a modified MOS transistor,
- it has an electrically isolated **floating gate**.

Operation:

- writing involves **injecting electrons** into the floating gate using a high voltage,
- the electrons remain **trapped** in the insulating material,
- the presence or absence of charge changes the transistor's behavior.

Read:

- non-destructive,
- based on measuring the conduction threshold.

Erasing:

- is performed in whole blocks,
- also requires high voltages.

Flash memory stores information as physically trapped charges, which explains its persistence without power.

Concise comparison

Memory Type	Volatile	Transistors / bit	Refresh	Main Use
SRAM	yes	~6	no	registers, caches
DRAM	yes	1 + 1 capacitor	yes	central memory
Flash	no	1 (specific)	no	persistent storage

Principles of arithmetic operations

From C++ code to machine instruction

An arithmetic operation written in C++ is an **abstract expression**:

```
c = a + b;
```

For the processor, this corresponds to a well-defined sequence:

1. load `a` and `b` from memory into **registers**,
2. activate the arithmetic unit with the requested operation,
3. produce a binary result,
4. store the result in a register or in memory.

The processor never 'understands' variables or C++ types: it manipulates only **registers**, **opcodes**, and **bits**.

Central role of the ALU

Arithmetic and logical operations are performed by the **ALU** (*Arithmetic Logic Unit*).
Main functions of the ALU:

- addition and subtraction,
- logical operations (AND, OR, XOR),
- comparisons,
- binary shifts.

All these operations rely on:

- **combinational circuits**,
- composed of logic gates,
- themselves built from transistors.

The ALU receives:

- two operands from the registers,
- an opcode indicating the operation to perform,
- and produces a result as well as **status flags**.

Subtraction, comparisons and internal logic

In the ALU:

- the **subtraction** is implemented as a modified addition,
- the **comparisons** exploit the result of an internal subtraction,
- relational operators (`<`, `>`, `==`) produce only a single logic bit.

Conceptual example:

```
if (a < b) { ... }
```

In hardware:

- the processor computes `a - b`,
- observes the sign or the carry flag,
- and deduces the result of the test.

Multiplication and division: composite operations

Unlike addition, multiplication and division:

- require several internal steps,
- deploy more complex circuits,
- and are therefore more costly in cycles.

Multiplication relies on:

- shifts,
- partial additions,
- or highly optimized specialized units.

Division:

- is generally iterative,
- and constitutes one of the slowest arithmetic operations.

Operations on floating-point numbers

Floating-point calculations are handled by a separate unit: the **FPU**.
It performs:

- alignment of exponents,
- operation on the mantissas,
- normalization of the result,
- rounding according to the IEEE standard.

These operations are more costly than those on integers, but fully managed by hardware.

SIMD vector instructions

Modern processors have **vector units** capable of applying the same operation on multiple data elements simultaneously.
Principle:

- one instruction,
- several operands processed in parallel.

It is a direct extension of the basic arithmetic operations, used for:

- computer graphics,
- signal processing,
- scientific computing.

Scheduling and pipelining

Arithmetic operations are not executed in isolation:

- they are **pipelined**,
- reordered,
- executed in parallel when possible.

Thus:

- several additions may be in progress simultaneously,
- as long as data dependencies are respected.

Real cost of an operation

In a real program:

- the **memory access time** is often dominant,
- pure arithmetic computation is rarely the bottleneck.

Optimizing performance often comes down to:

- reducing memory accesses,
- improving data locality,
- exploiting parallelism.

Key idea to remember

Arithmetic operations are **elementary hardware building blocks**, orchestrated by the processor via the ALU, the FPU and vector units. C++ code expresses logical calculations, but their execution relies on scheduling, parallelism, and efficient data access.

7.4 Cache memory concept

Fundamental problem: memory latency

Accessing main memory (RAM) is:

- **much slower** than accessing registers or compute units.

Order of magnitude:

- register: ~1 cycle
- L1 cache: ~3–5 cycles
- L2 cache: ~10 cycles
- RAM: ~100–300 cycles

Without intermediate mechanisms, the CPU would spend its time **waiting for memory**.

Cache principle

The **memory cache** is an intermediate memory:

- smaller than RAM,
- much faster,
- integrated into the processor.

It stores **copies of blocks of memory recently used**.

Cache hierarchy

We typically distinguish:

- **L1**: very small, very fast, per core,
- **L2**: larger, somewhat slower,
- **L3**: shared between cores.

Each level acts as a buffer toward the lower level.

Temporal and spatial locality

The cache relies on two fundamental principles:

- **Temporal locality** A data item used recently is likely to be reused.

- **Spatial locality** If you access a memory address, neighboring addresses are likely to be used.

That's why:

- contiguous arrays,
- `std::vector`,
- sequential traversals,

are **much faster** than random accesses.

Link to C++ programming

Examples of cache-friendly code:

```
for(int i = 0; i < N; ++i)
    sum += array[i];
```

Unfavorable examples:

```
for(int i = 0; i < N; ++i)
    sum += array[random_index[i]];
```

In computer graphics, this point is **crucial**:

- vertex processing,
- particle simulation,
- traversal of GPU/CPU buffers.

8 Development Methodologies and Best Practices

This chapter presents the **fundamental methodological principles** enabling the production of C++ code:

- readable,
- robust,
- testable,
- maintainable,

all while respecting the language's performance and low-level constraints.

These principles apply just as well to small programs as to complex projects (simulation, graphics engine, parallel computing).

8.1 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 validated automatically.
- **Extensibility**: future modifications are possible without massive rewrites.
- **Measured performance**: optimization guided by measurements, not by intuition.

Note that when working with others, **code readability** should be the priority. Readable code:

- facilitates code reading and code reviews,
- reduces errors during modifications,
- speeds up the onboarding of new contributors,
- allows reasoning and testing more easily.

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 evident.

Best practices for readability: explicit names, short functions, comments when the code is not self-documenting, consistent formatting, and systematic code reviews.

8.2 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) :

```cpp
// Version condensée et moins lisible : logique imbriquée, calcul d'index
// difficile à suivre, tout est condensé sur quelques lignes.
int count_neighbors_ugly(const std::vector<int>& grid, size_t w, size_t h,
                         size_t x, size_t y)
{
    int c = 0;
    // balayer un rectangle 3x3 centré sur (x,y) en jouant sur les bornes
    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; // logique subtle
    }
    return c;
}

// Version claire et simple : fonctions auxiliaires et boucles explicites
```

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& g, size_t w, size_t x, size_t y) { return g[y * w + x]; }

int count_neighbors(const std::vector& 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*

A piece of logic should exist in only one place.

Note:
eliminating any duplication can lead to unnecessary abstractions.
A local and simple duplication is sometimes preferable to a complex generalization.

Example (DRY) :

```cpp
// Duplication (worse) : two very similar functions
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();
}

// Refactoring (DRY) : a generic implementation avoids 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); // works for int
// double a2 = average(vd); // works for double
```

YAGNI – *You Aren't Gonna Need It*

Do not implement features "just in case" if they are not necessary.

This principle is particularly important in C++, where: - templates, - generics, - and metaprogramming can encourage excessive complexity too early.

Example (YAGNI) :

```cpp
// Prematurely generalized (YAGNI)
template <typename T = float, int N = 3>
struct vec { T data[N]; };

// Simple and sufficient version for everyday use
struct vec3 { float x, y, z; };
```

8.3 Invariants, assertions et contrat de fonction

A robust program does not settle for "working in the normal cases": it **explicitly expresses its assumptions** and verifies that they are respected.

These assumptions constitute what is called the code's **contract**.

Why talk about a contract?

When a function is called, two viewpoints exist:

- **the caller's perspective** "What am I allowed to pass to this function?"
- **the function's perspective** "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,
- hard-to-diagnose bugs.

The contract allows us to formalize these rules. The set of these rules is what we call contract-based programming.

The three key notions of the contract

We distinguish three types of complementary rules.

1. Preconditions

A **Precondition** is a condition that **must be true before the call** of 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 an interval,
- 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 the contract of a stack.

```
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.

Runtime assertions (`assert`)

Assertions allow verifying these rules **during execution**, mainly in the development phase.
In C++, we use `assert` to detect **programming errors**.

```cpp
#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 logical errors,
- stop the program **at the exact point of the problem** during debugging.

They are therefore a **development tool**, not a user-facing error-handling mechanism.

Best practices with `assert`

- use `assert` for **programming errors**. The `asserts` are theoretically "useless" for the proper functioning of the program; they only serve to facilitate programming by detecting unexpected/unplanned 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 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 depend** on assertions to function correctly.

Compile-time Assertions (`static_assert`)

Some rules can be verified **before even executing**, at compile time.
That is 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 assumptions impossible to verify at runtime.

General rule: **prefer compile-time checks when possible**.

Complete example: stack with invariant and assertions

```cpp
#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 is full");

        data.push_back(v);

        // postcondition
        assert(data.back() == v && "push: top is incorrect");
    }

    int pop()
    {
        // precondition
        assert(!data.empty() && "pop: stack is empty");

        int v = data.back();
        data.pop_back();

        // invariant always holds
        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` verifies the contract at runtime (debug).
- `static_assert` verifies the contract at compile time.
- Used correctly, they make the code:
  - safer,

– more readable,
– and easier to maintain.

Alternatives to asserts

The function `assert` remains quite limited in terms of functionality. Alternative tools can help express and verify contracts in a more readable, safe, and maintainable way for large-scale code:

- **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 (contracts that can be activated/deactivated, centralized diagnostics).
- **Lightweight annotations (Expects/Ensures)**: define wrappers `Expects(condition)` to standardize messages and enable different behaviors depending on configuration (throw, abort, log).
- **Supplementary tools**: sanitizers (ASan/UBSan/TSan) and static analysis (clang-tidy, cppcheck) detect classes of errors that assertions alone do not cover.

8.4 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 later modifications. The **tests** allow automatically verifying 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 allow you to:

- detect an error before the end user,
- avoid regressions during modification 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 at every change (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**: it is easy to understand 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** checks 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** checks the interaction between several 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 of the tested code,
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 maintenance 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.

Minimal test tool (without a framework)

We can write tests with `assert`, but it is often useful to have more explicit messages, especially for floating points.

```
#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);
    }
}
```

8.5 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);

    // violation of precondition (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 falls under the **contract**: its violation is a programming error.

8.6 Test-Driven Development (TDD)

The **TDD** is a methodology in which the code is written **in response to tests**. It aims to transform 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 make the test pass,
3. **Refactor**: improve the code without breaking the tests.

This loop is repeated frequently.

Benefits of TDD

The TDD:

- forces you to clarify the API from the start,
- encourages short and testable functions,
- limits over-engineering (YAGNI),
- makes refactorings much safer.

8.7 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: `norm(v) > 0`.

Step 1: test (Red)

```cpp
#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)

```cpp
#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, you can:

- factorize `norm2`,
- improve performance,
- add non-regression tests.

Conclusion on tests and TDD

Tests constitute an **automatic verification of a function's contract**. TDD provides a simple methodology for writing code:

> define the behavior -> verify it automatically -> improve the implementation with confidence.

Used correctly, tests make code more reliable, more readable, and easier to evolve.

Testing invalid cases

Testing only valid cases is insufficient: robust code must also **detect invalid usages correctly**. It is therefore essential to write tests that verify that:

- the **violated preconditions** are properly detected (assertion, exception, error returned),
- the **invalid inputs** do not produce silently incorrect results,
- the program **fails in a controlled and explicit manner**, rather than continuing in an incoherent state.

These negative tests help ensure that the code's contract is actually respected, and not merely in ideal cases. They are particularly important during refactorings: an internal change must never turn a detected error into silent behavior.

According to the chosen error-handling policy, a test may 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.

Very well. Here is **a concrete, pedagogical example**, **and consistent with your chapter**, showing **how to test an invalid case**. You can insert it **right after the paragraph "Test invalid cases as well"**.

I propose **two complementary variants**:

1. invalid case detected by `assert` (programming error),
2. invalid case detected by error return (usage error).

Example: testing an invalid case detected by `assert`

We take back the function `normalize(v)` seen previously. Its **precondition** is that the vector must not be 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**.

```cpp
// 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 activated separately or checked manually,
- its role is to explicitly document the **expected behavior in case of invalid usage**.

Example: testing an invalid case with explicit error handling

If you want to handle invalid inputs without crashing the program, you can use a result type.

```cpp
#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};
}
```

Corresponding test:

```cpp
#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 verifies 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, you generally need to cover:

- the nominal cases,
- the edge cases,
- invalid inputs,
- and sometimes many parameter combinations.

Moreover, when code evolves (refactoring, API changes, adding parameters), tests must be **updated** in order to remain consistent with the new contract. This maintenance phase can account for a substantial 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 generating systematic unit tests from a clear specification,
- proposing edge or negative tests often overlooked,
- helping to adapt or regenerate tests after a code modification,
- automatically exploring different input combinations.

8.8 Error handling: principles and methodology

A robust program does not merely detect errors: it must **classify them**, **signal them correctly**, and **allow the caller to react** appropriately.

Error handling is an integral part of the **design** of the code and of its **API**.

Why explicit error handling?

Without a clear error handling strategy, one obtains:

- silent errors,
- undefined behaviors,
- inconsistent internal states,
- bugs that are difficult to reproduce.

Good error handling makes it possible:

- to make failures **visible and understandable**,
- to separate the nominal code from error handling code,
- to explicitly test invalid behaviors,
- to strengthen the contract between the caller and the 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 occur** 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 program termination.

```
assert(index < data.size() && "index hors limites");
```

These errors are generally **not recoverable**.

2. Usage or environment errors

These are **foreseeable** situations, even if the code is correct.
Examples:

- missing file,
- malformed data,

- invalid user input,
- unavailable hardware resource.

These errors must be **reported to the caller**.
Recommended handling:

- exceptions,
- return codes,
- result types (`optional`, `expected`, `Result`).

Error handling strategies in C++

The choice of strategy depends:

- on the type of error,
- on the context (library, application, real-time),
- on performance and readability constraints.

1. Exceptions

Exceptions allow you to **clearly separate the nominal code from the error-handling code**.

```
float parse_float(std::string const& s)
{
    return std::stof(s); // may throw an exception
}
```

Advantages :

- readable, straightforward code,
- automatic propagation of the error,
- well-suited for rare errors.

Disadvantages :

- potential cost (depending on context),
- less explicit flow control,
- sometimes forbidden at low-level / real-time.

To be used with discipline, and clearly documented.

2. Return codes

Historical and explicit approach.

```
bool read_file(std::string const& name, Data& out);
```

Advantages :

- simple,
- no exceptions,
- explicit control.

Disadvantages :

- easy to forget to check,
- not very expressive without an associated structure.

3. Result types (`optional`, `expected`, `Result`)

A 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,
- highly testable.

Often the best compromise for modern APIs.

Full example: Robust API with a 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)}; }
};
```

Reading a file containing a floating point number per line :

```
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, "Unable to open the file"});
    }

    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, "Parsing error", 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** handle recoverable errors,
- the **negative tests** verify that errors are properly detected,
- the **contract** documents what falls under one or the other.

Here is an **enriched and pedagogical** version of your section **Best practices for API design**, with **concrete "bad / good" examples** for each principle, while remaining coherent with the rest of `08-methodology.md`.
You can **replace your current section entirely with this one**.

8.9 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 may 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 is made up 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 should be:

- **clear**: hard to misuse,
- **predictable**: consistent behaviors in similar situations,
- **documented by the type**: the types express constraints,
- **testable**: easy to use in unit tests,
- **stable**: changes do not break existing code unnecessarily.

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 that it is invalid,
- the behavior in case of an error 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 intention is clear,
- errors are detected at compile time.

Another example: ambiguous bool vs dedicated type

```
void draw(bool wireframe); // what does true mean?
```

Better design:

```cpp
enum class RenderMode { Solid, Wireframe };
void draw(RenderMode mode);
```

Limit invalid states

A good API makes invalid states **impossible or difficult to represent**.

Problematic example: partially valid state

```cpp
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

```cpp
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 inconsistent state.

Separate interface and implementation

The API should expose **what the code does**, not **how it does it**.

Header (`.hpp`) : interface

```cpp
// image.hpp
class Image {
public:
    Image(int w, int h);
    void clear();
    void save(const std::string& filename) const;
};
```

Source (`.cpp`) : implementation

```cpp
// image.cpp
#include "image.hpp"

void Image::clear()
{
    // internal details invisible to the user
}
```

Advantages:

- freedom to change the implementation,
- faster compilation,
- more stable API.

Avoid hidden side effects

A function should not modify global states in an unexpected way.

Bad example

```cpp
void render()
{
    global_state.counter++; // hidden side effect
}
```

Better example

```cpp
void render(RenderContext& ctx)
{
    ctx.counter++;
}
```

Dependencies are explicit and testable.

Practical API design rules

- clearly document the **preconditions** and **postconditions**,
- make errors visible in the type or behavior,
- avoid ambiguous parameters (`bool`, undocumented `int`),
- prefer small, orthogonal functions,
- test the API as if we 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 is executed.

It guides the user toward proper usage, makes errors explicit, and facilitates testing, maintenance, and evolution of the code.