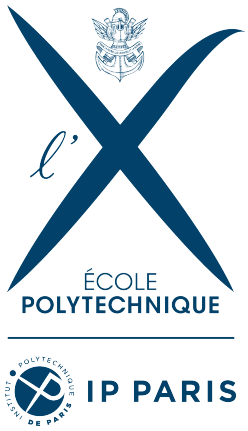

Programmation C++

Application à l'Informatique Graphique

[CSC-43043-EP]

2026

Damien ROHMER - damien.rohmer@polytechnique.edu

Contents

1 Introduction au C++

1.1 Préambule

Le langage C++, créé au début des années 1980 par le chercheur Bjarne Stroustrup chez Bell Labs, est introduit initialement comme une extension du langage C avec lequel il est intrinsèquement lié. Le langage C est un langage dit de "bas niveau", en étant proche du matériel (processeur, mémoire) particulièrement adapté pour coder des applications efficaces liées au système d'exploitation. Le langage C++ a été introduit pour préserver les possibilités du langage C, tout en l'étendant à des mécanismes de structuration et d'abstraction pour la description de logiciels de grande envergure.

Le C++ se distingue des autres langages de programmation par sa capacité unique à combiner **performance bas niveau** et **abstraction de haut niveau**. Héritier direct du C, il permet un contrôle précis de la mémoire et du matériel, indispensable dans les domaines où l'efficacité est critique (systèmes embarqués, calcul scientifique, moteurs de jeu, etc.). Contrairement à des langages comme Python ou Java, qui reposent sur une machine virtuelle ou un interpréteur qui ajoute une étape d'indirection lors de l'exécution, le C++ est un langage compilé qui produit du code machine optimisé directement lu et exécuté par le processeur, garantissant ainsi une exécution très rapide.

Une autre spécificité majeure du C++ est son support simultané de plusieurs manières de programmer, appelées **paradigmes de programmation** :
- **Procédural**, hérité du C, pour une approche classique fondée sur des fonctions et des structures de contrôle.
- **Orienté objet**, introduit avec les classes, l'encapsulation, l'héritage et le polymorphisme, facilitant la conception modulaire de logiciels complexes.
- **Générique**, grâce aux **templates** (généricité paramétrée par type), qui permettent d'écrire du code réutilisable et indépendant des types.
- **Fonctionnel**, de plus en plus présent depuis C++11 avec les **lambdas** (fonctions anonymes) et les algorithmes de la bibliothèque standard.

Ce mélange de paradigmes fait aujourd'hui du C++ un langage reconnu comme extrêmement flexible, capable de s'adapter à une grande variété de contextes. Il reste incontournable pour des domaines où la performance et la maîtrise fine de la mémoire sont essentielles, comme les moteurs de jeux, les logiciels embarqués, la simulation numérique, le calcul haute performance ou encore la finance.

Évolutions du C++

Le langage C++ continue à intégrer des évolutions régulières.

- **C++98** et **C++03** ont normalisé le langage et ses bibliothèques de base.
- **C++11**, appelé "C++ moderne", a marqué un tournant important avec, notamment, l'arrivée des boucles étendues, facilitées d'initialisation des structures, le mot-clé `auto` (déduction de type), l'apparition des pointeurs intelligents (ex. `std::unique_ptr`, `std::shared_ptr`) et des fonctions lambdas (fonctions anonymes).
- **C++14** et **C++17** ont enrichi la syntaxe et la bibliothèque standard (structured bindings, filesystem, parallélisme).
- **C++20** a apporté les principes de **concepts** (contraintes pour templates), les **coroutines** (fonctions dont l'exécution peut être suspendue et reprise) et les **ranges** (opérations sur séquences).
- **C++23** continue cette modernisation, en affinant les bibliothèques et en simplifiant l'usage du langage.

Pourquoi utiliser le C++ ?

Le C++ est actuellement l'un des langages indispensables lorsqu'il s'agit de concevoir des applications à fortes contraintes de performance, de temps réel ou de calcul intensif.

Domaines d'application

- **Applications scientifiques et temps réel** : simulations physiques, calculs numériques, systèmes embarqués.

- **Moteurs de jeux (Game Engines)** : Unity, Unreal Engine, Godot, ainsi que pratiquement tous les jeux AAA utilisent massivement le C++.
- **Logiciels 2D/3D** : Maya, Blender, Photoshop, Premiere Pro, Catia, SolidWorks reposent en grande partie sur du C++.
- **Calcul parallèle et GPU** : CUDA (NVIDIA) est basé sur le C++.
- **Frameworks de Deep Learning et Vision** : PyTorch, TensorFlow, OpenCV s'appuient sur des cœurs en C++ pour optimiser les performances.
- **Systèmes d'exploitation** : Windows est écrit en grande majorité en C et C++.
- **Web et services massifs** : navigateurs (Chrome, Firefox) et infrastructures critiques (AWS, Facebook, etc.) utilisent le C++ pour les parties cœur de performance.

Points forts (+)

- **Performance** : compilation directe en code machine, optimisations très fines possibles.
- **Robustesse** : un langage mûr, utilisé et testé à très grande échelle.
- **Haut et bas niveau** : rare combinaison qui permet aussi bien d'écrire du code proche du matériel que d'utiliser des abstractions modernes de haut niveau.
- **Spécificité** : cette dualité n'est présente que dans C++ (et plus récemment Rust).
- **Liberté de programmation** : support de multiples paradigmes (procédural, objet, générique, fonctionnel).
- **Compatibilité C** : possibilité de réutiliser l'immense écosystème du langage C.

Points faibles (−)

- **Complexité** : la richesse du langage et la multiplicité des paradigmes peuvent être difficiles à maîtriser.
- **Gestion mémoire** : la gestion mémoire manuelle est un facteur de complexité et d'erreurs de programmation importantes.
- **Chaîne de compilation** : la compilation est plus lourde et parfois plus lente que dans d'autres langages modernes.

Comparaison rapide avec d'autres langages

- **C++ vs Java**
  Tous deux sont orientés objet, mais leur philosophie diffère.
  - C++ est compilé en code machine natif, ce qui le rend très performant et adapté aux systèmes où chaque cycle de calcul compte.
  - Java est exécuté sur une machine virtuelle (JVM), ce qui facilite la portabilité mais ajoute une couche d'abstraction.
  - Java gère la mémoire automatiquement via un **garbage collector**, alors que C++ laisse au programmeur le contrôle fin de l'allocation et de la libération.
- **C++ vs Python**
  Python est réputé pour sa simplicité d'écriture et sa rapidité de développement, mais reste un langage interprété, donc bien plus lent en exécution.
  - C++ demande plus de rigueur et de syntaxe, mais permet d'atteindre des performances maximales.

  - En pratique, Python est souvent utilisé pour le prototypage, le scripting et l'analyse de données, tandis que C++ est privilégié pour les parties critiques en performance (moteurs 3D, calcul scientifique, simulations).

  - Les deux langages sont parfois utilisés ensemble : Python comme couche de haut niveau, C++ pour les modules de calcul.

- **C++ vs Rust**
  Rust est un langage plus récent (2010), conçu pour offrir la même efficacité que le C++ mais avec une gestion de la mémoire plus sûre.

  - Rust supprime toute possibilité de fuite ou d'accès illégal à la mémoire grâce à son système d'**emprunts et de possession**.

  - C++ offre plus de flexibilité et dispose d'un immense écosystème logiciel existant, mais au prix d'une rigueur nécessaire pour éviter erreurs et failles de sécurité.

  - Rust est perçu comme une alternative moderne et sécurisée, mais C++ reste aujourd'hui largement dominant en industrie et dans les bibliothèques disponibles.

1.2 Premier programme en C++

On considère le programme C++ suivant:

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

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

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

Explications ligne par ligne

1. `#include <iostream>`
   - Cette directive indique au compilateur d'inclure la bibliothèque standard **iostream**, qui permet d'utiliser les flux d'entrée et de sortie (`std::cin`, `std::cout`, etc.).
2. `int main()`
   - C'est la fonction principale du programme.
   - Tout programme C++ doit posséder une fonction `main`.
   - Son exécution commence toujours ici.
   - Le mot `int` indique que la fonction `main` renvoie un entier au système d'exploitation (0 en cas de succès, une autre valeur en cas d'erreur).
3. `std::cout << "Hello, world!" << std::endl;`
   - `std::cout` est le flux de sortie standard (en général l'écran).
   - L'opérateur `<<` permet d'envoyer des données dans le flux.
   - `"Hello, world!"` est une chaîne de caractères.
   - `std::endl` insère un saut de ligne et force l'affichage immédiat.

4. `return 0;`
    - Indique que le programme s'est bien terminé.
    - La valeur retournée est transmise au système.

Rem. Chaque instruction se termine par un point virgule ";" en C++. L'indentation et les sauts de lignes sont optionnels, ils sont utiles pour la lisibilité du programme mais ne changent pas sa structure.

Première compilation (sous Linux/MacOS)

Pour transformer le fichier source C++ (par exemple `hello.cpp`) en un exécutable, on utilise un **compilateur C++**. Sous Linux ou macOS, les compilateurs les plus courants sont :

- **g++** (GNU C++ Compiler, issu de GCC)
- **clang++** (compilateur C++ développé dans le cadre du projet LLVM)

Supposons que le fichier s'appelle `hello.cpp`. Tapez en ligne de commande dans le répertoire contenant le fichier `hello.cpp`

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

- `g++` : lance le compilateur C++.
- `hello.cpp` : fichier source à compiler.
- `-o hello` : option qui indique le nom de l'exécutable produit (`hello`).

L'execution du programme se réalise avec la commande

```
./hello
```

Ce qui doit afficher le résultat suivant

```
Hello, world!
```

1.3 Déclaration de variables

En C++, une **variable** est une zone de mémoire qui contient une valeur et qui est identifiée par un nom.
Chaque variable a un **type** qui définit la nature des valeurs qu'elle peut contenir (entiers, nombres à virgule, texte, etc.).

Exemple simple

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

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

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

    return 0;
}
```

Types fondamentaux

Vous utiliserez principalement deux types fondamentaux dans vos codes :

- **int** : nombre entier (integer). Sur nos machines, un `int` est encodé sur **4 octets**.

```
int entier = 325;
```

- **float** : nombre à virgule flottante, dit à "simple précision". Encodé sur **4 octets**.

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

Vous rencontrerez également les types suivants :

- **bool** : valeur booléenne (`true` ou `false`). Introduit par C++ (absent du C), il rend le code plus lisible qu'un entier.

```
bool estEtudiant = true;
```

- **double** : nombre à virgule flottante à "double précision", encodé sur **8 octets**.

```
double pi = 3.14159;
```

Par défaut, un nombre décimal sans suffixe est interprété comme un `double`.
> Dans notre contexte, on utilisera plus souvent des `float` pour rester compatibles avec la carte graphique.

- **char** : caractère (1 octet). La correspondance entre valeurs et caractères est donnée par la **table ASCII**.

```
char initiale = 'A';
```

Un `char` peut également être utilisé pour manipuler directement la mémoire au niveau de l'octet.

Remarques importantes

1. **Division entière vs division flottante**

   Lorsqu'on divise deux entiers, le résultat est **tronqué** (division euclidienne) :

```
int a = 5 / 2;  // vaut 2
int b = 5 % 2;  // vaut 1 (reste de la division)
```

   Pour obtenir un résultat décimal, il faut qu'au moins un des opérandes soit flottant :

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

2. **Le mot-clé `auto`**

   Il permet au compilateur de déduire automatiquement le type :

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

   [Attention] Pour des types simples, il est préférable d'indiquer explicitement le type pour plus de lisibilité. `auto` est surtout utile pour des fonctions génériques ou des types complexes.

3. **Variables non initialisées**

   En C++, les variables fondamentales ne sont **pas initialisées par défaut**.

```cpp
int a; // contient une valeur indéfinie
```

 Pour éviter les **comportements indéterminés**, il est conseillé d'initialiser systématiquement vos variables :

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

Déclaration sans initialisation (exemple)

```cpp
int compteur;    // non initialisé
compteur = 10;  // affectation d'une valeur plus tard
```

[Attention]: une variable non initialisée contient une valeur indéfinie et ne doit pas être utilisée avant affectation.

Variables constantes (`const`)

En C++, une variable peut être déclarée **constante** grâce au mot-clé `const`. Une telle variable doit être **initialisée au moment de sa déclaration** et **ne peut plus être modifiée** ensuite.

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

int main() {
    std::cout << "Pi = " << pi << std::endl;
    // pi = 3.14; // ERREUR : impossible de modifier une constante
    return 0;
}
```

Intérêt

- Garantit que la valeur ne sera pas modifiée accidentellement dans le code.
- Rend le programme **plus lisible** et **plus sûr**.
- Peut permettre au compilateur d'optimiser certaines expressions.

Conversion de types (cast)

En C++, il est fréquent de convertir une valeur d'un type vers un autre : on appelle cela un **cast** (conversion de type).

Exemples : conversions implicites et explicites

```cpp
int i = 3;
float f = i;              // conversion implicite : int -> float

double d = 3.9;
int j = (int)d;           // cast C-style : tronque la partie décimale (narrowing)
int k = static_cast<int>(d); // cast C++-style : recommandé car plus sécurisé
```

Bonnes pratiques :

- Préférez `static_cast<T>(expr)` pour les conversions entre types numériques et entre pointeurs compatibles.
- `(int)d` est la notation C-style de cast ; on peut également trouver `int(d)` qui est la forme fonctionnelle (function-style) de cast. Pour les types fondamentaux, les deux se comportent de façon équivalente (tronquent la partie décimale).
- Notez que les conversions peuvent réduire la précision ou la plage (narrowing) : `double -> int` tronque, un entier non signé peut déborder (overflow).
- Il existe également `reinterpret_cast<T>(expr)`, qui réinterprète la représentation binaire d'un objet comme un autre type. C'est une opération bas niveau, potentiellement dangereuse (risques d'alignement, d'aliasing

ou comportement indéfini) ; n'utilisez□la que pour de l'interopérabilité ou de la lecture/écriture binaire clairement documentés.

Cette notion est utile pour contrôler explicitement les conversions et éviter des comportements surprises lors des opérations arithmétiques ou des passages d'arguments.

1.4 Affichage et lecture formatés : printf, scanf

`printf` et `scanf` (hérités du C)

En plus de `std::cout` et `std::cin`, C++ conserve les fonctions classiques du langage C :

- **`printf`** (*print formatted*) : pour un affichage formaté.
- **`scanf`** (*scan formatted*) : pour une lecture formatée.

Elles sont définies dans l'en-tête `<cstdio>` (ou `<stdio.h>` en C). Leur usage repose sur des **spécificateurs de format** (`%d`, `%f`, `%s`, etc.) qui indiquent le type de la variable.

Exemple d'affichage formaté avec `printf`

```cpp
#include <cstdio>

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

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

Sortie :

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

- `%d` : entier (`int`)
- `%f` : flottant (`float` ou `double`)
- `%.2f` : flottant affiché avec deux décimales

Exemple de lecture avec `scanf`

```cpp
#include <cstdio>

int main() {
    int age;
    printf("Entrez votre age : ");
    scanf("%d", &age);   // & = adresse mémoire
    printf("Vous avez %d ans.\n", age);
    return 0;
}
```

Dans `scanf`, il est nécessaire de fournir **l'adresse** de la variable (ici `&age`), car la fonction modifie directement sa valeur.

Principaux spécificateurs de format (`printf` / `scanf`)

Spécificateur	Type attendu	Exemple d'utilisation	Résultat affiché
`%d`	entier signé (`int`)	`printf("%d", 42);`	`42`
`%u`	entier non signé (`unsigned`)	`printf("%u", 42u);`	`42`
`%f`	flottant (`float` ou `double`)	`printf("%f", 3.14);`	`3.140000`
`%.nf`	flottant avec `n` décimales	`printf("%.2f", 3.14159);`	`3.14`
`%e`	flottant en notation scientifique	`printf("%e", 12345.0);`	`1.234500e+04`

Spécificateur	Type attendu	Exemple d'utilisation	Résultat affiché
`%c`	caractère (`char`)	`printf("%c", 'A');`	`A`
`%s`	chaîne de caractères (`char*`)	`printf("%s", "Bonjour");`	`Bonjour`
`%x`	entier en hexadécimal (min.)	`printf("%x", 255);`	`ff`
`%X`	entier en hexadécimal (maj.)	`printf("%X", 255);`	`FF`
`%p`	adresse mémoire (pointeur)	`printf("%p", &a);`	`0x7ffee3c8a4`
`%%`	caractère `%` littéral	`printf("%%d");`	`%d`

1.5 Conteneurs d'éléments contigus, tableaux

En C++, la **librairie standard (STL, Standard Template Library)** définit plusieurs conteneurs permettant de stocker des ensembles de valeurs.
Parmi eux, deux structures sont particulièrement importantes :

- **`std::array<T, N>`** : tableau **statique** de taille fixe.
  - Les éléments sont stockés de manière contiguë en mémoire.
  - La taille `N` doit être connue **à la compilation** et ne peut pas changer.
  - Les données sont stockées dans la **pile (stack memory)** : plus efficace en vitesse d'accès, mais limitée en taille (typiquement quelques Mo).
- **`std::vector<T>`** : tableau **dynamique**.
  - Les éléments sont aussi stockés de manière contiguë en mémoire.
  - La taille peut être modifiée au cours de l'exécution (ajout/retrait d'éléments).
  - Les données sont stockées dans le **tas (heap memory)** : légèrement plus coûteux en allocation, mais permet d'accéder à toute la mémoire vive (RAM).
- **Tableaux C classiques** (`T var[N]`) :
  - Taille fixe, connue à la compilation.
  - Pas de vérification de bornes.
  - Pas de méthodes utilitaires (`size()`, `push_back`, etc.).
  - Peu utilisés en C++ moderne, sauf pour interagir avec du code C ou pour des besoins très bas-niveau.

Exemple simple avec `std::vector`

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

int main() {
    // Création d'un vecteur vide d'entiers
    std::vector<int> vec;

    // Ajout d'éléments (redimensionnement automatique)
    vec.push_back(5);
    vec.push_back(6);
    vec.push_back(2);

    // Taille du vecteur
    std::cout << "Le vecteur contient " << vec.size() << " éléments" << std::endl;

    // Accès aux éléments par indice
    std::cout << "Premier élément : " << vec[0] << std::endl;
```

```
    // Modification 'dun élément
    vec[1] = 12;

    // Parcours du vecteur avec une boucle
    for (int k = 0; k < vec.size(); ++k) {
        std::cout << "Élément " << k << " : " << vec[k] << std::endl;
    }

    return 0;
}
```

Sécurité d'accès

[Attention] : accéder à un élément en dehors des bornes est un **comportement indéfini (undefined behavior)**, qui peut provoquer un crash du programme.

```
// Mauvais usage : peut provoquer une erreur ou un comportement imprévisible
// vec[8568] = 12;

// Accès sécurisé (vérification des bornes)
vec.at(0) = 42;
```

Redimensionnement

Un vecteur peut être redimensionné dynamiquement avec la méthode `.resize(N)` :

```
vec.resize(10000);
// Les anciens éléments sont conservés
// Les nouveaux sont initialisés à 0
```

Comparaison `std::array`, `std::vector` et tableaux C

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

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

    // std::array (statique, taille fixe)
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    // std::vector (dynamique, taille variable)
    std::vector<int> vec = {1, 2, 3};

    std::cout << "Taille du tab : " << 5 << " (fixe, connue à la compilation)" << std::endl;
    std::cout << "Taille du array : " << arr.size() << std::endl;
    std::cout << "Taille du vector : " << vec.size() << std::endl;

    vec.push_back(10); // possible
    // arr.push_back(10); // impossible : taille fixe
    // tab.push_back(10); // impossible : fonction inexistante

    return 0;
}
```

Résumé

- **Tableaux C (`T var[N]`)** : simples, mais limités et peu sûrs.
- **`std::array<T, N>`** : tableau statique, taille fixée à la compilation, stocké sur la pile (stack memory).

- `std::vector<T>` : tableau dynamique, taille modifiable, stocké sur le tas (heap memory).
- Les trois stockent leurs éléments de manière **contiguë en mémoire**.
- En pratique :
  - Utilisez `std::array` pour des petites tailles fixes connues à l'avance.
  - Utilisez `std::vector` pour des données dont la taille peut varier au cours du programme.
  - Évitez les tableaux C sauf cas particuliers (interopérabilité avec du code C, bas niveau).

1.6 Conditionnelles et boucles

if / else

Structure générale :

```
if (condition) {
    // instructions si la condition est vraie
} else {
    // instructions si la condition est fausse
}
```

[Attention] Les accolades `{}` sont **optionnelles** si une seule instruction est présente :

```
if (x > 0)
    std::cout << "x est positif" << std::endl;
```

Exemple :

```
int age = 20;

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

if / else if / else

Structure générale :

```
if (condition1) {
    // instructions
} else if (condition2) {
    // instructions
} else {
    // instructions par défaut
}
```

Exemple :

```
int note = 15;

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

Les boucles

La boucle while

Structure générale :

```
while (condition) {
    // instructions répétées tant que la condition est vraie
}
```

Exemple :

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

La boucle do … while

Structure générale :

```
do {
    // instructions exécutées au moins une fois
} while (condition);
```

Exemple :

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

La boucle for

Structure générale :

```
for (initialisation; condition-continuation; incrément) {
    // instructions répétées
}
```

Exemple :

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

La boucle for étendue (C++11)

Structure générale :

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

Exemple :

```
#include <vector>

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

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

Extension : switch / case

Le `switch` permet de tester plusieurs valeurs d'une même variable entière ou caractère.

Structure générale :

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

[Attention] Ne fonctionne qu'avec des types entiers ou caractères.
Le mot-clé `break` évite d'exécuter les blocs suivants.

1.7 Conteneurs associatifs : `std::map`

Un `std::map` est un conteneur associatif de la bibliothèque standard qui stocke des paires clé/valeur triées par clé. Chaque clé est unique et permet d'accéder efficacement à la valeur correspondante (recherche en O(log n)).

- **Inclus** : `#include <map>`
- **Ordre** : les éléments sont triés par leur clé (utilise `operator<` par défaut).
- **Accès** : `operator[]` crée une valeur par défaut si la clé n'existe pas ; `find` permet de tester l'existence sans créer.

Exemple simple : compter la fréquence de mots

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

int main() {
    std::map<std::string, int> counts;

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

    // Parcours et affichage
    for (auto pair : counts) {
        std::cout << pair.first << " : " << pair.second << std::endl;
    }
    // Affiche:
    // avocat : 8
    // banane : 4
    // pomme : 6

    // Recherche sans création
    auto it = counts.find("orange");
    if (it == counts.end())
        std::cout << "orange non trouvé" << std::endl;

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

    return 0;
}
```

Remarques :

- Utilisez `operator[]` pour insérer/accéder rapidement. Une entrée est automatiquement créé si la clé est absente.
- Pour tester l'existence sans créer, utilisez `find`.

1.8 Durée de vie des variables

En C++, la durée de vie (ou **scope**) d'une variable est déterminée par le **bloc d'instructions** dans lequel elle est déclarée.
Un bloc est défini par des accolades `{ ... }`.
La variable existe depuis sa déclaration jusqu'à l'accolade fermante `}` du bloc.

Exemple 1 : variable locale à un bloc

```cpp
int main()
{
    if (true) {
        int x = 5; // x est défini dans le bloc "if"
        std::cout << x << std::endl;
    }
    // Ici, x 'nexiste plus : il est détruit à la fin du bloc
}
```

Exemple 2 : variable définie dans un bloc englobant

```cpp
int main()
{
    int x = 5; // x est défini dans le bloc de la fonction main()
    if (true) {
        std::cout << x << std::endl; // x peut être utilisé dans ce sous-bloc
    }
    // x existe toujours 'jusquà la fin de main()
}
```

Remarques importantes

- Ce comportement est **différent de Python**, où une variable définie dans un `if` ou une boucle reste accessible jusqu'à la fin de la fonction.

- Il est **interdit** de définir plusieurs variables ayant le même nom dans un même bloc.
  - Cela est possible dans des **sous-blocs** :

    ```cpp
    int x = 5;
    {
        int x = 10; // autorisé mais à éviter, car peu lisible
        std::cout << x << std::endl; // affiche 10
    }
    std::cout << x << std::endl; // affiche 5
    ```

- **Bonne pratique** : déclarez vos variables dans le bloc de plus courte durée de vie possible.
  Cela améliore la lisibilité du code et réduit les risques d'erreurs.

1.9 Fonctions

En C++, une **fonction** est un bloc de code réutilisable qui effectue une tâche particulière.
La syntaxe générale est la suivante :

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

Exemple simple

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

- Une fonction qui ne renvoie pas de valeur aura pour type `void`.
- Une fonction qui ne prend pas d'argument aura simplement des parenthèses vides.
- La première ligne décrivant le nom et les types de la fonction est appelée **signature** ou **en-tête** de la fonction.
- Le reste est appelé le **corps** ou **implémentation** de la fonction.

Déclaration et définition

En C++, il est nécessaire que la **signature** d'une fonction soit déclarée avant son utilisation. Sinon, il y aura une erreur de compilation.

Exemple correct (définition avant utilisation)

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

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

Exemple correct (déclaration puis définition)

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

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

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

Exemple incorrect

```
int main()
{
    int c = addition(5, 3); // ERREUR : addition 'nest pas encore déclarée
}

int addition(int a, int b)
{
```

```
    return a + b;
}
```

Exemple : fonction `norm`

Écrivons une fonction qui calcule la **norme euclidienne** d'un vecteur 3D de coordonnées `(x, y, z)` :

```cpp
#include <iostream>
#include <cmath> // pour std::sqrt

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

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

Sortie attendue :

```
Norme de (1,0,0) : 1
Norme de (0,3,4) : 5
Norme de (1,2,2) : 3
```

Fonctions mathématiques utiles

- Carré : `float x2 = x * x;`
- Racine carrée : `float y = std::sqrt(x);`
- Puissance : `float y = std::pow(x, p);`

[Attention] Ne pas utiliser `^` ni `**` en C++ : ce ne sont **pas** des opérateurs de puissance.

Surcharge de fonctions (Function Overloading)

En C++, plusieurs fonctions peuvent partager le **même nom** tant que leurs **paramètres diffèrent**. C'est ce qu'on appelle la **surcharge (overloading)**.

Exemple

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

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

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

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

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

Résumé

- Une fonction a une **signature** (en-tête) et un **corps** (implémentation).
- Elle doit être déclarée avant utilisation.
- Les fonctions peuvent renvoyer une valeur (`return`) ou être `void`.
- Les **fonctions surchargées** permettent d'utiliser un même nom avec des paramètres différents.

1.10 Passage d'arguments: copie, référence

En C++, les **arguments des fonctions** sont passés par **copie** par défaut :
- Les modifications faites dans la fonction restent locales.
- Pour de gros objets (vecteurs, tableaux, structures), la copie peut être **coûteuse** en performance.

Exemple avec passage par copie

```cpp
#include <iostream>

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

int main() {
    int x = 3;
    increment(x);
    std::cout << x << std::endl; // affiche 3 (x n'est pas modifié)
}
```

Ici, la variable `x` n'est pas modifiée dans `main` car `increment` travaille sur une **copie**.

Passage par référence

On peut utiliser le symbole `&` dans la signature pour passer un argument **par référence**.
Cela permet de modifier directement la variable originale :

```cpp
#include <iostream>

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

int main() {
    int x = 3;
    increment(x);
    std::cout << x << std::endl; // affiche 4 (x est modifié)
}
```

Une **référence** est un alias : la fonction accède à la variable originale et non à une copie.

Exemple avec `std::vector`

Considérons une fonction qui multiplie les valeurs d'un vecteur :

```cpp
#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;
}
```

Sortie attendue :

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

Ici, `vec` est passé **par copie** à `multiply_values`.
La modification est faite sur une copie locale, donc `vec` dans `main` reste inchangé.

Passage par référence (correction)

Modifions la signature pour passer le vecteur par référence :

```
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;
}
```

Résultat attendu :

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

Références constantes

Si l'on souhaite éviter la copie **sans modifier** le vecteur, on peut utiliser une **référence constante** :

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

Ce type de passage permet :
1. D'éviter la copie des données.
2. D'assurer que les valeurs ne seront pas modifiées dans la fonction.

Bonne pratique : utiliser des **références constantes** pour les gros objets qui ne doivent pas être modifiés.

1.11 Classes

En C++, une **classe** (ou une **struct**) est un moyen de regrouper dans une même entité :

- des **attributs** (données membres),
- et des **méthodes** (fonctions membres) qui opèrent sur ces données.

On parle alors d'**objet** pour désigner une instance de la classe.

Déclaration et utilisation d'un objet simple

```cpp
#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

En C++, les objets peuvent être définis avec le mot-clé **`struct`** ou **`class`** :

```cpp
struct vec3 {
    float x, y, z; // Par défaut : public
};

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

Différence principale :

- Dans une **struct**, les membres sont **publics par défaut**.
- Dans une **class**, les membres sont **privés par défaut**.

En pratique :

- On utilise souvent `struct` pour des objets simples qui agrègent des données publiques.
- On préfère `class` lorsque l'on souhaite encapsuler des données privées avec des méthodes d'accès.

Méthodes (fonctions membres)

Une classe peut définir des méthodes, c'est-à-dire des fonctions qui manipulent directement ses attributs.

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

struct vec3 {
    float x, y, z;

    float norm() const;    // méthode qui ne modifie pas 'lobjet
    void display() const;  // idem
    void normalize();      // méthode qui modifie (x,y,z)
};

// Implémentation des méthodes
float vec3::norm() const {
    return std::sqrt(x * x + y * y + z * z);
}
```

```cpp
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};

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

    // Normalisation
    p2.normalize();

    // Affichage
    p2.display();

    return 0;
}
```

Remarques

- Les méthodes peuvent accéder directement aux attributs de l'objet sans utiliser `this->`, bien que ce soit possible.
- On sépare généralement la **déclaration** (dans la struct/class) et l'**implémentation** (avec `NomClasse::NomMethode`).
- Le mot-clé **`const`** placé après une méthode indique qu'elle ne modifie pas l'objet. Cela améliore la robustesse et la lisibilité.

Constructeurs et destructeur

Une classe peut définir des **constructeurs** pour initialiser ses objets et un **destructeur** pour exécuter du code lors de leur destruction.

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

struct vec3 {
    float x, y, z;

    // Constructeur vide
    vec3();

    // Constructeur personnalisé
    vec3(float v);

    // Destructeur
    ~vec3();
};

// Initialisation à 0
vec3::vec3() : x(0.0f), y(0.0f), z(0.0f) { }

// 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()
}
```

Constructeur ou destructeur par défaut (`= default`)

Dans certains de cas, on ne souhaite pas redéfinir un constructeur ou un destructeur, mais simplement demander explicitement au compilateur de générer automatiquement **l'implémentation par défaut**. On utilise alors la syntaxe `= default`.

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

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

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

Ceci est équivalent à ne rien écrire, mais a deux avantages :

- Lisibilité : cela rend explicite qu'un constructeur ou un destructeur existe et doit être celui fourni par le compilateur.
- Robustesse : permet d'éviter certaines suppressions implicites de constructeur/destructeur si d'autres sont définis dans la classe.

Fonctions membres vs fonctions non membres

En C++, le choix entre une **méthode** (fonction membre) et une **fonction externe** est laissé au développeur. Par exemple, la norme peut aussi être définie comme une fonction indépendante :

```
#include <cmath>

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

// Norme comme fonction non-membre
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); // appel en tant que fonction
}
```

L'utilisation de `const&` évite de copier inutilement l'objet.

1.12 Écriture/lecture de fichiers externes

En C++, la bibliothèque **`<fstream>`** permet d'écrire et de lire des données dans des fichiers. Celle-ci fournit trois classes principales :

- **`std::ifstream`** (*input file stream*) : pour lire un fichier (entrée).
- **`std::ofstream`** (*output file stream*) : pour écrire dans un fichier (sortie).
- **`std::fstream`** : pour combiner lecture et écriture.

Exemple : écriture d'un vecteur dans un fichier

On souhaite sauvegarder les coordonnées d'un `vec3` dans un fichier texte.

```cpp
#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;
}
```

Après exécution, le fichier `vec3.txt` contient :

```
Bonjour C++ !
1 2 3.5
```

Exemple : lecture d'un vecteur depuis un fichier

On peut ensuite relire ce `vec3` depuis le fichier :

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

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;
}
```

Affichage attendu :

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

Modes d'ouverture

Lors de l'ouverture d'un fichier, on peut préciser des **modes** :

- `std::ios::in` : lecture (par défaut pour `ifstream`).
- `std::ios::out` : écriture (par défaut pour `ofstream`).
- `std::ios::app` : ajout à la fin du fichier sans l'effacer.
- `std::ios::binary` : lecture/écriture en mode binaire (ex. images).

Exemple :

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

1.13 Organisation des fichiers de code

Lorsqu'un programme devient volumineux, il est nécessaire de **séparer le code en plusieurs fichiers** afin de préserver la lisibilité, la modularité et de faciliter la maintenance.

Une organisation typique avec des classes en C++ repose sur **trois types de fichiers** :

1. **Fichier d'en-tête (.hpp ou .h)**
   - Contient les **déclarations** des classes, des structures et des fonctions.
   - Sert d'interface publique : ce que les autres fichiers doivent connaître pour utiliser la classe.
2. **Fichier d'implémentation (.cpp)**
   - Contient le **code des méthodes** et des fonctions déclarées dans le `.hpp`.
   - Réalise l'implémentation détaillée des comportements.
3. **Fichier principal ou d'utilisation (main.cpp, etc.)**
   - Contient la fonction `main()` et utilise les classes/fonctions en incluant le fichier d'en-tête.

Exemple : organisation avec une classe `vec3`

Fichier d'en-tête — `vec3.hpp`

```
#pragma once
#include <cmath>

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

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

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

Fichier d'implémentation — `vec3.cpp`

```
#include "vec3.hpp"


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

```cpp
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;
}
```

Fichier d'utilisation — `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;
}
```

Remarques importantes

- La directive `#include "vec3.hpp"` **copie-colle** le contenu du fichier `.hpp` au moment de la compilation.
- **Tous les fichiers** qui utilisent `vec3` doivent inclure son fichier d'en-tête (`vec3.hpp`).
- Ne jamais inclure directement un fichier `.cpp` dans un autre fichier.
- Les déclarations partagées doivent toujours être dans un **fichier d'en-tête unique**, inclus par tous les fichiers concernés.

À propos de `#pragma once`

La directive `#pragma once` est utilisée en en-tête pour éviter les inclusions multiples d'un même fichier. Lorsqu'un fichier `.hpp` est inclus plusieurs fois (directement ou indirectement), cela peut provoquer des erreurs de compilation liées à des redéfinitions de classes ou de fonctions.

Avec `#pragma once`, le compilateur garantit que le contenu du fichier ne sera inclus qu'une seule fois, même si plusieurs fichiers tentent de l'inclure.
C'est une alternative plus concise et lisible que les gardes d'inclusion classiques utilisant `#ifndef`, `#define` et `#endif`.

En pratique, il est recommandé d'ajouter systématiquement `#pragma once` en tête de vos fichiers d'en-tête.

1.14 Compilation

En C++, la **compilation** est le processus qui transforme le code source lisible par un humain (fichiers `.cpp` et `.hpp`) en un programme exécutable compréhensible par l'ordinateur. Cette transformation s'effectue en plusieurs étapes. Le compilateur commence par analyser le code et le traduit en **code assembleur**.

Le **code assembleur** est un langage de bas niveau qui correspond directement aux instructions compréhensibles par le processeur. Contrairement au C++ qui est portable entre systèmes et processeurs, l'assembleur est **dépendant de l'architecture matérielle** (Intel x86, ARM, etc.). Chaque ligne de C++ peut ainsi donner lieu à une ou plusieurs instructions assembleur, telles que des opérations de calcul, de copie mémoire ou de saut conditionnel.

Ensuite, ce code assembleur est converti en **code machine** binaire qui constitue le langage natif du processeur. Ce code est stocké dans un fichier objet binaire. Finalement, un **éditeur de liens** (*linker*) assemble les différents fichiers objets et les bibliothèques utilisées pour produire l'exécutable final.

Ainsi, le rôle de la compilation est de **traduire un langage de haut niveau (C++) en instructions de bas niveau (assembleur, puis machine)** que le processeur peut exécuter directement, tout en optimisant les performances.

Schéma simple du pipeline de compilation

```
Fichier source (.cpp)
        ↓ (compilateur)
   Fichier objet (.o)
        ↓ (linker / éditeur de liens)
   Exécutable (programme binaire)
```

Schéma avec plusieurs fichiers sources

```
 main.cpp   vec3.cpp   utils.cpp
     ↓         ↓          ↓
  (compilateur) (compilateur) (compilateur)
     ↓         ↓          ↓
 main.o    vec3.o     utils.o
     ↓         ↓          ↓
  [linker / éditeur de liens]
            ↓
     programme exécutable
```

Exemple de code assembleur

Exemple C++

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

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

Exemple d'assembleur généré (x86-64, simplifié)

```
add(int, int):              # Début de la fonction add
    mov     eax, edi        # Copier le 1er argument (a) dans eax
    add     eax, esi        # Ajouter le 2ème argument (b)
    ret                     # Retourner eax (résultat)

main:                       # Début de la fonction main
    push    rbp             # Sauvegarde du pointeur de base
    mov     edi, 2          # Charger 2 dans le registre edi (1er argument)
    mov     esi, 3          # Charger 3 dans le registre esi (2e argument)
    call    add(int, int)   # Appeler la fonction add
    pop     rbp             # Restaurer le pointeur de base
    ret                     # Retourner le résultat dans eax
```

Explications

- **`edi` et `esi`** : registres utilisés pour passer les 1er et 2e arguments aux fonctions (convention d'appel x86-64 System V).
- **`eax`** : registre où le résultat est stocké et retourné par la fonction.
- **`mov`** : copie une valeur dans un registre.
- **`add`** : effectue une addition entre deux registres.

- `ret` : retourne de la fonction, en utilisant la valeur présente dans `eax` comme résultat.

Sous Linux/MacOS

Sur Linux et MacOS, les compilateurs les plus utilisés sont **g++** (GNU) et **clang++** (LLVM).
Pour compiler un programme simple (un seul fichier) :

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

ou

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

- `main.cpp` : fichier source C++ à compiler.
- `-o programme` : nom de l'exécutable produit.

Si le projet contient plusieurs fichiers, il devient fastidieux de tout compiler à la main. On utilise alors un **Makefile** avec l'outil **make**, qui décrit les dépendances et les règles de compilation.

Exemple minimal de Makefile :

Voici ton **Makefile annoté** avec la **syntaxe générale en commentaire** :

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

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

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

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

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

Windows

Sur Windows, le compilateur est fourni directement par **Microsoft Visual Studio** (MSVC). Il ne repose pas sur `make` ni sur des Makefiles. Au lieu de cela, le code est organisé dans un **projet Visual Studio** (`.sln`) qui décrit les fichiers, dépendances et options de compilation.

L'IDE Visual Studio se charge de lancer automatiquement le compilateur MSVC lorsque vous appuyez sur "Build" ou "Run". Ainsi, il n'est pas nécessaire (et pas pratique) d'appeler manuellement `cl.exe` en ligne de commande.

Meta-configuration via CMake

Pour éviter d'écrire un Makefile spécifique à Linux **et** un projet Visual Studio spécifique à Windows, on utilise **CMake**.

- CMake est un outil de **génération de projet**.
- Il lit un fichier de configuration (`CMakeLists.txt`) et génère automatiquement les fichiers adaptés à votre système :
  - **Linux/MacOS →** un **Makefile** utilisable avec `make`.
  - **Windows →** un **projet Visual Studio** (`.sln`).

Exemple d'utilisation sous Linux/MacOS:

```
# Depuis le répertoire du projet
mkdir build
cd build
cmake ..
make          # sous Linux/MacOS
```

En résumé

- Linux/MacOS : compilation via `g++` ou `clang++`, automatisation via **Makefile**.
- Windows : compilation via MSVC à travers un projet Visual Studio.
- CMake : outil multi-plateforme qui génère automatiquement le bon type de projet (Makefile ou `.sln`).

2 Types fondamentaux, encodage

En C++, les variables sont **typées** : chaque variable correspond à un **espace mémoire** (une ou plusieurs cases) interprété selon un **type**. Exemples de types fondamentaux :

```cpp
int a = 5;        // entier signé (typiquement 4 octets)
float b = 5.0f;   // flottant simple précision (4 octets)
double c = 5.0;   // flottant double précision (8 octets)
char d = 'k';     // caractère (1 octet = 8 bits), équivaut à 107 en ASCII
size_t e = 100;   // entier non signé permettant d'encoder une position en mémoire (8 octets sur machines 64
     bits), il est utilisé pour indiquer les tailles de tableaux ex. size() d'un std::vector.
```

Remarques importantes :

- La taille des types dépend de l'**architecture** et du compilateur (sauf `char` garanti sur 1 octet).
- Aucun type n'occupe moins d'un octet (8 bits).
- Pour des raisons d'efficacité, la mémoire est souvent **alignée** : certaines structures ajoutent du **padding** (octets vides) pour s'aligner sur 4 ou 8 octets.

2.1 Encodage des entiers

Représentation binaire

Un entier est représenté en **binaire** :

- Chaque bit vaut `0` ou `1`.
- Un ensemble de bits est regroupé en **octets** (8 bits).
- Les valeurs sont interprétées en base 2.

Exemple :

Décimal	Binaire (8 bits)
0	`00000000`
1	`00000001`
2	`00000010`
3	`00000011`
4	`00000100`
156	`10011100`

Un entier peut être représenté sur plusieurs octets :

- 4 octets (`int` classique) = 32 bits → jusqu'à `2^32` valeurs possibles.
- 8 octets (`long long`) = 64 bits → jusqu'à `2^64` valeurs possibles.

Entiers non signés

Un **unsigned int** sur 4 octets (32 bits) code des valeurs de `0` à `2^32 - 1 = 4 294 967 295`.
Exemple en hexadécimal (représentation pratique des octets) :

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

Rappel :

- 1 octet (8 bits) = 2 caractères hexadécimaux
- Ex. : `10011100` = `9C` en hexadécimal = `156` en décimal

Entiers signés et complément à deux

Les entiers signés utilisent le bit le plus à gauche (**MSB**) pour coder le signe :

- `0` → positif
- `1` → négatif

Méthode d'encodage : **complément à deux**.

- Pour obtenir la valeur négative d'un entier :
  1. On inverse tous les bits.
  2. On ajoute 1.

Exemple sur 8 bits :

```
  00000101 = +5
Inverse → 11111010
Ajout +1 → 11111011 = −5
```

Conséquence :

- Sur 8 bits, plage de valeurs : de `−128` à `+127`.
- Sur 32 bits (`int`) : de `−2 147 483 648` à `+2 147 483 647`.

Exemple pratique

Prenons l'entier signé encodé sur 2 octets :

```
C4 8D (hexadécimal)
= 11000100 10001101 (binaire)
```

- Interprété comme **non signé** : `50317`.
- Interprété en **signé complément à deux** :
  - Inversion des bits → `00111011 01110010`
  - Ajout 1 → `00111011 01110011` = `15219`
  - Donc la valeur = `−15219`.

2.2 Encodage des nombres flottants

Les flottants (`float`, `double`) suivent la norme **IEEE 754**.
Un nombre flottant est représenté par trois parties :

1. **Signe** (1 bit)
2. **Exposant** (8 bits pour `float`, 11 bits pour `double`)
3. **Mantisse** (23 bits pour `float`, 52 bits pour `double`)

Formule :

$$x = (-1)^s \times (1 + mantisse) \times 2^{exposant - biais}$$

- `float` (32 bits) → biais = 127
- `double` (64 bits) → biais = 1023

Exemple : `46 3F CC 30` (float en hexadécimal) = `12275.046875` en décimal.
[Attention] Propriétés importantes :

- La précision dépend de la valeur : plus grande autour de 0, plus faible pour de très grands nombres.
- Certains nombres ne sont **pas représentables exactement** (ex. `0.1`, `0.4`).
- Toujours comparer deux flottants avec une **tolérance ε** :

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

2.3 Notion d'endianness

Quand un entier occupe plusieurs octets (par exemple un `int` de 4 octets), l'ordinateur doit décider **dans quel ordre les octets sont stockés en mémoire**. C'est ce qu'on appelle **l'endianness** (ou ordre des octets).

Deux conventions principales

1. **Little Endian** (Intel x86, ARM en mode par défaut)
   - L'octet **de poids faible** (least significant byte) est stocké en premier (à l'adresse la plus petite).
   - Exemple :

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

   Représentation mémoire (adresses croissantes) :

   ```
   Adresse : 1000   1001   1002   1003
   Contenu : 78     56     34     12
   ```

2. **Big Endian** (certaines architectures réseau, PowerPC, anciens processeurs)
   - L'octet **de poids fort** (most significant byte) est stocké en premier.
   - Pour la même valeur `0x12345678` :

   ```
   Adresse : 1000   1001   1002   1003
   Contenu : 12     34     56     78
   ```

Pourquoi est-ce important ?

- **Compatibilité réseau** Les protocoles (TCP/IP, etc.) imposent le Big Endian (*network byte order*). Les PC classiques (Intel) utilisent le Little Endian : il faut donc convertir avant d'envoyer ou après réception.
- **Fichiers binaires** Si un programme écrit un fichier binaire en Little Endian, il doit préciser cet ordre. Sinon, sur une machine Big Endian, les valeurs lues seront fausses.
- **Interopérabilité** Toute communication entre machines hétérogènes doit expliciter l'ordre des octets.

2.4 Synthèse des types fondamentaux

Type	Description	Taille typique (x86/64 bits)	Exemple de déclaration
`char`	caractère ASCII (ou petit entier signé)	1 octet	`char c = 'A';`
`bool`	valeur booléenne (`true` ou `false`)	1 octet (optimisé en vector)	`bool b = true;`
`short`	entier court signé	2 octets	`short s = 123;`
`int`	entier signé standard	4 octets	`int a = 42;`
`long`	entier signé (taille variable selon archi)	4 octets (Windows), 8 (Linux)	`long l = 100000;`
`long long`	entier long signé (garanti >= 64 bits)	8 octets	`long long x = 1e12;`
`unsigned`	entier non signé (≥0 uniquement)	même taille que signé	`unsigned u = 42;`
`float`	nombre flottant simple précision (IEEE754)	4 octets	`float f = 3.14f;`
`double`	nombre flottant double précision	8 octets	`double d = 2.718;`
`long double`	flottant précision étendue (dépend archi)	8, 12 ou 16 octets	`long double pi = 3.14159;`
`size_t`	entier non signé pour l'adressage mémoire	8 octets (64 bits)	`size_t n = vec.size();`

Type	Description	Taille typique (x86/64 bits)	Exemple de déclaration
`wchar_t`	caractère large (Unicode, dépend plateforme)	2 octets (Windows), 4 (Linux)	`wchar_t wc = 'é';`

Attention: La taille peut varier selon le compilateur et l'architecture, **sauf `char` qui fait toujours 1 octet**.

2.5 Obtenir la taille avec `sizeof`

En C et C++, l'opérateur `sizeof` retourne la taille en octets d'un type ou d'une variable.

Exemples :

```c
#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;
}
```

Sortie typique sur une machine 64 bits :

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

Rem. : le spécificateur `%zu` est celui prévu par la norme pour afficher une valeur de type `size_t` (par ex. le résultat de `sizeof`). Il est également possible de convertir vers `unsigned long` et utiliser `%lu`.

2.6 Remarques importantes

- `sizeof(type)` est évalué **à la compilation**, sans exécuter le programme.
- La taille d'un type peut changer selon l'architecture (32 bits vs 64 bits).
- L'alignement mémoire peut introduire du **padding** dans les `struct`.
- Pour connaître les tailles avec certitude sur ta machine, il est conseillé d'écrire un petit programme avec `sizeof`.

2.7 Types à tailles spécifiques

Pour obtenir des tailles déterministes (indépendantes de l'architecture), le standard C/C++ définit les types dans l'en-tête `<cstdint>` (C++11 / C99). Ces types garantissent un nombre de bits précis, ce qui est essentiel pour la sérialisation, les formats binaires et les protocoles réseau.

Principaux types fixes :

- `uint8_t` / `int8_t` : entier non signé / signé sur 8 bits
- `uint16_t` / `int16_t` : entier non signé / signé sur 16 bits
- `uint32_t` / `int32_t` : entier non signé / signé sur 32 bits
- `uint64_t` / `int64_t` : entier non signé / signé sur 64 bits

Exemples utiles complémentaires :

- `int_fast32_t`, `uint_fast32_t` : types entiers au moins de 32 bits mais choisis pour de meilleures performances sur la plateforme
- `int_least16_t`, `uint_least16_t` : types entiers d'au moins 16 bits (garantie minimale)
- `intptr_t`, `uintptr_t` : entiers signés/non signés capables de contenir une valeur de pointeur

Exemple d'utilisation :

```cpp
#include <cstdint>
#include <cinttypes> // pour les macros 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));

  // utilisation sûre avec printf :
  std::printf("c = %" PRIu32 "\n", c);
  return 0;
}
```

2.8 Opérations bit à bit

Les opérations bit à bit (bitwise) permettent de manipuler directement les bits d'un entier. Elles sont très utiles pour travailler sur des flags, des masques, optimiser des calculs simples, ou pour le traitement bas-niveau de données (compression, formats binaires, etc.).

Principales opérations en C/C++ :

- `&` : ET bit à bit
- `|` : OU bit à bit
- `^` : XOR (OU exclusif) bit à bit
- `~` : NOT (négation) bit à bit
- `<<` : décalage à gauche (shift left)
- `>>` : décalage à droite (shift right)

Exemples simples :

```cpp
unsigned a = 0b1100; // la notation 0bxxxx permet de définir une valeur en binaire, ici 1100 en binaire => 12 en
     base décimale.
unsigned b = 0b1010; // 1010 en binaire => 10 en décimale

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 de tous les bits

// décalements
unsigned left  = a << 1; // 11000 (24) : décalage vers la gauche (multiplication par 2)
unsigned right = a >> 2; // 0011 (3)  : décalage vers la droite (division par 2)

// affichez en hex / décimale selon besoin
```

Masques et tests de bits

On utilise des masques pour isoler, définir ou effacer des bits :

```cpp
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

// activer un flag
flags |= FLAG_B; // flags = 0b0010
```

```cpp
// tester si un flag est activé
bool hasB = (flags & FLAG_B) != 0;

// désactiver un flag
flags &= ~FLAG_B; // efface le bit 1

// basculer (toggle) un flag
flags ^= FLAG_C; // inverse l'état du bit 2
```

Conseils importants

- Utilisez des types non signés (`unsigned`, `uint32_t`, `uint64_t`) pour les opérations bit à bit : le comportement des décalages sur des entiers signés négatifs peut être indéfini ou dépendre de l'implémentation.
- Le décalage à gauche `x << n` multiplie par `2^n` lorsque cela ne provoque pas de débordement. Le décalage à droite `x >> n` divise par `2^n` pour les types non signés.
- Pour isoler un octet dans un mot (utile pour endianness ou extraction) :

```cpp
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)
```

Utiliser `std::bitset` pour afficher/manipuler des bits de façon sûre et lisible :

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

std::bitset<8> bs(0b10110010);
std::cout << bs << "\n"; // affiche 10110010
bs.flip(0); // bascule le bit 0
bs.set(3);  // met à 1 le bit 3
bs.reset(7);// met à 0 le bit 7
```

2.9 Résumé

- Les types classiques couvrent les **entiers signés/non signés**, les **flottants** et les **caractères**.
- Leur taille n'est pas toujours fixe (sauf `char` = 1 octet garanti).
- `sizeof` permet de connaître précisément la taille d'un type ou d'une variable sur une architecture donnée.

3 Pointeurs

3.1 Notion de stockage et d'adressage en mémoire

La mémoire d'un ordinateur peut être vue comme un **grand tableau linéaire** de cases.

- Chaque case contient **un octet** (soit 8 bits).
- Chaque case possède une **adresse unique**, qui est un nombre permettant d'y accéder.

On peut donc imaginer la mémoire comme une succession de cases numérotées :

```
Adresse   Contenu
1000      10101010
1001      00001111
1002      11110000
1003      01010101
...
```

Ici :

- chaque ligne représente **un octet** de mémoire,
- l'**adresse** (1000, 1001, …) est un entier géré par le processeur,
- le **contenu** est un ensemble de 8 bits (`0` ou `1`).

Adresses et variables

Quand on déclare une variable en C++ :

```
int a = 42;
```

- Le compilateur réserve **4 octets consécutifs** (sur une architecture 32 bits ou 64 bits).
- Supposons que la variable commence à l'adresse 1000. La mémoire peut ressembler à ceci :

```
Adresse   Contenu
1000      00101010   (0x2A)
1001      00000000
1002      00000000
1003      00000000
```

Ainsi :

- la variable `a` est vue comme un **tout** (`42`),
- mais en réalité, elle est stockée sous forme de **quatre octets successifs** en mémoire.

Taille et alignement

- **char** : 1 octet
- **short** : 2 octets
- **int** : 4 octets (le plus souvent)
- **long long** : 8 octets
- **float** : 4 octets
- **double** : 8 octets

Remarque: La taille peut varier selon l'architecture, mais **1 octet = 8 bits est garanti**.

Par souci de performance, le compilateur peut introduire du **padding** (remplissage avec des 0) pour que certaines variables commencent à des adresses multiples de 2, 4 ou 8. Cela facilite l'accès mémoire pour le processeur.

Importance de l'adresse

L'adresse mémoire est ce qui permet :

- d'identifier précisément où se trouve une variable,
- d'accéder à ses octets,
- de manipuler des structures de données complexes.

Exemple d'analogie

On peut comparer la mémoire :

- à une **bibliothèque** où chaque case mémoire serait un livre,
- l'**adresse** est le numéro du rayon + numéro du livre,
- le **contenu** est l'information écrite dans ce livre (les bits).

Pour accéder à une donnée, le processeur doit connaître l'**adresse exacte**.

Résumé

- La mémoire est organisée en cases de **1 octet** (8 bits).
- Chaque case possède une **adresse unique**.
- Les variables occupent une ou plusieurs cases **consécutives**.
- Les adresses permettent au processeur de retrouver et manipuler ces valeurs.
- Cette vision est essentielle pour comprendre comment fonctionnent les **pointeurs** et l'**allocation mémoire dynamique**.

3.2 Adresse d'une variable

Chaque variable en mémoire possède une **adresse**, c'est-à-dire la position de son premier octet dans le grand tableau de la mémoire. En langage C (et donc aussi en C++), on peut accéder à cette adresse grâce à l'opérateur `&` (dit *adresse de*).

Exemple simple

```c
#include <stdio.h>

int main() {
    int a = 42;

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

    return 0;
}
```

Sortie possible (l'adresse dépend de l'exécution et de la machine) :

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

- `%d` affiche la valeur entière (`42` ici).
- `%p` affiche une adresse mémoire (format pointeur).
- `&a` signifie "l'adresse de la variable `a`".

Lecture et écriture via la fonction C `scanf`

Quand on utilise `scanf`, on doit fournir l'**adresse de la variable** dans laquelle stocker le résultat.

```c
#include <stdio.h>

int main() {
    int age;

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

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

    return 0;
}
```

- Ici `scanf("%d", &age)` place la valeur lue directement dans la case mémoire de `age`.
- Si on avait écrit `scanf("%d", age)` (sans `&`), le programme plantera, car `scanf` a besoin de **l'adresse** pour modifier la variable.

Observation de l'adresse

On peut constater que deux variables successives en mémoire ont des adresses différentes, séparées par leur taille en octets.

```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;
}
```

Exemple de sortie :

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

Remarque: Les adresses sont proches mais pas forcément dans l'ordre croissant, car le compilateur et le système peuvent organiser les variables différemment (pile, alignement mémoire, etc.).

3.3 Passage d'argument

Passage par valeur (comportement par défaut)

En C et C++, les arguments des fonctions sont **passés par valeur** :

- Quand on appelle une fonction, le programme crée une **copie de la variable** dans la mémoire de la fonction.
- La fonction travaille donc sur sa propre copie.

Exemple :

```c
#include <stdio.h>

void increment(int x) {
    x = x + 1;  // modifie uniquement la copie locale
}

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

Explication mémoire :

- `a` dans `main` occupe une zone mémoire.
- Lors de l'appel `increment(a)`, la valeur `5` est copiée dans une nouvelle variable `x` locale à la fonction.
- Modifier `x` ne change pas `a`, car ce sont deux variables indépendantes.

Passage par adresse avec un pointeur

Si on veut qu'une fonction **puisse modifier la variable originale**, il faut lui transmettre non pas la valeur, mais **l'adresse de la variable**.

Exemple :

```c
#include <stdio.h>

void increment(int* p) {
    *p = *p + 1; // modifie la valeur à l'adresse pointée
}

int main() {
    int a = 5;
    increment(&a); // on passe l'adresse de a
    printf("a = %d\n", a); // affiche 6
    return 0;
}
```

Explication détaillée :

1. Dans `main`, on a la variable `a` (valeur 5) stockée à une certaine adresse mémoire (par ex. 1000).
2. L'expression `&a` produit cette adresse (1000).
3. Lors de l'appel `increment(&a)`, ce n'est **pas `a` qui est copié**, mais son **adresse** (1000).
    - La fonction reçoit donc un **pointeur** `p`, qui est une copie de l'adresse.
4. À l'intérieur de `increment`, `*p` signifie « la valeur contenue à l'adresse `p` ».
    - Donc `*p = *p + 1;` va chercher la valeur `5` à l'adresse 1000, l'incrémente, et stocke `6` à la même place.
5. Comme `p` désigne la mémoire de `a`, la variable `a` est réellement modifiée.

Résumé des mécanismes

- **Passage par valeur** : on copie la valeur dans une nouvelle variable locale. La variable d'origine n'est jamais modifiée.
- **Passage par adresse (pointeur)** : on copie l'**adresse** dans un pointeur. La fonction a donc accès à la même zone mémoire, et peut modifier la variable originale via `*p`.

Schéma (simplifié en ASCII) :

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

Appel increment(&a) :
    copie de l'adresse 1000 dans p

increment:
 p = 1000
 *p → valeur stockée à l'adresse 1000 → 5
 *p = 6   (modifie la mémoire partagée avec a)
```

Bonnes pratiques avec les pointeurs

Un pointeur est une variable qui contient une adresse mémoire. Cependant, si un pointeur n'est pas initialisé, il peut contenir **une adresse aléatoire**, ce qui conduit à des comportements imprévisibles (segmentation fault, corruption mémoire).

Règle essentielle : toujours initialiser les pointeurs.

En C++ moderne, on utilise `nullptr` pour indiquer qu'un pointeur ne pointe vers rien :

```cpp
#include <iostream>

int main() {
    int* p = nullptr; // pointeur initialisé, mais ne pointe vers rien

    if(p == nullptr) {
        std::cout << "Le pointeur est vide, pas 'daccès dangereux." << std::endl;
    }

    return 0;
}
```

Exemple de mauvaise pratique

```cpp
int* p;      // pointeur non initialisé (dangereux !)
*p = 10;     // comportement indéfini → crash probable
```

Ici, `p` contient une valeur indéterminée : accéder à `*p` est **dangereux**.

Exemple correct

```cpp
int* p = nullptr;   // pointeur sûr, mais vide
if(p != nullptr) {
    *p = 10;        // on accède uniquement si p pointe vers une variable valide
}
```

Résumé

- Toujours initialiser vos pointeurs (avec `nullptr` par défaut).
- Toujours vérifier qu'un pointeur n'est pas nul avant de l'utiliser.
- Préférez les références (`&`) ou les conteneurs modernes (`std::vector`, `std::unique_ptr`, `std::shared_ptr`) quand c'est possible, afin d'éviter les erreurs de gestion mémoire.

3.4 Cas des tableaux contigus

Tableaux C

En C et C++, un **tableau** est toujours stocké en mémoire comme une suite **contiguë d'octets**. Cela signifie que les éléments se suivent les uns après les autres, sans espace entre eux.

Exemple :

```cpp
#include <stdio.h>

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

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

    return 0;
}
```

Sortie possible :

```
Adresse de tab[0] : 0x7ffee6c4a90
Adresse de tab[1] : 0x7ffee6c4a94
Adresse de tab[2] : 0x7ffee6c4a98
```

On remarque que les adresses sont espacées de 4 octets (la taille d'un `int`), ce qui confirme la **contiguïté mémoire**.

Arithmétique des pointeurs

Le nom d'un tableau (`tab`) est automatiquement converti en **pointeur vers son premier élément** (`&tab[0]`). On peut alors utiliser l'**arithmétique des pointeurs** :

- `p + N` : décale le pointeur de `N` éléments.
- `*(p + N)` : accède à la valeur du `N`-ième élément.

Cela revient exactement à écrire `tab[N]`.
Exemple :

```c
#include <stdio.h>

int main() {
    int tab[3] = {10, 20, 30};
    int* p = tab; // équivaut à &tab[0]

    printf("%d\n", *(p + 0)); // 10
    printf("%d\n", *(p + 1)); // 20
    printf("%d\n", *(p + 2)); // 30

    return 0;
}
```

Ces deux écritures sont équivalentes :

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

Schéma mémoire (exemple avec tab[3])

```
Adresse : 1000   1004   1008
Contenu : 10     20     30
Indice  : tab[0] tab[1] tab[2]

p = 1000
*(p+0) → valeur à 1000 → 10
*(p+1) → valeur à 1004 → 20
*(p+2) → valeur à 1008 → 30
```

Adaptation à la taille mémoire des éléments

La contiguïté mémoire s'applique à tout type de tableau, pas seulement aux entiers. Si on définit un tableau d'objets plus volumineux (par exemple des `double` ou des `struct`), les éléments restent stockés les uns à la suite des autres.

Exemple avec `double`

```c
#include <stdio.h>

int main() {
    double tab[3] = {1.1, 2.2, 3.3};

    printf("Adresse de tab[0] : %p\n", &tab[0]);
    printf("Adresse de tab[1] : %p\n", &tab[1]);
```

```
    printf("Adresse de tab[2] : %p\n", &tab[2]);

    return 0;
}
```

Sortie possible (chaque `double` = 8 octets) :

```
Adresse de tab[0] : 0x7ffee6c4a90
Adresse de tab[1] : 0x7ffee6c4a98
Adresse de tab[2] : 0x7ffee6c4aa0
```

On voit que les adresses sont espacées de 8, car un `double` occupe 8 octets.

En C/C++, l'expression `p + N` **ne signifie pas** "ajouter N octets", mais "aller au N-ième élément à partir de p".

- Si `p` est de type `int*` et que `sizeof(int) == 4`, alors :

```
p + 1   → avance de 4 octets
p + 2   → avance de 8 octets
```

- Si `p` est de type `double*` et que `sizeof(double) == 8`, alors :

```
p + 1   → avance de 8 octets
p + 2   → avance de 16 octets
```

- De manière générale :

```
Adresse(p + N) = Adresse(p) + N * sizeof(type)
```

C'est le compilateur qui traduit l'opération en calcul d'adresse, et c'est le processeur qui fait l'addition lors de l'exécution.

Tableaux dynamiques en C++ : `std::vector`

En C++ moderne, on utilise `std::vector` plutôt que des tableaux statiques, car il offre :

- une **taille dynamique** (on peut ajouter des éléments avec `push_back`),
- une **gestion automatique de la mémoire**,
- et il conserve la **contiguïté mémoire**.

Exemple :

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

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

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

Sortie typique :

```
Adresse de v[0] : 0x7ffee6c4a90
Adresse de v[1] : 0x7ffee6c4a94
Adresse de v[2] : 0x7ffee6c4a98
```

On observe la même contiguïté qu'avec les tableaux classiques.

Arithmétique des pointeurs sur `std::vector`

On peut récupérer un pointeur sur les données internes grâce à `v.data()` ou `&v[0]`, puis utiliser la même logique que pour les tableaux C.

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

int main() {
    std::vector<int> v = {10, 20, 30};
    int* p = v.data(); // pointeur vers le premier élément

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

Résumé

- Les tableaux C et les `std::vector` stockent leurs éléments de façon **contiguë**.
- Cela permet un accès rapide par indice (`tab[i]`) ou via l'arithmétique des pointeurs (`*(p+i)`).
- Les `std::vector` offrent en plus une **taille dynamique** et une gestion sûre de la mémoire, mais conservent les mêmes propriétés fondamentales de contiguïté.

3.5 Contiguité dans les classes et struct

En C et C++, les **structures** (`struct`) et **classes** regroupent plusieurs variables (membres) dans un seul bloc mémoire. Par défaut, les champs sont rangés les uns à la suite des autres, ce qui garantit une **contiguïté mémoire**.

Exemple simple

```cpp
#include <stdio.h>

struct Point2D {
    int x;
    int y;
};

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

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

    return 0;
}
```

Sortie possible :

```
Adresse de p.x : 0x7ffee3b5a90
Adresse de p.y : 0x7ffee3b5a94
```

Ici, les deux entiers `x` et `y` (4 octets chacun) sont stockés l'un après l'autre de manière contiguë.

Padding et alignement

Pour des raisons de performance, le compilateur peut insérer des **octets de padding** entre les membres afin de respecter un alignement mémoire optimal.

Exemple :

```cpp
struct Test {
    char a;   // 1 octet
    int b;    // 4 octets
};
```

Organisation en mémoire :

```
Adresse   Contenu
1000      a (1 octet)
1001–1003 padding (3 octets inutilisés)
1004–1007 b (4 octets)
```

Exemple avec plusieurs champs

```cpp
struct Mixed {
    char c;    // 1 octet
    double d;  // 8 octets
    int i;     // 4 octets
};
```

Disposition typique sur une machine 64 bits :

```
Adresse   Champ
1000      c (1 octet)
1001–1007 padding (7 octets)
1008–1015 d (8 octets)
1016–1019 i (4 octets)
1020–1023 padding (4 octets pour alignement global)
```

Taille totale : 24 octets.

Contiguïté dans les classes

En C++, une `class` se comporte comme une `struct` du point de vue mémoire :

- Les membres de données sont placés contigus, avec les mêmes règles de padding et d'alignement.
- La différence entre `struct` et `class` est uniquement dans la visibilité par défaut (`public` vs `private`).

`std::vector` de structures

En C++ moderne, on peut stocker plusieurs objets `struct` ou `class` dans un **`std::vector`**. Le `vector` garantit que les éléments sont placés **contigus en mémoire**, exactement comme pour un tableau C.

Exemple :

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

struct Point2D {
    int x;
    int y;
};

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

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

Schéma ASCII d'un `std::vector<Point2D>`

Chaque `Point2D` occupe `sizeof(Point2D)` octets (ici, 8 octets : 2 entiers de 4 octets). Les éléments du `std::vector` sont rangés **dos à dos** en mémoire :

```
Mémoire d'un std::vector<Point2D> avec 3 éléments

Adresse : 2000       2008       2016
Contenu : [x=1, y=2] [x=3, y=4] [x=5, y=6]
Taille  :  8 octets   8 octets   8 octets
```

On voit que chaque élément est un **bloc structuré**, mais que les blocs restent **contigus**.

Résumé

- Les champs d'une `struct` ou `class` sont stockés contigus, avec du **padding** éventuel pour respecter l'alignement.
- La taille réelle peut être plus grande que la somme des champs.
- Un **`std::vector<struct>`** permet de créer un tableau dynamique de structures également contigu en mémoire.
- Cette contiguïté rend possible un parcours rapide en mémoire et une compatibilité avec des fonctions C via `points.data()`.

3.6 Organisation mémoire AoS vs SoA

Lorsque l'on manipule des données structurées en grande quantité (par exemple des coordonnées 3D, des particules, des sommets en graphique), il existe deux façons classiques d'organiser les données en mémoire :

Array of Structs (AoS)

C'est la représentation classique avec un **`std::vector<struct>`**. Chaque élément du tableau est une structure complète.

Exemple :

```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}
};
```

Mémoire (chaque `Point3D` = bloc contigu de 12 octets) :

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

Ici, la contiguïté s'applique **au niveau des structures** :

- Les `Point3D` sont rangés dos à dos.
- Chaque `Point3D` lui-même contient ses champs `x, y, z` contigus.

Avantage : pratique pour manipuler un point complet. **Inconvénient :** si l'on ne veut traiter que les `x`, il faut parcourir inutilement les `y` et `z`.

Struct of Arrays (SoA)

Ici, on inverse l'organisation : au lieu de stocker un tableau de structures, on stocke une structure qui contient un tableau par champ.

Exemple :

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

```
};
```

Mémoire (chaque champ est contigu séparément) :

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

Ici, la contiguïté s'applique **au niveau des champs** :

- Tous les `x` sont stockés les uns à la suite des autres.
- Tous les `y` sont contigus, et de même pour les `z`.

Avantage : très efficace si l'on fait un traitement massif sur un seul champ (ex. appliquer une transformation sur toutes les coordonnées `x`). **Inconvénient :** moins naturel si l'on veut travailler sur un point complet (`x,y,z` regroupés).

Contiguïté : deux visions complémentaires

- **AoS** : contiguïté *par objet*. Chaque élément du tableau est un bloc structuré (`{x,y,z}`), et les blocs se suivent.
- **SoA** : contiguïté *par champ*. Chaque champ est regroupé dans son propre tableau, et les valeurs se suivent par dimension.

Les deux approches utilisent donc la **contiguïté mémoire**, mais pas au même niveau de structuration.

Choix en pratique

- **AoS** : souvent préféré quand les données sont manipulées comme des entités indépendantes (ex. liste de particules, objets de jeu, vecteurs 3D dans un moteur physique).
- **SoA** : utilisé en **simulation haute performance**, **calcul scientifique**, **GPU** ou **traitement de données vectorisées**, car il favorise les accès séquentiels optimisés (cache, SIMD).

3.7 Allocation et désallocation mémoire

L'**allocation mémoire** consiste à réserver dynamiquement une zone de mémoire pendant l'exécution du programme, et la **désallocation** consiste à la libérer lorsqu'elle n'est plus nécessaire. Cette gestion dynamique est indispensable lorsque la taille des données n'est pas connue à la compilation ou lorsque leur durée de vie dépasse un bloc local.

En C et en C++, la mémoire dynamique est stockée dans une zone appelée le **tas (heap)**, par opposition à la **pile (stack)** utilisée pour les variables locales.

Pile (stack) vs tas (heap)

Variables sur la pile :

- allocation automatique à l'entrée d'un bloc,
- libération automatique à la sortie du bloc,
- très rapide,
- taille limitée.

```
void f() {
    int x = 10; // sur la pile
}
```

Mémoire dynamique sur le tas :

- allocation explicite par le programmeur,
- durée de vie indépendante des blocs,
- doit être libérée explicitement.

Allocation dynamique en C : `malloc` et `free`

En C, on utilise les fonctions de la bibliothèque standard `<stdlib.h>`.

```
#include <stdlib.h>

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

Ici :

- `malloc` réserve un bloc de mémoire de `sizeof(int)` octets,
- elle retourne un pointeur de type `void*`,
- ce pointeur est converti explicitement en `int*`.

Utilisation :

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

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

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

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

Points importants :

- `malloc` **n'initialise pas** la mémoire,
- `free` doit être appelé **exactement une fois** pour chaque allocation réussie.

Allocation de tableaux dynamiques en C

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

Accès :

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

Libération :

```
free(tab);
```

Allocation dynamique en C++ : `new` et `delete`

En C++, on dispose des opérateurs `new` et `delete`, qui sont **conscients des types** et appellent les constructeurs et destructeurs.

Allocation d'un objet :

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

Libération :

```
delete p;
```

Pour un tableau :

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

Libération correspondante :

```
delete[] tab;
```

Règle fondamentale :

- `new` ↔ `delete`
- `new[]` ↔ `delete[]`

Les mélanger conduit à un **comportement indéfini**.

Allocation d'objets et appel des constructeurs

```
struct Point {
    float x, y;
    Point(float a, float b) : x(a), y(b) {}
};

int main() {
    Point* p = new Point(1.0f, 2.0f); // constructeur appelé
    delete p;                         // destructeur appelé
}
```

Exemple classique d'erreur : fuite mémoire

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

À chaque appel de `f`, la mémoire est allouée mais jamais libérée : **fuite mémoire**.

Double libération (dangereux)

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

Cela provoque un comportement indéfini.

Pointeur nul après libération

Bonne pratique :

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

Cela évite l'accès à un pointeur libéré (*dangling pointer*).

Résumé

- L'allocation dynamique se fait sur le **tas**.
- En C : `malloc` / `free` (mémoire brute, `void*`).
- En C++ : `new` / `delete` (types + constructeurs).
- Toute allocation doit être associée à une désallocation.
- Les erreurs classiques sont : fuites mémoire, double libération, pointeurs pendants.

- En C++ moderne, préférer les conteneurs et les abstractions sûres.

  La gestion manuelle de la mémoire est puissante mais dangereuse. En C++, elle doit être limitée aux cas nécessaires et remplacée autant que possible par des abstractions sûres.

3.8 La copie memoire: memcpy

En C et C++, on a souvent besoin de **copier un bloc d'octets** (tableau, struct, buffer reçu du réseau/fichier, etc.). La fonction standard pour ça est **`memcpy`**, dans `<string.h>` (C) ou `<cstring>` (C++).

Prototype

```
#include <string.h>

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

- `src` : adresse source
- `dest` : adresse destination
- `n` : nombre **d'octets** copiés
- retour : `dest`

Exemple simple : copier un tableau d'entiers

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

int main() {
    int a[3] = {10, 20, 30};
    int b[3] = {0, 0, 0};

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

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

Ici, `memcpy` copie exactement `3 * sizeof(int)` octets.

Exemple : copier une structure simple

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

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

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

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

    printf("%d %d\n", p2.x, p2.y); // 1 2
    return 0;
}
```

Lire un “buffer brut” et reconstruire des types avec `memcpy`

Cas typique : on reçoit un tableau d’octets (réseau, fichier binaire, capteur…) et on veut en extraire des valeurs typées.

Supposons un message binaire au format suivant :

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

Soit : `4 + 4 + 2 = 10` octets.

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

int main() {
    // Buffer brut simulé (par ex. reçu du réseau)
    uint8_t buf[10] = {
        0xD2, 0x04, 0x00, 0x00,   // id = 1234 en little-endian
        0x00, 0x00, 0x48, 0x42,   // float 50.0f en IEEE-754 (little-endian)
        0x07, 0x00                // count = 7 en little-endian
    };

    size_t offset = 0;

    uint32_t id;
    float temp;
    uint16_t count;

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

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

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

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

3.9 Le pointeur générique `void*`

En C et en C++, il existe un type de pointeur particulier : `void*`, appelé **pointeur générique**. Un `void*` peut contenir **l’adresse de n’importe quel type de donnée**, sans connaître sa nature.

Il représente donc une **adresse brute**, sans information de type associée.

Déclaration et principe

```
void* p;
```

Ici :

- `p` peut stocker l’adresse d’un `int`, d’un `float`, d’une `struct`, etc.
- le compilateur **ne sait pas** ce que pointe `p`.

Cela signifie que :

- on peut **stocker une adresse** dans `p`,
- mais on **ne peut pas accéder directement à la valeur pointée**.

Exemple simple

```c
#include <stdio.h>

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

    void* p;

    p = &a;  // p pointe vers un int
    p = &b;  // p pointe maintenant vers un float

    return 0;
}
```

Dans cet exemple :

- `p` peut successivement contenir l'adresse de `a` puis celle de `b`,
- mais **aucune information de type n'est conservée**.

Impossibilité de déréférencer directement

Il est **interdit** de faire :

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

Pourquoi ?

- `*p` signifie « accéder à la valeur pointée »,
- mais le compilateur ne connaît **ni la taille**, ni la nature du type pointé.

Le type `void` signifie littéralement : **absence d'information de type**.

Conversion explicite (cast)

Pour accéder à la valeur pointée, il faut **convertir explicitement** le `void*` vers le bon type de pointeur.

```c
#include <stdio.h>

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

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

    return 0;
}
```

Étapes :

1. `p` contient l'adresse de `a`,
2. on indique explicitement au compilateur : « considère cette adresse comme un `int*` »,
3. on peut alors déréférencer correctement.

Exemple avec plusieurs 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;
}
```

Ici :

- le `void*` permet de passer **n'importe quel type**,
- mais on doit gérer **manuellement** l'interprétation correcte.

Lien avec l'arithmétique des pointeurs

Contrairement aux autres pointeurs (`int*`, `double*`, etc.), **l'arithmétique des pointeurs est interdite sur `void*` en C++**.

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

Raison :

- `p + 1` nécessite de connaître `sizeof(type)`,
- or `void` n'a **pas de taille**.

En C (mais pas en C++), certains compilateurs autorisent `void*` comme une extension non standard, en le traitant comme un `char*`.

`void*` et tableaux / mémoire brute

Le `void*` est souvent utilisé pour manipuler de la **mémoire brute**, par exemple avec `malloc`, `memcpy`, ou des APIs bas niveau.

Exemple :

```
#include <stdlib.h>

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

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

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

Ici :

- `malloc` renvoie un `void*`,
- le programmeur décide ensuite **comment interpréter** cette mémoire.

Exemple plus complet d'utilisation de `void*`

Voici un exemple typique d'utilisation de `void*` : on reçoit un bloc d'octets brut (réseau, fichier, trame capteur, image, …), stocké dans un void*, puis on reconstruit une structure "interprétable".

Imaginons un serveur qui envoie un message binaire composé de :

1. un en-tête (header) avec :
    - `uint32_t id`
    - `uint16_t width`
    - `uint16_t height`
2. puis des données (payload) : ici, par exemple, une image en niveaux de gris de taille `width * height` octets.

On reçoit l'information comme un **buffer brut** (typiquement `void* + taille`) que l'on doit "restructurer".

- Le principe général est le suivant:
    1. **reconnaître** la structure (l'en-tête),
    2. **calculer** où commencent les données utiles,
    3. faire des **casts** (souvent via `uint8_t*` pour faire de l'arithmétique en octets),
    4. vérifier les **tailles** (sinon crash / faille).

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

#pragma pack(push, 1) // pour éviter le padding (dépendant compilateur/ABI)
typedef struct {
    uint32_t id;
    uint16_t width;
    uint16_t height;
} Header;
#pragma pack(pop)

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

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

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

    // --- Reconstruction / interprétation ---
    // 1) Lire l'en-tête
    Header header;
    memcpy(&header, buffer, sizeof(Header));

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

    // 2) Accéder au payload (image) après l'en-tête
    size_t image_size = (size_t)header.width * (size_t)header.height;

    // Vérification minimale de cohérence
    if (sizeof(Header) + image_size > total) {
        printf("Buffer incomplet ou corrompu !\n");
        free(buffer);
        return 1;
    }

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

    // Exemple : afficher les pixels (ligne par ligne)
    for (uint16_t y = 0; y < header.height; ++y) {
        for (uint16_t x = 0; x < header.width; ++x) {
            printf("%3u ", image[y * header.width + x]);
```

```
        }
        printf("\n");
    }

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

Usage en pratique

Le `void*` est principalement utilisé :

- en **C pur** (interfaces génériques, bibliothèques système),
- pour des APIs bas niveau,
- pour manipuler de la mémoire brute,
- dans des fonctions génériques historiques (`qsort`, `bsearch`).

En **C++ moderne**, on préfère :

- les **templates**,
- les **pointeurs typés**,
- les conteneurs (`std::vector`, `std::array`),
- les smart pointers (`std::unique_ptr`, `std::shared_ptr`).

Point clé à retenir

> `void*` est un pointeur sans information de type : il offre une grande flexibilité, mais **aucune sécurité**. Toute utilisation correcte repose sur des **conversions explicites** et la rigueur du programmeur.

3.10 Références

En C++, les **références** sont introduites comme une alternative plus simple et plus sûre aux pointeurs. On peut les voir comme un **alias** vers une variable existante, et surtout comme un **sucre syntaxique** au-dessus de la notion de pointeur :

- Comme un pointeur, une référence permet de travailler directement sur une variable originale sans en faire une copie.
- Contrairement au pointeur, on n'a pas besoin d'écrire $*$ ou `->` : la référence se manipule comme la variable elle-même.

Passage d'arguments : comparaison valeur, pointeur, référence

Passage par valeur (par défaut en C/C++)

```cpp
#include <iostream>

int ma_fonction(int b) {
    b = b + 2;  // modifie la copie locale
    return b;
}

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

Ici :

- `b` est une **copie** de `a`.
- Modifier `b` n'affecte pas `a`.

Passage par adresse avec pointeur (style C)

```
#include <iostream>

void ma_fonction(int* b) {
    *b = *b + 2; // modifie la valeur pointée
}

int main() {
    int a = 5;
    ma_fonction(&a); // on passe 'ladresse de a
    std::cout << a << std::endl; // affiche 7
}
```

Ici :

- `b` est une copie du pointeur vers `a`.
- On doit utiliser `*b` pour accéder/modifier la valeur.
- Syntaxe plus lourde, avec risque d'erreurs (pointeur nul, oubli du `*`).

Passage par référence (style C++)

```
#include <iostream>

void ma_fonction(int& b) {
    b = b + 2; // on a 'limpression de manipuler b comme une variable
}

int main() {
    int a = 5;
    ma_fonction(a); // pas de &
    std::cout << a << std::endl; // affiche 7
}
```

Ici :

- `b` est une **référence** alias de `a`.
- Pas de syntaxe particulière, on manipule `b` comme s'il s'agissait d'une variable locale.
- C'est un *sucre syntaxique* : derrière, le compilateur génère un passage par adresse, mais la syntaxe est simplifiée.

Initialisation des références

Une référence doit toujours être **initialisée** au moment de sa déclaration :

```
int main() {
    int a = 5;
    int& ref_a = a; // OK : ref_a est un alias de a
    ref_a = 9;      // modifie a

    int& ref_b;     // ERREUR : une référence doit être initialisée
}
```

Contrairement à un pointeur, une référence :

- ne peut pas être nulle,
- ne peut pas être réassignée vers une autre variable après initialisation.

Références constantes

Une **référence constante** (`const &`) permet de :

- éviter une copie coûteuse,
- tout en garantissant que l'objet ne sera pas modifié.

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

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

int main() {
    std::string text = "Bonjour";
    printMessage(text); // pas de copie, et sécurité garantie
}
```

Les références constantes sont largement utilisées pour passer des objets volumineux (vecteurs, chaînes, structures) **sans copie**.

Exemple concret : vecteurs et structures

```cpp
#include <iostream>

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

// passage par référence pour modifier
void multiply(vec4& v, double s) {
    v.x *= s; v.y *= s; v.z *= s; v.w *= s;
}

// passage par référence constante pour éviter une copie
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); // modifie v
    print(v);         // affiche sans recopier
}
```

Accesseurs par référence

En C++, les références sont très pratiques pour écrire des accesseurs :

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

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

    // accesseur read/write : retourne une référence
    float& value(unsigned int i) {
        return T[i];
    }
};

int main() {
    Vec50 v;
```

```
    v.init();

    std::cout << v.value(10) << std::endl; // lecture
    v.value(10) = 42;                      // écriture via référence
    std::cout << v.value(10) << std::endl;
}
```

Bonnes pratiques

À faire

- Utiliser les références pour simplifier le code par rapport aux pointeurs.
- Utiliser `const &` pour passer des objets lourds (vecteurs, chaînes, classes).
- Retourner une référence si l'objectif est d'autoriser la modification (accesseur `set`).

À éviter

- Ne pas abuser des références non-const dans les paramètres de fonction → le lecteur doit comprendre immédiatement si une variable est modifiée.
- Ne jamais retourner une référence vers une variable locale (elle n'existe plus après la sortie de la fonction).

Résumé

- Une **référence** est un alias d'une variable.
- Elle est implémentée comme un pointeur, mais avec une syntaxe simplifiée (*syntactic sugar*).
- Les références constantes (`const &`) sont fondamentales pour écrire du code sûr et efficace.
- Bien utilisées, les références combinent la puissance des pointeurs et la lisibilité d'un code clair.

3.11 Allocation dynamique

Jusqu'ici, nous avons vu des **variables automatiques** (déclarées dans une fonction), stockées sur la **pile (stack)** et **détruites automatiquement** à la fin du bloc.

Mais dans certains cas, on a besoin de **données dont la durée de vie dépasse la fin d'un bloc** (par exemple : conserver un tableau créé dans une fonction, gérer de grandes structures, ou construire des graphes dynamiques). Dans ce cas, on utilise la **mémoire dynamique**, allouée sur le **tas (heap)**.

La pile (stack) vs le tas (heap)

Caractéristique	Pile (stack)	Tas (heap)
Allocation	Automatique	Manuelle (ou contrôlée par objets)
Durée de vie	Limitée au bloc courant	Jusqu'à libération explicite
Taille maximale	Limitée (quelques Mo)	Très grande (plusieurs Go)
Gestion	Par le compilateur	Par le programmeur
Exemple	`int a;` ou `int tab[10];`	`new int;` ou `new int[n];`

Sur la plupart des systèmes, la pile a une taille limitée (~8 Mo par défaut), alors que le tas peut utiliser plusieurs gigaoctets. L'allocation dynamique permet donc de **créer des structures volumineuses** ou de **tailles variables** à l'exécution.

Exemple : durée de vie limitée avec variables automatiques

```
#include <iostream>

int* createValue() {
    int a = 42;   // variable locale sur la pile
    return &a;    // ⚠ Dangereux : a est détruit à la fin de la fonction
}
```

```cpp
int main() {
    int* p = createValue();
    std::cout << *p << std::endl; // comportement indéfini !
}
```

`a` est détruit à la sortie de `createValue()`. Le pointeur retourné devient **dangling** (dangereux).

Exemple : durée de vie prolongée avec allocation dynamique

```cpp
#include <iostream>

int* createValue() {
    int* p = new int(42); // alloué sur le tas
    return p;             // valide même après la fin de la fonction
}

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

Ici, la variable `*q` persiste après la fin de `createValue()`. Mais **le programmeur doit libérer la mémoire** avec `delete`.

Allocation dynamique d'un tableau

```cpp
#include <iostream>

int* createArray(int n) {
    int* arr = new int[n]; // allocation de n entiers
    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; // libération obligatoire
}
```

Utilité : `n` est connu uniquement à l'exécution → impossible d'utiliser un tableau statique.

Schéma mémoire

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

- La pile contient les variables locales (`n`, `arr`).
- Le tas contient les données allouées dynamiquement.
- La mémoire du tas n'est pas libérée automatiquement → `delete[] arr;` obligatoire.

Problèmes classiques

1. **Fuite mémoire :**

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

   → la mémoire reste occupée tant que le programme tourne.

2. **Double libération :**

```
int* p = new int(5);
delete p;
delete p; // erreur : libération double
```

3. **Utilisation après libération :**

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

Exemple : redimensionnement (principe)

Quand on redimensionne un tableau dynamique manuellement, il faut :

1. Allouer un nouvel espace.
2. Copier les anciennes données.
3. Libérer l'ancien espace.

```
Ancien tableau (@100) : [10 20 30]
Nouveau tableau (@320) : [10 20 30 40]
delete[] @100
```

Note: Le réallongement d'un tableau demande toujours une **nouvelle allocation + copie**, d'où le coût.
Les conteneurs modernes (`std::vector`) automatisent ce processus efficacement.

Structures dynamiques : listes et graphes

L'allocation dynamique permet aussi de créer des structures **chaînées** ou **hiérarchiques**, où chaque élément contient des pointeurs vers d'autres.

Exemple : liste chaînée minimale

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

int main() {
    Node* n1 = new Node{5, nullptr};
    Node* n2 = new Node{8, nullptr};
    // Remarque : l'opérateur `->` permet d'accéder à un membre via un pointeur.
    // `p->membre` est équivalent à `(*p).membre`.
    n1->next = n2;

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

    // libération
    delete n2;
```

```
    delete n1;
}
```

Chaque élément (`Node`) est alloué séparément sur le tas. [**Attention**]: Il faut penser à **libérer chaque élément** pour éviter les fuites.

Bonnes pratiques modernes

En C++, on évite aujourd'hui `new` / `delete` directs. On privilégie :

1. `std::vector` pour les tableaux dynamiques

Exemple:

```
#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; // gestion automatique
}

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

→ La mémoire est gérée automatiquement (constructeur / destructeur).

2. Pointeurs intelligents (`std::unique_ptr`, `std::shared_ptr`)

Les **pointeurs intelligents** sont des classes de la bibliothèque standard C++ (`<memory>`) qui encapsulent un pointeur brut (`T*`) et **gèrent automatiquement la durée de vie de la ressource** pointée.

Ils suivent le principe du **RAII** : la ressource est libérée automatiquement quand le pointeur sort de portée (destruction de l'objet). Ainsi, plus besoin d'appeler `delete` manuellement : la mémoire est libérée dès que l'objet n'est plus utilisé.

Exemple avec `std::unique_ptr` Exemple:

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

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

Explication :

- `std::unique_ptr<int>` possède l'exclusivité de la ressource : un seul pointeur gère l'objet alloué.
- `std::make_unique<int>(42)` crée dynamiquement un `int` contenant `42` et renvoie un `unique_ptr` qui en devient propriétaire.
- Quand `p` sort de portée (fin du `main`), son destructeur appelle **automatiquement** `delete` sur l'objet qu'il gère.
- La mémoire est donc proprement libérée, même en cas d'exception ou de sortie prématurée de la fonction.

Caractéristiques de `std::unique_ptr` :

- Ownership *unique* (non copiable).
- Léger, sûr et très performant.
- Idéal pour représenter la possession exclusive d'une ressource.

Exemple avec `std::shared_ptr` Exemple:

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

int main() {
    auto p1 = std::make_shared<int>(10);
    auto p2 = p1; // partage de la ressource
    std::cout << *p2 << std::endl;
} // mémoire libérée quand le dernier shared_ptr disparaît
```

Explication détaillée :

- `std::shared_ptr` permet à **plusieurs pointeurs** de partager la même ressource.
- Chaque copie (`p2 = p1;`) **augmente un compteur de référence** interne.
- Lorsqu'un `shared_ptr` est détruit, le compteur est décrémenté.
- Quand ce compteur atteint zéro (plus aucun propriétaire), le destructeur appelle **`delete` automatiquement** sur la ressource.

Ainsi, la mémoire est libérée exactement quand elle n'est plus utilisée par personne.
Caractéristiques de `std::shared_ptr` :

- Copiable : plusieurs instances peuvent pointer vers la même donnée.
- Référence comptée : destruction automatique quand le dernier propriétaire disparaît.
- Légèrement plus coûteux qu'un `unique_ptr` (compteur atomique interne).
- Idéal pour des structures partagées ou des graphes non hiérarchiques.

Comparaison des deux types de pointeurs intelligents

Type	Copiable	Partage de ressource	Destruction	Cas d'usage typique
`std::unique_ptr<T>`	Non	Non	Automatique, dès que le pointeur sort de portée	Possession exclusive (ex. composant interne d'un objet)
`std::shared_ptr<T>`	Oui	Oui (compteur de références)	Automatique, quand le dernier pointeur est détruit	Ressources partagées entre plusieurs objets ou fonctions

---

Illustration mémoire

```
Cas unique_ptr :
+---------------------+
| unique_ptr<int> p   ──►| [42]
+---------------------+
          |
    delete automatique à la fin du bloc


Cas shared_ptr :
+---------------------+        +---------------------+
| shared_ptr<int> p1  ──┐|    | compteur = 2        |
| shared_ptr<int> p2  ──┘──►| [10]
+---------------------+        +---------------------+
                      |
      delete automatique quand compteur = 0
```

Pourquoi les pointeurs intelligents remplacent `new` et `delete`

- Ils **évitent les fuites mémoire** en gérant la libération automatiquement.
- Ils **préservent la sécurité** (pas de double libération ni de pointeur pendu).
- Ils **simplifient le code** : plus besoin d'appeler `delete`.
- Ils s'intègrent naturellement avec les autres classes du C++ moderne (`std::vector`, `std::map`, `std::thread`, etc.).

4 Classes

4.1 Introduction

En C++, une **classe** permet de regrouper dans une même entité des **données** (appelées *attributs*) et des **fonctions** (appelées *méthodes*) qui manipulent ces données. Une instance d'une classe est appelée un **objet**. Cette organisation facilite la structuration du code, sa lisibilité et sa maintenance.

Regrouper des données : premier exemple avec `struct`

On commence souvent par une `struct` pour représenter un objet simple :

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

Ici, `vec3` regroupe trois valeurs représentant un vecteur 3D. Les membres sont **publics par défaut**, ce qui signifie qu'ils sont accessibles directement :

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

Ce type de structure est bien adapté pour des **agrégats de données simples**, très fréquents en informatique graphique.

Ajouter un comportement : méthodes

Une classe ou une struct peut aussi contenir des **fonctions membres** :

```
#include <cmath>

struct vec3 {
    float x, y, z;

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

La méthode `norm()` opère directement sur les attributs `x`, `y` et `z` de l'objet :

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

Remarque : le `const` placé après la signature d'une méthode (ici `norm() const`) indique que la méthode **ne modifie pas l'état de l'objet**. Une méthode `const` peut être appelée sur un objet `const`, et le compilateur interdit toute modification des membres non `mutable` à l'intérieur de cette méthode.

Le pointeur implicite `this`

Dans les méthodes d'une classe, le compilateur fournit **implicitement** un pointeur nommé `this` qui pointe vers l'objet courant. Il est utile pour accéder explicitement aux membres, désambiguïser des paramètres et retourner une référence sur l'objet.

Exemple :

```
struct S {
        int x;
        void set(int x) { this->x = x; }      // désambiguïse le champ x
```

```
        int get() const { return this->x; }   // this est const
};
```

Cette notion est basique mais importante : `this` permet de manipuler l'objet courant à l'intérieur des méthodes et rend explicite certaines opérations (transfert de *ownership*, retour de `*this`, …).

`struct` vs `class`

Le mot-clé `class` fonctionne exactement comme `struct`, à la différence que : les membres sont **privés par défaut**.

```
class vec3 {
    float x, y, z; // privés par défaut
};
```

Ce code **ne compile pas** :

```
vec3 v;
v.x = 1.0f; // ERREUR : x est privé
```

Pour rendre certains membres accessibles, il faut préciser les niveaux d'accès.

Attributs publics et privés

On utilise les mots-clés `public` et `private` pour contrôler l'accès aux membres :

```
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;
};
```

Utilisation :

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

float n = v.norm(); // OK
// v.x = 3.0f;      // ERREUR : x est privé
```

Ici :

- les **attributs sont privés** → protégés contre des modifications incontrôlées,
- les **méthodes sont publiques** → interface accessible à l'utilisateur.

Encapsulation et sécurité

Grâce à cette encapsulation, l'objet garantit sa cohérence interne. Par exemple, on peut forcer certaines règles :

```
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;
};
```

Ici, le rayon ne peut jamais devenir négatif, car l'accès direct à `r` est interdit.

Bonnes pratiques

- Utiliser **`struct`** pour :
  - des objets simples,
  - principalement porteurs de données,
  - sans invariants complexes.
- Utiliser **`class`** pour :
  - encapsuler des données,
  - contrôler les accès,
  - garantir des invariants internes.

4.2 Initialization, constructeurs

En C++, l'initialisation d'un objet est prise en charge par les **constructeurs**. Un constructeur est une fonction spéciale (même nom que la classe, pas de type de retour) appelée automatiquement à la création de l'objet. Son but est de garantir que l'objet est dans un **état valide** dès le départ.

Problème classique : attributs non initialisés

Si une classe/struct contient des types fondamentaux (**`int`**, **`float`**, etc.), ils ne sont pas forcément initialisés automatiquement.

```
#include <iostream>

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

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

Dans le cas d'une `struct` agrégée, on peut forcer une initialisation à zéro avec `{}` :

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

Mais dès qu'on veut contrôler précisément l'état de l'objet, on utilise des constructeurs.

Constructeur par défaut

Le constructeur par défaut ne prend aucun argument. Il sert souvent à mettre des valeurs cohérentes.

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

    vec3() : x(0.0f), y(0.0f), z(0.0f) {}
};

int main() {
    vec3 v; // appelle vec3()
}
```

Ici, `v` est garanti valide : ses champs valent 0.

Liste d'initialisation

L'écriture : `x(...), y(...), z(...)` est la liste d'initialisation. Elle initialise les attributs avant d'entrer dans le corps du constructeur.

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

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

Utilisation :

```
vec3 v(1.0f, 2.0f, 3.0f);
vec3 w{1.0f, 2.0f, 3.0f}; // uniforme (souvent recommandé)
```

Cette liste est préférable à une affectation dans le corps du constructeur, car elle évite une "double étape" (construction puis réaffectation) et elle est requise pour certains membres.

Constructeurs surchargés

On peut définir plusieurs constructeurs pour offrir différentes manières de créer un objet.

```
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)
}
```

Constructeur à un argument et `explicit`

Un constructeur à un seul argument peut servir de conversion implicite, ce qui peut provoquer des effets de bord. Le mot-clé `explicit` empêche ces conversions automatiques.

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

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

```
vec3 a(1.0f);   // OK
// vec3 b = 1.0f; // interdit grâce à explicit
```

Cela rend le code plus sûr et plus lisible.

Membres `const` et références : constructeur obligatoire

Les attributs `const` et les références doivent être initialisés via la liste d'initialisation.

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

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

Sans liste d'initialisation, ce code ne compile pas, car `id` et `ref` ne peuvent pas être "assignés" après coup : ils doivent être initialisés immédiatement.

Destructeur (rappel)

Le destructeur est appelé automatiquement quand l'objet est détruit (fin de scope, `delete`, etc.). Il sert surtout à libérer des ressources (fichier, mémoire, GPU…).

```
#include <iostream>

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

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

Bonnes pratiques

- Initialiser systématiquement les attributs (via constructeur ou `{}`).
- Préférer la liste d'initialisation `:` pour initialiser les membres.
- Utiliser `explicit` pour les constructeurs à un argument, sauf si la conversion implicite est désirée.
- Concevoir les constructeurs pour garantir des objets toujours valides.

4.3 Opérateurs

En C++, il est possible de **surcharger des opérateurs** pour des classes et des structures afin de rendre leur utilisation plus naturelle et expressive. Cette fonctionnalité est particulièrement utile en informatique graphique, où l'on manipule fréquemment des vecteurs, matrices, couleurs ou transformations, et où des expressions comme `v1 + v2` ou `2.0f * v` sont bien plus lisibles qu'un appel de fonction explicite.

Principe général

La surcharge d'opérateurs consiste à définir une **fonction spéciale** dont le nom est `operator<symbole>`. Du point de vue du compilateur, une expression comme :

```
a + b
```

est traduite en :

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

ou, dans le cas d'un opérateur membre :

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

La surcharge ne crée **pas de nouvel opérateur** : elle redéfinit simplement le comportement d'un opérateur existant pour un type donné.

Opérateurs membres et non-membres

Un opérateur peut être défini :

- comme **méthode membre** de la classe,
- ou comme **fonction non-membre** (souvent préférable pour les opérateurs symétriques).

Règle courante :

- les opérateurs qui **modifient l'objet** (`+=`, `*=`, `[]`, etc.) sont souvent des méthodes membres ;
- les opérateurs binaires symétriques (`+`, `-`, `*`) sont souvent des fonctions non-membres.

Exemple : opérateurs arithmétiques pour un vecteur 3D

```
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;
    }
};
```

L'opérateur `+=` modifie l'objet courant et retourne une référence sur celui-ci.
On définit ensuite `+` comme opérateur non-membre en réutilisant `+=` :

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

Utilisation :

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

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

Opérateurs avec types différents

On peut définir des opérateurs entre types différents, par exemple la multiplication par un scalaire :

```
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;
}
```

Cela permet une écriture naturelle :

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

Opérateurs de comparaison

Les opérateurs de comparaison permettent de comparer des objets :

```
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);
}
```

Depuis C++20, il existe également l'opérateur `<=>` (three-way comparison), mais son utilisation dépasse le cadre de cette introduction.

Opérateur d'accès `[]`

L'opérateur `[]` est souvent utilisé pour donner un accès indexé aux données internes :

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

    float& operator[](int i) {
        return (&x)[i]; // accès contigu
    }

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

Utilisation :

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

La version `const` est indispensable pour permettre l'accès en lecture sur un objet constant.

Opérateur d'affichage <<

Pour faciliter le débogage, on surcharge souvent l'opérateur `<<` avec `std::ostream` :

```
#include <iostream>

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

Utilisation :

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

Bonnes pratiques

- Toujours utiliser des **références constantes** pour les paramètres en lecture.
- Retourner `*this` par référence pour les opérateurs de modification (`+=`, `*=`, etc.).
- Éviter les surcharges qui rendent le code ambigu ou contre-intuitif.
- Ne pas surcharger un opérateur si son sens mathématique ou logique n'est pas clair.

La surcharge d'opérateurs permet d'écrire du code plus lisible et plus expressif, mais elle doit rester **simple, cohérente et prévisible**.

4.4 Héritage

L'**héritage** est un mécanisme central de la programmation orientée objet qui permet de **définir une nouvelle classe à partir d'une classe existante**. La classe dérivée hérite des attributs et des méthodes de la classe de base, ce qui favorise la **réutilisation du code** et la structuration hiérarchique des concepts. En C++, l'héritage est souvent utilisé pour factoriser des comportements communs tout en permettant des spécialisations.

Principe général

On définit une classe dérivée en indiquant la classe de base après `:` :

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

```
};
```

Le mot-clé `public` indique que l'interface publique de la classe de base reste publique dans la classe dérivée. C'est le cas le plus courant et celui utilisé dans la majorité des conceptions orientées objet.

Exemple simple d'héritage

Considérons une classe de base représentant une forme géométrique :

```
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;
    }
};
```

On peut définir une classe dérivée qui spécialise ce comportement :

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

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

Utilisation :

```
Circle c(0.0f, 0.0f, 1.0f);
c.translate(1.0f, 2.0f); // méthode héritée de Shape
```

La classe `Circle` hérite automatiquement de `x`, `y` et de la méthode `translate`.

Constructeurs et héritage

Le constructeur de la classe dérivée **doit appeler explicitement** le constructeur de la classe de base dans sa liste d'initialisation.

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

Si le constructeur de la classe de base n'est pas appelé explicitement, le compilateur tentera d'appeler le constructeur par défaut, ce qui peut entraîner une erreur s'il n'existe pas.

Accès aux membres : `public`, `protected`, `private`

Le niveau d'accès des membres de la classe de base détermine leur visibilité dans la classe dérivée :

- `public` : accessible partout, y compris dans les classes dérivées.
- `protected` : accessible uniquement dans la classe et ses dérivées.
- `private` : accessible uniquement dans la classe de base.

Exemple :

```
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
    }
};
```

Redéfinition de méthodes

Une classe dérivée peut **redéfinir** une méthode de la classe de base afin de fournir un comportement spécifique.

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

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

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

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

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

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

Ici, `Rectangle::area` masque la version définie dans `Shape`. Ce mécanisme prépare naturellement l'introduction du **polymorphisme**, qui sera étudié dans le chapitre suivant.

Héritage et factorisation du code

L'héritage permet d'éviter les duplications :

```
class Vehicle {
  public:
    float speed;

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

class Car : public Vehicle {
    // comportement spécifique
};

class Plane : public Vehicle {
    // comportement spécifique
};
```

Les classes `Car` et `Plane` partagent le même comportement de base sans duplication.

Bonnes pratiques

- Utiliser l'héritage pour exprimer une relation **est-un** (*is-a*).
- Préférer des classes de base **simples et stables**.

4.5 Polymorphisme

Le **polymorphisme** permet de manipuler des objets de types différents **à travers une interface commune**, tout en appelant automatiquement la bonne implémentation selon le **type réel** de l'objet. En C++, il repose sur l'héritage, les **fonctions virtuelles** et l'utilisation de **pointeurs ou références** vers une classe de base. Il est particulièrement utile lorsqu'on souhaite **stocker des objets hétérogènes dans un même conteneur** et les traiter de manière uniforme.

Le problème : stocker des objets différents dans un même conteneur

Supposons que l'on souhaite représenter différentes formes géométriques et calculer leur aire totale.

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

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

Ces deux types possèdent une méthode `area()`, mais **ils n'ont aucun lien de type**. Il est donc impossible d'écrire :

```
std::vector<Circle> shapes;    // uniquement des cercles
std::vector<Rectangle> shapes; // uniquement des rectangles
```

et surtout impossible de faire :

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

Sans polymorphisme, on est contraint soit :

- de dupliquer le code,
- d'utiliser des tests sur le type,
- ou de concevoir une structure artificielle regroupant tous les cas possibles.

Le polymorphisme fournit une solution élégante à ce problème.

Interface commune via une classe de base

On commence par définir une **classe de base** représentant le concept général de "forme" :

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

Cette classe est **abstraite** :

- elle définit une interface,
- elle ne peut pas être instanciée.

Classes dérivées spécialisées

Chaque forme concrète hérite de `Shape` et implémente `area()` :

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

    explicit Circle(float r_) : r(r_) {}

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

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

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

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

Stockage polymorphique dans un conteneur

Grâce à l'héritage et aux fonctions virtuelles, on peut maintenant stocker des **pointeurs vers la classe de base** dans un même conteneur :

```
#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(); // appel polymorphique
    }
}
```

Ici :

- le conteneur ne connaît que le type `Shape`,
- chaque élément pointe vers un objet d'un type concret différent,
- l'appel à `area()` est résolu **dynamiquement** selon le type réel (`Circle` ou `Rectangle`).

Rôle de `virtual` et du dispatch dynamique

L'appel :

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

est résolu à l'exécution grâce à la **table virtuelle** :

- si `s` pointe vers un `Circle`, `Circle::area()` est appelée,
- si `s` pointe vers un `Rectangle`, `Rectangle::area()` est appelée.

C'est le cœur du polymorphisme dynamique.

Importance du destructeur virtuel

Les objets sont détruits via un pointeur vers la classe de base. Le destructeur doit donc être virtuel :

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

Sans cela, le destructeur de la classe dérivée ne serait pas appelé, ce qui pourrait provoquer des fuites de ressources.

Pourquoi des pointeurs et pas des objets ?

On ne peut pas stocker directement des objets dérivés dans un conteneur de type `std::vector<Shape>` car cela entraînerait un **slicing** (perte de la partie dérivée). Les pointeurs (souvent intelligents) évitent ce problème et permettent la liaison dynamique.

Coût et alternatives

Le polymorphisme dynamique implique :

- une indirection,
- un coût d'appel légèrement supérieur à une fonction non virtuelle.

Dans des boucles très critiques en performance, on privilégiera parfois le **polymorphisme statique** via les templates, abordé ultérieurement.

Utilisation de pointeurs bruts (*raw pointers*)

Dans les exemples précédents, nous avons utilisé des **pointeurs intelligents** (`std::unique_ptr`) pour gérer automatiquement la durée de vie des objets. Il est toutefois important de comprendre que le polymorphisme en C++ fonctionne historiquement avec des **pointeurs bruts** (`Shape*`). Ceux-ci offrent plus de liberté, mais exigent une **gestion manuelle de la mémoire**, ce qui augmente fortement le risque d'erreurs.

Exemple avec pointeurs bruts

```
#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(); // appel polymorphique
    }

    // Libération manuelle de la mémoire
    for (Shape* s : shapes) {
        delete s;
    }
}
```

Ici :

- les objets sont alloués dynamiquement avec `new`,
- le conteneur stocke des pointeurs vers la classe de base `Shape`,
- les appels à `area()` sont résolus dynamiquement,
- **le programmeur doit impérativement libérer la mémoire** avec `delete`.

Rôle critique du destructeur virtuel

Avec des pointeurs bruts, le destructeur virtuel est absolument indispensable :

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

Sans destructeur virtuel, l'appel :

```
delete s;
```

ne détruirait que la partie `Shape` de l'objet, et non la partie dérivée (`Circle`, `Rectangle`), entraînant des fuites de ressources et un comportement indéfini.

Problèmes fréquents avec les pointeurs bruts

L'utilisation de pointeurs bruts expose à plusieurs erreurs classiques :

- oubli de `delete` → **fuite mémoire** ;
- double `delete` → **comportement indéfini** ;
- suppression dans le mauvais ordre ;
- exception ou retour anticipé empêchant la libération ;
- confusion sur la responsabilité de destruction.

Ces problèmes sont difficiles à détecter et à corriger, en particulier dans des projets de grande taille.

Bonnes pratiques

- Utiliser le polymorphisme pour résoudre des problèmes de **traitement uniforme d'objets hétérogènes**.
- Définir des classes de base abstraites comme interfaces.
- Toujours déclarer un destructeur virtuel dans une hiérarchie polymorphique.
- Utiliser `override` pour sécuriser les redéfinitions.
- Combiner polymorphisme et pointeurs intelligents (`std::unique_ptr`).

Le polymorphisme permet ainsi de concevoir des systèmes extensibles où de nouveaux types peuvent être ajoutés sans modifier le code existant, en particulier lorsqu'il s'agit de manipuler des collections d'objets variés.

4.6 Gestion d'accès : `const`

En C++, le mot-clé **`const`** appliqué aux **méthodes de classe** joue un rôle central dans la gestion des accès et dans la sécurité du code. Il ne s'agit pas d'un simple indicateur documentaire : une méthode `const` et une méthode non `const` sont considérées par le compilateur comme **deux méthodes différentes**, pouvant parfaitement **cohabiter dans une même classe avec le même nom**.

Sens d'une méthode `const`

Une méthode déclarée avec `const` après sa signature garantit qu'elle **ne modifie pas l'état de l'objet**.

```
class vec3 {
  public:
    float x, y, z;

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

Le `const` signifie ici que la méthode ne peut pas modifier `x`, `y` ou `z`. Toute tentative de modification provoquerait une erreur de compilation.

```
float norm() const {
    x = 0.0f; // ERREUR : modification interdite
    return 0.0f;
}
```

Objets constants et méthodes accessibles

Un objet déclaré `const` ne peut appeler **que des méthodes `const`**.

```
const vec3 v{1.0f, 2.0f, 3.0f};

v.norm();     // OK
// v.normalize(); // ERREUR si normalize() n'est pas const
```

Cela impose naturellement une séparation claire entre :

- les méthodes de **lecture** (accès, calculs),
- les méthodes de **modification** (mise à jour de l'état).

Méthodes `const` et non `const` : deux signatures différentes

Une méthode `const` et une méthode non `const` portant le même nom **ne sont pas la même fonction**. Elles peuvent être définies simultanément dans une classe.

```
class vec3 {
  public:
    float x, y, z;

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

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

Ici :

- la version **non `const`** est appelée sur un objet modifiable,
- la version **`const`** est appelée sur un objet constant.

Utilisation :

```
vec3 a{1,2,3};
a[0] = 5.0f; // appelle la version non const

const vec3 b{1,2,3};
float x = b[0]; // appelle la version const
```

Le compilateur choisit automatiquement la version appropriée en fonction du **caractère const de l'objet**.

Exemple classique : accesseur en lecture et écriture

```
class Buffer {
  public:
    float& value() {
        return data;
    }

    float value() const {
        return data;
    }
```

```
  private:
    float data;
};
```

Ici :

- `value()` (non `const`) permet de modifier la donnée,
- `value() const` permet seulement de la lire.

```
Buffer b;
b.value() = 3.0f; // version non const

const Buffer c;
// c.value() = 3.0f; // ERREUR
float v = c.value(); // version const
```

Intérêt conceptuel

Cette distinction permet :

- d'exprimer clairement les intentions du code,
- de garantir que certaines opérations sont sans effet de bord,
- de détecter des erreurs dès la compilation,
- d'écrire des interfaces plus robustes.

Dans une conception bien structurée, la majorité des méthodes devraient être `const`. Les méthodes non `const` correspondent à des **opérations de modification explicites**.

Bonnes pratiques

- Marquer toute méthode qui ne modifie pas l'objet comme `const`.
- Fournir systématiquement une version `const` et non `const` lorsque l'accès peut être en lecture ou en écriture.
- Considérer une méthode `const` et une méthode non `const` comme **deux contrats distincts**.
- Utiliser `const` comme un outil de conception, pas seulement comme une contrainte syntaxique.

4.7 Mot clé : `static`

4.8 Gestion d'accès : le mot-clé `static` dans les classes

Le mot-clé **`static`**, appliqué aux membres d'une classe, modifie profondément leur **nature** et leur **durée de vie**. Un membre `static` n'appartient **pas à un objet**, mais à la **classe elle-même**. Il est donc **partagé par toutes les instances** de cette classe. Ce mécanisme est essentiel pour représenter des données ou des comportements globaux liés à un concept, plutôt qu'à un objet particulier.

Attributs statiques

Un **attribut statique** est unique pour toute la classe, quel que soit le nombre d'objets créés.

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

    static int get_count() {
        return count;
    }

  private:
    static int count;
};
```

La déclaration dans la classe **ne suffit pas**. L'attribut statique doit être **défini une seule fois** dans un fichier `.cpp` :

```
int Counter::count = 0;
```

Utilisation :

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

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

Tous les objets `Counter` partagent la **même variable** `count`.

Accès aux attributs statiques

Un attribut statique :

- peut être accédé **sans objet**, via le nom de la classe,
- peut aussi être accédé depuis un objet, mais ce n'est pas recommandé.

```
Counter::get_count(); // forme recommandée
```

Cela souligne le fait que la donnée appartient à la classe, et non à une instance particulière.

Méthodes statiques

Une **méthode statique** est une fonction associée à la classe, mais indépendante de toute instance.

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

Utilisation :

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

Contraintes des méthodes statiques

Une méthode statique :

- n'a **pas de pointeur `this`**,
- ne peut accéder qu'aux **membres statiques** de la classe,
- ne peut pas accéder directement aux attributs non statiques.

```
class Example {
  public:
    static void f() {
        // x = 3; // ERREUR : x n'est pas statique
        y = 4;    // OK
    }

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

`static` et initialisation

Depuis C++17, il est possible d'initialiser directement certains attributs statiques dans la classe s'ils sont `constexpr` ou de type littéral.

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

Utilisation :

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

Dans ce cas, aucune définition supplémentaire dans un `.cpp` n'est nécessaire.

Cas d'usage courants

Le mot-clé `static` est utilisé pour :

- compter le nombre d'instances d'une classe,
- stocker des constantes globales liées à un concept,
- partager des ressources communes,
- regrouper des fonctions utilitaires liées à une classe,
- implémenter des fabriques (*factory methods*).

Exemple : identifiant unique par objet

```
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;
```

Chaque objet reçoit un identifiant unique, généré à partir d'un compteur partagé.

Bonnes pratiques

- Utiliser `static` pour exprimer une **appartenance à la classe**, pas à l'objet.
- Accéder aux membres statiques via `NomClasse::membre`.
- Limiter l'usage des attributs statiques modifiables pour éviter les dépendances cachées.
- Préférer `constexpr static` pour les constantes connues à la compilation.

Idée clé à retenir

Un membre `static` est **unique et partagé**, il appartient à la **classe**, pas aux objets.

4.9 Espaces de noms (`namespace`)

Quand un projet grandit, il devient fréquent d'avoir des **noms identiques** dans des parties différentes du code : `vec3`, `add`, `normalize`, `load`, etc. En C++, un **espace de noms** (*namespace*) permet de **regrouper** des fonctions, types et constantes sous un préfixe commun, afin de :

- **éviter les conflits de noms** entre modules/bibliothèques,

- **structurer** le code par domaines (math, io, gpu, …),
- rendre l'API plus **claire** et plus **prédictible**.

L'exemple le plus connu est la bibliothèque standard : `std::vector`, `std::string`, `std::cout`.

Déclaration et utilisation

Un `namespace` crée une "boîte" logique :

```
namespace math {

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

float dot(vec3 const& a, vec3 const& b)
{
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

} // namespace math
```

Utilisation :

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

float p = math::dot(a, b);
```

Ici, `math::` est le **qualificateur** : il désambiguïse les symboles.

Exemple : éviter un conflit de noms

Deux bibliothèques peuvent proposer une fonction `load()` mais pour des usages différents. Sans namespace, cela devient ambigu.

```
namespace io {
    int load(char const* filename) { /* ... */ return 0; }
}

namespace gpu {
    int load(char const* shader_file) { /* ... */ return 1; }
}
```

Usage explicite et non ambigu :

```
int a = io::load("mesh.obj");
int b = gpu::load("shader.vert");
```

`using` : importer des noms (avec prudence)

Il existe deux syntaxes :

1) Importer un nom précis (recommandé)

```
using math::vec3;

vec3 v{1,2,3}; // équivalent à math::vec3
```

2) Importer tout un namespace (à éviter dans un header)

```cpp
using namespace std;
```

Cela permet d'écrire `vector` au lieu de `std::vector`, mais peut créer des conflits.
Bonne pratique :

- `using namespace ...;` est acceptable dans un petit `.cpp` local,
- **à éviter dans un `.hpp`**, car il pollue tous les fichiers qui incluent cet en-tête.

Espaces de noms imbriqués

On peut structurer par modules :

```cpp
namespace engine {
namespace math {
    struct vec2 { float x, y; };
}
namespace io {
    void save();
}
}
```

Depuis C++17, on peut écrire plus simplement :

```cpp
namespace engine::math {
    struct vec2 { float x, y; };
}
```

Espaces de noms anonymes (visibilité locale)

Un namespace anonyme rend les symboles **visibles uniquement dans le fichier courant** (équivalent à `static` pour des fonctions globales, mais plus général).

```cpp
namespace {
    int helper(int x) { return 2*x; }
}

int f(int a)
{
    return helper(a);
}
```

Intérêt :

- éviter d'exposer des fonctions internes au reste du projet,
- limiter la surface de l'API publique.

Alias de namespace

Utile si un nom est long :

```cpp
namespace em = engine::math;

em::vec2 v{1,2};
```

Bonnes pratiques

- Utiliser des namespaces pour structurer un projet (ex. `engine::math`, `engine::io`, `engine::render`).
- Garder les `using namespace ...;` hors des headers.
- Préférer `using nom::symbole;` plutôt qu'importer tout le namespace.
- Utiliser un namespace anonyme pour les détails d'implémentation dans un `.cpp`.
- Concevoir une API publique stable via un namespace clair (ex. `myproject::`).

Si tu veux, je peux aussi te proposer une mini-convention d'organisation "type projet" (ex. `namespace cgp` / `namespace csc43043`, structure fichiers, exposer uniquement `include/` vs `src/`) pour rendre l'ensemble homogène avec les chapitres sur l'organisation des fichiers.

5 Threads et parallélisme

Le **parallélisme** désigne la capacité d'un programme à exécuter **plusieurs tâches simultanément**. En C++, cette notion est directement liée aux **threads**, qui permettent d'exploiter les **cœurs multiples** des processeurs modernes. Comprendre les threads est essentiel pour écrire des programmes performants, mais aussi sûrs et corrects.

5.1 Notion de thread

Un **thread** est un **fil d'exécution** indépendant à l'intérieur d'un même programme.

- Un programme classique possède **un seul thread** (exécution séquentielle).
- Un programme multithread possède **plusieurs threads**, exécutés en parallèle ou quasi-parallèle.

Tous les threads d'un même programme :

- partagent le **même espace mémoire** (heap, variables globales),
- possèdent chacun leur **pile d'exécution** (variables locales, appels de fonctions).

(Petit rappel : en C++ on manipule souvent les threads via la classe `std::thread` fournie dans `<thread>`.)

5.2 Création d'un thread en C++

Depuis C++11, la bibliothèque standard fournit `std::thread`.
(`std::thread` : classe qui représente un fil d'exécution et permet de lancer une fonction dans un thread séparé ; définie dans `<thread>`.)
Exemple simple :

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

void task() {
    std::cout << "Hello depuis un thread" << std::endl;
}

int main() {
    std::thread t(task); // création du thread
    t.join();            // attendre la fin du thread
    return 0;
}
```

Points importants :

- le thread démarre **immédiatement** à sa création,
- `join()` bloque le thread principal jusqu'à la fin du thread `t`,
- `detach()` dissocie le thread du thread appelant : il devient indépendant et n'est plus joignable,
- ne pas appeler `join()` ou `detach()` avant la destruction d'un objet `std::thread` provoque `std::terminate()` à l'exécution.

Dans cet exemple :

- `task()` s'exécute dans un **thread séparé**,
- le thread principal attend la fin de `t` grâce à `join()`.

5.3 Exemple d'exécution parallèle

Considérons maintenant deux threads exécutant une tâche visible dans le temps.

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

void task(int id) {
    for(int i = 0; i < 5; ++i) {
        std::cout << "Thread " << id << " : étape " << 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;
}
```

(Remarque : `std::chrono` (dans `<chrono>`) fournit des types pour durées et horloges, p.ex. `milliseconds`.)
Sortie typique (l'ordre exact peut varier) :

```
Thread 1 : étape 0
Thread 2 : étape 0
Thread 1 : étape 1
Thread 2 : étape 1
Thread 2 : étape 2
Thread 1 : étape 2
Thread 1 : étape 3
Thread 2 : étape 3
Thread 2 : étape 4
Thread 1 : étape 4
```

Ce que l'on observe :

- les deux threads **progressent en même temps**,
- leurs affichages sont **entrelacés**,
- l'ordre n'est **pas déterministe**.

5.4 Passage d'arguments aux threads

Les arguments sont copiés par défaut.

```cpp
void print(int x) {
    std::cout << x << std::endl;
}

std::thread t(print, 42);
t.join();
```

En suivant le format générique de passage d'arguments.

```cpp
std::thread t(fonction, arg1, arg2, arg3, ...);
```

Pour passer une référence :

```cpp
#include <functional>

void increment(int& x) {
    x++;
}

int main() {
    int a = 5;
    std::thread t(increment, std::ref(a));
```

```
    t.join();
}
```

5.5 Threads multiples et parallélisme réel

Exemple avec plusieurs threads :

```
#include <thread>
#include <vector>

void work(int id) {
    // calcul indépendant
}

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();
}
```

Chaque thread peut être exécuté sur un cœur différent.

5.6 Mémoire partagée

Les threads partagent la mémoire, ce qui introduit des **risques majeurs** :

- conditions de course (*race conditions*),
- incohérences de données,
- comportements non déterministes.

Exemple dangereux :

```
int counter = 0;

void increment() {
    counter++; // non atomique
}
```

Si plusieurs threads exécutent `increment()`, le résultat est imprévisible.

5.7 Synchronisation et sections critiques

Une **section critique** est une zone de code qui ne doit être exécutée que par **un seul thread à la fois**.

En C++, on utilise `std::mutex`.

(`std::mutex` : mutex (verrou) défini dans `<mutex>` utilisé pour protéger une section critique.)

```
#include <mutex>

int counter = 0;
std::mutex m;

void increment() {
    std::lock_guard<std::mutex> lock(m);
    counter++;
}
```

- le mutex empêche l'accès concurrent,
- `lock_guard` garantit le déverrouillage automatique.

5.8 Variables atomiques

Pour des opérations simples, on peut utiliser `std::atomic`.

```cpp
#include <atomic>

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

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

Avantages :

- plus rapide qu'un mutex,
- sûr pour des opérations élémentaires.

Limite :

- inadapté aux structures complexes.

Coût et limites du multithreading

Créer des threads a un coût :

- création,
- synchronisation,
- contention sur la mémoire.

Trop de threads peut :

- dégrader les performances,
- augmenter la latence,
- compliquer le raisonnement.

Bonne pratique :

- utiliser un nombre de threads proche du nombre de cœurs,
- privilégier les tâches grossières plutôt que très fines.

6 Programmation générique, template

La **programmation générique** permet d'écrire du code **indépendant des types**, tout en conservant les **performances du C++ compilé**. En C++, ce paradigme repose principalement sur les **templates**, qui permettent de définir des fonctions et des classes paramétrées par des types (ou des valeurs). Les templates sont omniprésents dans la bibliothèque standard (STL) et constituent un outil fondamental pour écrire du code réutilisable, expressif et efficace.

6.1 Principe général des templates

Un **template** est un modèle de code qui n'est **pas directement compilé**. Le compilateur génère automatiquement une version spécialisée du code pour chaque type utilisé.

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

Le mot-clé `typename` (ou alternativement `class` dans ce contexte) introduit un **paramètre de type** dans la déclaration `template <typename T>`.

Utilisation :

```
int a = add(2, 3);          // T = int
float b = add(1.5f, 2.5f);  // T = float
```

Pour chaque type (`int`, `float`), le compilateur génère une fonction différente, avec les mêmes performances qu'un code écrit à la main.

Templates de fonctions

Les templates de fonctions permettent d'écrire des algorithmes génériques sans dupliquer le code.

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

Cette fonction fonctionne pour tout type supportant l'opérateur `>` :

```
maximum(3, 5);          // int
maximum(2.0f, 1.5f);    // float
```

Si le type ne supporte pas l'opérateur requis, l'erreur est détectée **à la compilation**.

Templates de classes

Les templates peuvent aussi être utilisés pour définir des classes génériques.

```
template <typename T>
struct Box {
    T value;

    explicit Box(T v) : value(v) {}
};
```

Utilisation :

```
Box<int> a(3);
Box<float> b(2.5f);
```

Ici, `Box<int>` et `Box<float>` sont **deux types distincts** générés par le compilateur.

Exemples pour des vecteurs

En informatique graphique, les templates sont très utilisés pour :

- vecteurs et matrices de dimensions ou types variés,
- buffers CPU/GPU typés,
- algorithmes indépendants de la précision (`float`, `double`).

Exemple de vecteur générique :

```
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;
    }
};
```

Utilisation :

```
vec3<float> vf(1.0f, 2.0f, 3.0f);
vec3<double> vd(1.0, 2.0, 3.0);
```

Paramètres templates non typés

Un template peut aussi prendre des **paramètres non typés**, connus à la compilation.

```
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]; }
};
```

Utilisation :

```
Array<float, 3> v;   // taille connue à la compilation
```

Ce principe est utilisé dans `std::array<T, N>`.

Spécialisation de templates

Il est possible de fournir une implémentation spécifique pour un type donné.

```
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;
    }
};
```

La spécialisation permet d'adapter le comportement sans modifier le code générique.

6.2 Principes de compilation: duck typing, instanciation et fichiers d'en-tête

La compilation des templates en C++ obéit à des règles spécifiques, différentes de celles du code classique. Comprendre ces principes est essentiel pour interpréter les messages d'erreur du compilateur et organiser correctement son code.

Duck typing statique

Les templates reposent sur un principe appelé **duck typing statique**.

Le principe est le suivant :

> *Un type est valide s'il fournit toutes les opérations utilisées dans le template.*

Par exemple :

```
template <typename T>
T square(T x) {
    return x * x;
}
```

Ce template n'impose **aucune contrainte explicite** sur `T`. Cependant, lors de l'instanciation, le compilateur exige que le type utilisé possède l'opérateur `*`.

```
square(3);       // OK : int supporte *
square(2.5f);    // OK : float supporte *
```

En revanche :

```
struct A {};

square(A{}); // ERREUR de compilation
```

L'erreur apparaît **au moment où le template est instancié**, et non lors de sa définition. C'est une caractéristique clé des templates :

- le code générique peut être syntaxiquement correct,
- mais invalide pour certains types concrets.

Ce mécanisme explique pourquoi les erreurs liées aux templates peuvent être longues et complexes : le compilateur tente d'instancier le code avec un type donné et échoue lorsqu'une opération requise n'existe pas.

Instanciation des templates

Un template n'est **pas compilé tant qu'il n'est pas utilisé**. La compilation réelle se fait lors de l'**instanciation**, c'est-à-dire lorsque le compilateur rencontre une utilisation concrète :

```
add<int>(2, 3);
add<float>(1.5f, 2.5f);
```

Chaque instanciation génère :

- une fonction différente pour chaque type,
- ou un type différent pour chaque combinaison de paramètres template.

Ainsi :

```
Box<int>
Box<float>
```

sont deux **types distincts**, sans relation d'héritage entre eux.

Conséquence importante : code visible à la compilation

Pour que le compilateur puisse instancier un template, il doit avoir accès à **l'implémentation complète** du template au moment de la compilation.

Cela a une conséquence majeure sur l'organisation des fichiers.

Templates et fichiers d'en-tête (`.hpp`)

Contrairement aux fonctions et classes classiques, **le corps des templates doit être visible partout où ils sont utilisés**. C'est pourquoi :

- les templates sont définis **dans les fichiers d'en-tête** (`.hpp`),
- ils ne sont généralement **pas séparés** en `.hpp` / `.cpp`.

Exemple correct :

```
// vec.hpp
#pragma once

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

```
// main.cpp
#include "vec.hpp"

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

Si le corps du template était placé dans un `.cpp`, le compilateur ne pourrait pas générer les versions spécialisées, car l'implémentation ne serait pas visible au moment de l'instanciation.

Pourquoi les templates ne peuvent pas être compilés séparément

Dans un code classique :

- le compilateur produit un fichier objet (`.o`) à partir d'un `.cpp`,
- le linker assemble ensuite les symboles.

Avec les templates :

- le code généré dépend des **types utilisés**,
- ces types ne sont connus qu'au point d'utilisation.

Le compilateur ne peut donc pas produire à l'avance une version générique unique du template. Il doit voir **à la fois** :

- la définition du template,
- et le type concret utilisé.

Exceptions et cas particuliers

Il existe des techniques avancées (instanciation explicite) permettant de séparer partiellement l'implémentation, mais elles restent complexes, en pratique, la règle simple est :

> **Tout template doit être entièrement défini dans un fichier d'en-tête.**

Résumé des principes clés

- Les templates utilisent un **duck typing statique** : les contraintes sur les types sont implicites.
- Les erreurs sont détectées **à l'instanciation**, pas à la définition.
- Chaque combinaison de paramètres template génère un code spécifique.
- Le compilateur doit voir **l'implémentation complète** du template.
- Les templates sont donc définis dans des fichiers `.hpp`, pas `.cpp`.

Ces règles expliquent à la fois la **puissance** et la **complexité** des templates en C++.

6.3 Meta-programmation statique

La **méta-programmation statique** désigne l'ensemble des techniques permettant d'effectuer des **calculs au moment de la compilation**, avant même l'exécution du programme. En C++, les templates et les expressions `constexpr` permettent de déplacer une partie de la logique du programme vers le compilateur. Le résultat est un code **plus rapide à l'exécution**, car certaines décisions et certains calculs sont déjà résolus.

Principe général

L'idée centrale est la suivante :

> utiliser le compilateur comme un **moteur de calcul**.

Les valeurs produites par la méta-programmation :

- sont connues à la compilation,
- ne coûtent **aucun temps de calcul** à l'exécution,
- peuvent être utilisées comme **paramètres templates**, tailles de tableaux, ou constantes.

Méta-programmation avec paramètres templates entiers

Les paramètres templates non typés (entiers) sont le premier outil de méta-programmation.

```
template <int N>
int static_square()
{
    return N * N;
}
```

Utilisation :

```
int main()
{
    const int a = static_square<5>();     // évalué à la compilation
    float buffer[static_square<3>()];     // taille connue statiquement

    std::cout << a << std::endl;
    std::cout << sizeof(buffer) / sizeof(float) << std::endl;
}
```

Ici :

- `static_square<5>()` est calculé par le compilateur,
- aucune multiplication n'est exécutée au run-time.

`constexpr` : calculs évalués par le compilateur

Depuis C++11, le mot-clé `constexpr` permet de demander explicitement une **évaluation à la compilation**, si les arguments sont constants.

```
constexpr int square(int N)
{
    return N * N;
}
```

Le compilateur :

- vérifie que l'expression peut être évaluée statiquement,
- génère une constante si c'est le cas.

Comparaison avec une fonction classique :

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

Utilisation dans un paramètre template :

```
template <int N>
void print_value()
{
    std::cout << N << std::endl;
}

int main()
{
    print_value<square(5)>();        // OK : expression constante
    // print_value<runtime_square(5)>(); // ERREUR : non constante
}
```

Calculs récursifs à la compilation

Les templates et `constexpr` permettent d'écrire des calculs **récursifs** évalués à la compilation.
Exemple : calcul du factoriel.

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

Utilisation comme paramètre template :

```
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);
}
```

Le calcul de `4!` est effectué **entièrement à la compilation**.

Méta-programmation par templates (forme historique)

Avant `constexpr`, la méta-programmation reposait exclusivement sur des **templates récursifs**.

```
template <int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static constexpr int value = 1;
};
```

Utilisation :

```
int size = Factorial<5>::value; // évalué à la compilation
```

Cette technique est plus complexe et moins lisible, mais elle est importante historiquement et encore présente dans certaines bibliothèques.

Cas d'usage typiques

La méta-programmation statique est utilisée pour :

- tailles de tableaux connues à la compilation,
- algorithmes spécialisés selon des paramètres constants,
- choix de code conditionnel (`if constexpr` en C++17),
- optimisation agressive sans coût à l'exécution,
- structures mathématiques génériques (vecteurs, matrices).

Exemple avec `if constexpr` :

```
template <typename T>
void process(T v)
{
    if constexpr (std::is_integral_v<T>)
        std::cout << "Entier" << std::endl;
    else
        std::cout << "Non entier" << std::endl;
}

Note: `std::is_integral_v` est fourni par l'en-tête `<type_traits>`.
```

La branche non pertinente est **supprimée à la compilation**.

Limites et précautions

- Augmente le **temps de compilation**.
- Peut rendre les erreurs plus difficiles à comprendre.
- Le code peut devenir moins lisible si la méta-programmation est excessive.

6.4 Déduction de types dans les templates

L'un des objectifs majeurs de la programmation générique est de rendre le code **à la fois générique et lisible**. En C++, le compilateur est capable de **déduire automatiquement les paramètres template** dans de nombreux cas, à partir des arguments fournis lors de l'appel. Comprendre quand cette déduction fonctionne — et quand elle échoue — est essentiel pour écrire des interfaces génériques efficaces.

Principe général de la déduction

Lorsqu'un template est utilisé **sans préciser explicitement ses paramètres**, le compilateur tente de les déduire à partir des types des arguments.

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

Utilisation :

```
int a = add(2, 3);       // T déduit comme int
float b = add(1.2f, 3.4f); // T déduit comme float
```

Ici, le compilateur déduit `T` automatiquement à partir des arguments passés à la fonction.

Limites de la déduction automatique

La déduction de types fonctionne **uniquement** à partir des **paramètres de la fonction**. Elle ne fonctionne pas à partir du type de retour.

```
template <typename T>
T identity();
```

Ce template **ne peut pas être appelé** sans préciser `T`, car le compilateur n'a aucune information pour le déduire.

```
// identity();   // ERREUR
identity<int>(); // OK
```

Exemple problématique : produit scalaire générique

Considérons une fonction générique de produit scalaire :

```
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;
}
```

Utilisation :

```
vecN<float,3> v0, v1;

// Appel lourd et peu lisible
float p = dot<vecN<float,3>, float, 3>(v0, v1);
```

Dans ce cas :

- `TYPE_INPUT`, `TYPE_OUTPUT` et `SIZE` **ne peuvent pas être déduits automatiquement**,
- l'appel devient verbeux et difficile à lire.

Pourquoi la déduction échoue ici

La déduction échoue car :

- `TYPE_OUTPUT` n'apparaît que dans le **type de retour**,
- `SIZE` n'apparaît que comme **paramètre template**, pas dans les arguments de la fonction.

Le compilateur ne peut déduire un paramètre template que s'il est **directement lié aux types des arguments**.

Exposer les paramètres template dans les types

Une solution consiste à exposer explicitement les paramètres templates dans la classe générique.

```
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];
};
```

On peut alors écrire une fonction bien plus lisible :

```cpp
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;
}
```

Utilisation :

```cpp
float p = dot(v0, v1); // types et taille déduits automatiquement
```

Ici :

- `V` est déduit comme `vecN<float,3>`,
- le type de retour est extrait via `V::value_type`,
- la taille est connue à la compilation via `V::size()`.

Accès aux types internes : `typename`

Lorsqu'un type dépend d'un paramètre template, il doit être précédé de `typename` pour indiquer au compilateur qu'il s'agit bien d'un type.

```cpp
typename V::value_type
```

Sans `typename`, le compilateur ne peut pas savoir si `value_type` est un type ou une valeur statique.

Déduction partielle et paramètres par défaut

Les templates peuvent aussi utiliser des **paramètres par défaut** pour réduire la verbosité :

```cpp
template <typename T, int N = 3>
struct vecN;
```

Ce mécanisme permet de simplifier certaines utilisations, mais ne remplace pas une bonne conception des interfaces.

Déduction avec `auto` et C++17+

Depuis C++17, `auto` peut être utilisé pour déduire le type de retour d'une fonction template :

```cpp
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;
}
```

Cela améliore la lisibilité tout en conservant la généricité.

6.5 Spécialisation des templates

La **spécialisation des templates** permet d'adapter le comportement d'un template générique à **un cas particulier**, sans modifier l'implémentation générale. Elle est utilisée lorsque, pour un type ou un paramètre précis, le comportement par défaut n'est pas adapté, inefficace ou incorrect.

La spécialisation est un mécanisme **résolu à la compilation**, et fait partie intégrante de la programmation générique en C++.

Principe général

On commence par définir un **template générique** (cas général), puis on fournit une implémentation **spécialisée** pour un type ou une valeur donnée.

```
template <typename T>
struct Printer
{
    static void print(T const& v)
    {
        std::cout << v << std::endl;
    }
};
```

Ce template fonctionne pour tout type compatible avec `operator<<`.

Spécialisation complète d'un template

Une **spécialisation complète** remplace entièrement l'implémentation du template pour un type précis.

```
template <>
struct Printer<bool>
{
    static void print(bool v)
    {
        std::cout << (v ? "true" : "false") << std::endl;
    }
};
```

Utilisation :

```
Printer<int>::print(5);     // utilise la version générique
Printer<bool>::print(true); // utilise la spécialisation
```

Le compilateur choisit automatiquement la version la plus spécifique disponible.

Spécialisation de templates de fonctions

Les templates de fonctions peuvent également être spécialisés, mais leur usage est plus délicat.

```
template <typename T>
void display(T v)
{
    std::cout << v << std::endl;
}

template <>
void display<bool>(bool v)
{
    std::cout << (v ? "true" : "false") << std::endl;
}
```

Ici aussi, la version spécialisée est utilisée lorsque `T = bool`.

Spécialisation partielle (templates de classes)

La **spécialisation partielle** permet de spécialiser un template pour **une famille de types**, mais elle n'est autorisée **que pour les templates de classes**, pas pour les fonctions.

Exemple : spécialisation selon un paramètre entier.

```
template <typename T, int N>
struct Array
{
    T data[N];
};
```

Spécialisation partielle pour `N = 0` :

```cpp
template <typename T>
struct Array<T, 0>
{
    // tableau vide
};
```

Ici, tous les types `Array<T,0>` utilisent cette version spécifique.

Spécialisation partielle avec types pointeurs

Autre exemple classique :

```cpp
template <typename T>
struct is_pointer
{
    static constexpr bool value = false;
};

template <typename T>
struct is_pointer<T*>
{
    static constexpr bool value = true;
};
```

Utilisation :

```cpp
is_pointer<int>::value;    // false
is_pointer<int*>::value;  // true
```

Ce type de spécialisation est largement utilisé dans la STL (`std::is_pointer`, `std::is_integral`, etc.).

Spécialisation totale (ou complète)

La **spécialisation totale** consiste à fournir une implémentation spécifique pour **une combinaison entièrement fixée des paramètres template** (types et/ou valeurs). Pour cette combinaison précise, le template générique **n'est pas utilisé du tout** : la spécialisation le remplace intégralement.

Dans le contexte des **vecteurs génériques**, cela permet par exemple :

- d'optimiser un cas particulier (dimension courante),
- de définir un comportement différent pour une taille donnée,
- ou d'adapter une représentation interne.

Exemple : vecteur générique de taille fixe

On définit d'abord un template générique pour un vecteur de taille arbitraire connue à la compilation.

```cpp
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]; }
};
```

Ce template fonctionne pour **tout type** `T` et **toute taille** `N`.

Spécialisation totale pour un vecteur 2D

Supposons que l'on souhaite un traitement particulier pour les vecteurs 2D, par exemple :

- accès direct via `x` et `y`,
- code plus lisible,
- éventuellement plus optimisable.

On définit alors une **spécialisation totale** :

```
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;
    }
};
```

Ici :

- `vec<T,2>` est **un type complètement différent** de `vec<T,N>`,
- le tableau `data[N]` n'existe plus,
- le comportement est entièrement redéfini pour `N = 2`.

Utilisation

```
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>` utilise le **template générique**,
- `vec<float,2>` utilise la **spécialisation totale**.

Le choix est fait **à la compilation**, sans aucun test à l'exécution.

Spécialisation totale pour un type et une taille précis

Il est aussi possible de spécialiser pour **un type et une taille précis**.

```
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;
    }
};
```

Utilisation :

```
vec<float,3> v(1.f, 2.f, 3.f);
std::cout << v.norm2() << std::endl;
```

Ici :

- cette version est utilisée **uniquement** pour `vec<float,3>`,
- toutes les autres combinaisons (`vec<double,3>`, `vec<float,4>`, etc.) utilisent le template générique.

Comparaison avec la spécialisation partielle

- **Spécialisation totale** Tous les paramètres template sont fixés (`vec<float,3>`). → un cas unique, comportement entièrement redéfini.
- **Spécialisation partielle** Seule une partie des paramètres est fixée (`vec<T,2>`). → une famille de types partageant un comportement spécifique.

6.6 Priorité entre spécialisation et surcharge

Il est fréquent de confondre **surcharge (overloading)** et **spécialisation de templates**, mais ce sont deux mécanismes distincts qui interviennent **à des moments différents** de la compilation. Comprendre leur **ordre de priorité** est essentiel pour éviter des comportements surprenants.

L'idée clé est la suivante :

> **La surcharge est résolue avant la spécialisation de templates.**

Autrement dit, le compilateur choisit **d'abord quelle fonction appeler**, puis seulement **quelle version de template instancier**.

Étape 1 : résolution de surcharge (overloading)

Lorsque plusieurs fonctions portent le même nom, le compilateur commence par appliquer les règles classiques de surcharge :

- correspondance exacte des types,
- conversions implicites,
- templates vs fonctions non templates.

Exemple :

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

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

Appel :

```
display(3);
```

Résultat :

```
fonction normale int
```

Une fonction non template est toujours prioritaire sur une fonction template si elle correspond exactement.

Étape 2 : sélection du template

Si aucune fonction non-template ne correspond, le compilateur considère les **fonctions templates** et tente d'en déduire les paramètres.

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

display(3.5); // T = double
```

Ici, le template est sélectionné car aucune fonction classique ne correspond.

Étape 3 : spécialisation du template

Une fois qu'un **template a été choisi**, le compilateur cherche s'il existe une **spécialisation plus spécifique** pour les paramètres déduits.

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

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

Appels :

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

Résultat :

```
template generique
specialisation bool
```

La spécialisation **ne participe pas à la surcharge**. Elle est sélectionnée **après** que le template générique a été choisi.

Cas subtil : spécialisation vs surcharge

Considérons maintenant :

```cpp
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";
}
```

Appel :

```cpp
display(3);
```

Résultat :

```
fonction normale int
```

Explication :

1. le compilateur voit une fonction non-template `display(int)` → **prioritaire**,
2. le template n'est même pas considéré,
3. la spécialisation du template est ignorée.

Une spécialisation ne peut jamais battre une surcharge non-template.

Pourquoi ce comportement ?

Parce que :

- la surcharge est une décision **syntaxique et locale**,
- la spécialisation est une décision **interne au template**,
- mélanger les deux niveaux rendrait la compilation ambiguë.

C++ impose donc une hiérarchie stricte.

Résumé de la priorité (ordre exact)

Lors d'un appel de fonction :

1. Sélection des fonctions candidates (nom, portée).
2. Résolution de surcharge :
   - fonctions non templates,
   - puis fonctions templates.
3. Si un template est choisi :
   - sélection de la spécialisation la plus spécifique.
4. Instanciation du code correspondant.

Règle pratique à retenir

La surcharge choisit la fonction. La spécialisation choisit l'implémentation du template.

Bonnes pratiques

- Utiliser la **surcharge** pour proposer des interfaces différentes.
- Utiliser la **spécialisation** pour adapter un comportement interne à un template.
- Éviter de mélanger surcharge et spécialisation sur un même nom sans raison claire.

6.7 Alias

Alias de types dans les templates (`typedef` et `using`)

Les **alias de types** permettent de donner un **nom plus lisible** ou plus **expressif** à un type, souvent complexe. Ils jouent un rôle central en programmation générique, car ils facilitent la **déduction de types**, l'**écriture de fonctions génériques** et la **lisibilité des interfaces**.

En C++, il existe deux mécanismes équivalents :

- `typedef` (historique),
- `using` (moderne, recommandé).

Alias avec `typedef` (forme historique)

```
typedef unsigned int uint;
```

Ce mécanisme fonctionne, mais devient rapidement peu lisible avec des types complexes, notamment en présence de templates.

Alias avec `using` (forme moderne)

Depuis C++11, on préfère utiliser `using`, plus clair et plus puissant.

```
using uint = unsigned int;
```

Cette syntaxe est équivalente à `typedef`, mais beaucoup plus lisible, surtout avec des templates.

Alias dans une classe template

Les alias sont très souvent utilisés **à l'intérieur des classes templates** pour exposer leurs paramètres internes.

Exemple avec un vecteur générique :

```
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];
};
```

Ici :

- `vec<T,N>::value_type` donne accès au type stocké,
- `vec<T,N>::size()` donne accès à la taille connue à la compilation.

Ces alias rendent la classe **auto-descriptive** et facilitent son utilisation dans du code générique.

Utilisation des alias dans des fonctions templates

Grâce aux alias, on peut écrire des fonctions génériques **sans connaître explicitement les paramètres template**.

```
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;
}
```

Utilisation :

```
vec<float,3> v;
v[0] = 1.0f; v[1] = 2.0f; v[2] = 3.0f;

float s = sum(v);
```

Ici :

- le type de retour est automatiquement déduit via `value_type`,
- la fonction fonctionne pour **tout type de vecteur compatible**.

Alias et types dépendants (`typename`)

Lorsque l'on accède à un alias dépendant d'un paramètre template, il est nécessaire d'utiliser le mot-clé `typename` pour indiquer qu'il s'agit bien d'un **type**.

```
typename V::value_type
```

Sans `typename`, le compilateur ne peut pas savoir si `value_type` est un type ou une valeur statique.

Alias templates (alias paramétrés)

Les alias eux-mêmes peuvent être **templates**, ce qui permet de simplifier des types très complexes.

```
template <typename T>
using vec3 = vec<T, 3>;
```

Utilisation :

```
vec3<float> a;
vec3<double> b;
```

Ici :

- `vec3<float>` est équivalent à `vec<float,3>`,
- l'alias améliore fortement la lisibilité.

Alias et cohérence des interfaces génériques

Les alias sont largement utilisés dans la STL :

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

Respecter ces conventions permet de rendre ses classes **compatibles avec les algorithmes génériques**.
Exemple :

```
template <typename Container>
void print_container(Container const& c)
{
    for (typename Container::value_type const& v : c)
        std::cout << v << " ";
}
```

7 Vue matérielle

Ce chapitre propose une **vue simplifiée mais cohérente du matériel** sous-jacent à l'exécution d'un programme C/C++. L'objectif n'est pas d'entrer dans l'électronique fine, mais de comprendre **comment le code est physiquement exécuté**, et pourquoi certaines notions (mémoire, cache, alignement, performances) sont cruciales en informatique graphique et scientifique.

7.1 Principe du transistor

Le **transistor** est l'élément fondamental de tout circuit électronique moderne. Un processeur contient aujourd'hui **des milliards de transistors**, chacun se comportant comme un **interrupteur contrôlable électroniquement**.

Rôle fondamental

Un transistor peut être vu comme :

- un **interrupteur ouvert ou fermé**,
- commandé par un signal électrique.

On associe classiquement :

- état bloqué → `0`
- état passant → `1`

Ces deux états permettent de représenter l'**information binaire**.

Principe physique du transistor

Principe physique du transistor

Le **transistor** est avant tout un **objet physique**, dont le fonctionnement repose sur les propriétés électriques de la matière à l'échelle microscopique. Comprendre son principe physique permet de saisir comment un phénomène continu (tensions, champs électriques, électrons) est exploité pour produire une **logique discrète** (`0` / `1`).

Le silicium et la conduction électrique

Le matériau central de l'électronique moderne est le **silicium**, un cristal dont les électrons sont liés aux atomes par des liaisons covalentes. À l'état pur :

- le silicium conduit très mal le courant,
- il ne se comporte ni comme un isolant parfait, ni comme un bon conducteur.

Sa conductivité peut cependant être **contrôlée** grâce au **dopage**.

Dopage et porteurs de charge

Le dopage consiste à introduire une très faible quantité d'atomes étrangers dans le cristal de silicium.

- **Dopage de type N** Atomes avec un électron en excès → apparition d'électrons libres (charges négatives)
- **Dopage de type P** Atomes avec un électron manquant → apparition de trous (charges positives effectives)

Ces porteurs de charge sont mobiles sous l'effet d'un champ électrique, ce qui permet le passage du courant.

Jonction PN et contrôle du courant

Lorsqu'une région dopée P est mise en contact avec une région dopée N, il se forme une **jonction PN**.
À l'interface :

- les électrons et les trous se recombinent,
- une zone appauvrie en charges apparaît,
- cette zone crée une **barrière de potentiel**.

Selon la tension appliquée :

- la barrière est abaissée → courant autorisé,
- la barrière est renforcée → courant bloqué.

C'est la première brique physique du **contrôle électrique**.

Le transistor MOSFET : champ électrique plutôt que courant

Les processeurs modernes utilisent presque exclusivement des **transistors MOSFET** – Metal Oxide Semiconductor Field Effect Transistor.

- **Metal-Oxide-Semiconductor (MOS)** : Décrit la structure physique (une "grille" métallique isolée du semiconducteur par une couche d'oxyde).
- **Field-Effect (Effet de Champ)** : Décrit le principe de commande. C'est un champ électrique (créé par une tension) qui contrôle le passage du courant, et non un courant (contrairement au transistor bipolaire).

Contrairement aux anciens transistors, ils sont contrôlés par un champ électrique (une tension), et non par un courant, ce qui réduit considérablement leur consommation.
Un MOSFET est constitué de quatre terminaux principaux :

1. **Source (S) :** L'entrée des électrons.
2. **Drain (D) :** La sortie des électrons.
3. **Grille (G) :** L'électrode de commande.
4. **Substrat (Body) :** Le corps du transistor.

L'innovation clé : La Grille est isolée électriquement du canal par une couche d'**Oxyde** extrêmement fine.

3. L'Analogie du Robinet Pour comprendre le fonctionnement, visualisez un robinet d'eau :

Composant MOSFET	Analogie Robinet	Rôle
Source	Arrivée d'eau	Fournit le courant.
Drain	Sortie d'eau	Reçoit le courant.
Grille	Poignée	Contrôle le débit sans toucher l'eau.
Tension	Force sur la poignée	La commande d'ouverture.

4. La Physique de la Commutation : La Tension de Seuil () Le passage du courant n'est pas instantané. Il repose sur un phénomène appelé **l'inversion**.

- **Au repos () :** Le transistor est une barrière. Aucun courant ne passe entre Source et Drain.
- **Sous tension () :** La tension positive sur la grille agit comme un aimant. Elle attire les électrons minoritaires du substrat vers la surface, juste sous l'oxyde.
- **Le Seuil () :** Lorsque la tension dépasse une valeur critique appelée **Tension de Seuil** (), la concentration d'électrons est suffisante pour former un "pont" conducteur entre la Source et le Drain : le **canal** est créé.

5. Les Régimes de Fonctionnement (Mathématique simplifiée) Le comportement du courant de drain () suit trois régimes selon les tensions appliquées :

1. **Régime de Blocage (Cut-off) :**

-
- Le robinet est fermé. .
- *Interprétation logique :* **État 0**.

2. **Régime Linéaire (Ohmique) :**

- Le canal est ouvert, mais la différence de potentiel Drain-Source est faible. Le transistor agit comme une simple résistance.
- est proportionnel à la tension appliquée.

3. **Régime de Saturation :**

- Le canal est pleinement passant et le courant est maximal et constant pour une tension de grille donnée.
- C'est le régime utilisé pour l'amplification, ou l'état "pleinement passant" en logique numérique.
- *Interprétation logique :* **État 1**.

6. Du phénomène physique au bit logique En informatique, on abstraite ces comportements complexes pour ne garder que deux états stables :

- **Tension basse ($ < V_{th}$) :** Transistor Bloqué Bit `0`
- **Tension haute ($ > V_{th}$) :** Transistor Saturé Bit `1`

Cependant, à l'échelle nanométrique actuelle (transistors de quelques nanomètres), des contraintes physiques réapparaissent :

- **Courants de fuite :** Même bloqué, le transistor laisse passer un infime courant (effet tunnel), ce qui chauffe le processeur.
- **Dissipation thermique :** C'est la limite principale à l'augmentation de la fréquence des horloges (GHz).

Échelle nanométrique et contraintes physiques

Les transistors actuels mesurent quelques **nanomètres**. À cette échelle :

- les effets quantiques deviennent significatifs,
- des courants de fuite apparaissent,
- la dissipation thermique devient critique.

Ces contraintes expliquent :

- les limites de fréquence des processeurs,
- la nécessité du parallélisme,
- l'importance de l'efficacité énergétique.

Du transistor à la logique

En combinant plusieurs transistors, on construit :

- des **portes logiques** (AND, OR, NOT, XOR),
- puis des circuits plus complexes :
  - additionneurs,
  - multiplexeurs,
  - registres,
  - unités de calcul.

Exemple conceptuel :

- une addition entière est réalisée par une **cascade de portes logiques**,
- chaque porte est elle-même constituée de transistors.

Ainsi, toute instruction C++ (addition, comparaison, saut conditionnel) se traduit ultimement par des **commutations de transistors**.

7.2 Structure de base de la mémoire et des opérations arithmétiques

Principes de la mémoire

La sauvegarde d'un **élément en mémoire** repose sur une **organisation très précise de transistors**, différente selon le type de mémoire. Voici une explication progressive, en partant du **bit** jusqu'aux mémoires utilisées dans un CPU.

7.3 Organisation minimale : stocker un bit

Un bit doit :

- représenter `0` ou `1`,
- rester stable dans le temps,
- pouvoir être lu et modifié.

On notera plusieurs manière de stocker l'information à l'aide de transistors:

1. La mémoire statique (SRAM)

La **SRAM** (Static Random Access Memory) est utilisée pour :

- les **registres** du processeur,
- les **caches** L1, L2 et L3.

Temps d'accès typique: 0,3 à 2 ns
Elle est :

- extrêmement rapide,
- **non rafraîchie** (tant qu'elle est alimentée),
- très stable,
- mais **coûteuse en surface**, car chaque bit utilise plusieurs transistors.

Principe général

Un bit de SRAM est stocké à l'aide d'une **bascule électronique bistable**, réalisée avec des transistors.
Organisation classique :

- **6 transistors par bit** (cellule 6T) :
  - 4 transistors forment deux inverseurs croisés,
  - 2 transistors servent à l'accès en lecture/écriture.

Fonctionnement :

- les inverseurs se maintiennent mutuellement dans un état stable,
- l'état correspond à `0` ou `1`,
- tant que l'alimentation est présente, l'état est conservé.

Lecture :

- non destructive,
- très rapide.

Écriture :

- force temporairement un état sur la bascule.

La SRAM stocke donc l'information sous forme d'un **équilibre électrique actif** entre transistors.

2. La mémoire dynamique (DRAM)

La **DRAM** (Dynamic Random Access Memory) constitue la **mémoire centrale** d'un ordinateur (RAM).
Temps d'accès typique: 50 à 100 ns
Elle est :

- plus lente que la SRAM,
- **volatile**,
- nécessite un **rafraîchissement périodique**,
- beaucoup plus dense (moins de transistors par bit).

Principe général

Un bit de DRAM est stocké sous forme de **charge électrique**.
Organisation classique :

- **1 transistor + 1 condensateur par bit** (cellule 1T1C).

Fonctionnement :

- le condensateur stocke une charge (`1`) ou est vide (`0`),
- le transistor contrôle l'accès à la cellule.

Lecture :

- la charge est mesurée,
- la lecture est **destructive** (le condensateur se décharge),
- la valeur doit être réécrite immédiatement.

Rafraîchissement :

- les charges fuient naturellement,
- chaque cellule doit être relue et réécrite périodiquement (environ toutes les 64 ms).

La DRAM stocke l'information sous forme de **charge passive**, d'où la nécessité du rafraîchissement.

3. La mémoire flash

La **mémoire flash** est une mémoire **non volatile** utilisée pour :

- les SSD,
- les clés USB,
- les cartes mémoire,
- le stockage firmware (BIOS, microcontrôleurs).

Elle est :

- persistante sans alimentation,
- plus lente à l'écriture que la RAM,
- limitée en nombre de cycles d'écriture,
- très dense.

Temps d'accès typique: 50 à 100 µs (microsecondes) en lecture, 200 µs à quelques ms en écriture

Principe général

Un bit de mémoire flash est stocké grâce à un **transistor à grille flottante**.
Organisation :

- la cellule est un transistor MOS modifié,
- elle possède une **grille flottante isolée électriquement**.

Fonctionnement :

- l'écriture consiste à **injecter des électrons** dans la grille flottante à l'aide d'une haute tension,
- les électrons restent **piégés** dans l'isolant,
- la présence ou l'absence de charge modifie le comportement du transistor.

Lecture :

- non destructive,
- basée sur la mesure du seuil de conduction.

Effacement :

- se fait par blocs entiers,
- nécessite également des tensions élevées.

La mémoire flash stocke l'information sous forme de **charges piégées physiquement**, ce qui explique sa persistance sans alimentation.

Comparaison synthétique

Type de mémoire	Volatile	Transistors / bit	Rafraîchissement	Usage principal
SRAM	oui	~6	non	registres, caches
DRAM	oui	1 + 1 condensateur	oui	mémoire centrale
Flash	non	1 (spécifique)	non	stockage persistant

Principes des opérations arithmétiques

Du code C++ à l'instruction machine

Une opération arithmétique écrite en C++ est une **expression abstraite** :

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

Pour le processeur, cela correspond à une séquence bien définie :

1. charger `a` et `b` depuis la mémoire vers des **registres**,
2. activer l'unité arithmétique avec l'opération demandée,
3. produire un résultat binaire,
4. stocker le résultat dans un registre ou en mémoire.

Le processeur ne "comprend" jamais les variables ou les types C++ : il ne manipule que des **registres**, des **opcodes**, et des **bits**.

Rôle central de l'ALU

Les opérations arithmétiques et logiques sont réalisées par l'**ALU** (*Arithmetic Logic Unit*).
Fonctions principales de l'ALU :

- addition et soustraction,
- opérations logiques (AND, OR, XOR),
- comparaisons,
- décalages binaires.

Toutes ces opérations reposent sur :

- des **circuits combinatoires**,
- composés de portes logiques,
- elles-mêmes construites à partir de transistors.

L'ALU reçoit :

- deux opérandes depuis les registres,
- un code indiquant l'opération à effectuer,
- et produit un résultat ainsi que des **drapeaux d'état**.

Soustraction, comparaisons et logique interne

Dans l'ALU :

- la **soustraction** est implémentée comme une addition modifiée,
- les **comparaisons** exploitent le résultat d'une soustraction interne,
- les opérateurs relationnels (`<`, `>`, `==`) ne produisent qu'un bit logique.

Exemple conceptuel :

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

Matériellement :

- le processeur calcule `a - b`,
- observe le signe ou le flag de retenue,
- et en déduit le résultat du test.

Multiplication et division : opérations composées

Contrairement à l'addition, la multiplication et la division :

- nécessitent plusieurs étapes internes,
- mobilisent des circuits plus complexes,
- sont donc plus coûteuses en cycles.

La multiplication repose sur :

- des décalages,
- des additions partielles,
- ou des unités spécialisées fortement optimisées.

La division :

- est généralement itérative,
- et constitue l'une des opérations arithmétiques les plus lentes.

Opérations sur les nombres flottants

Les calculs sur les flottants sont pris en charge par une unité distincte : la **FPU**.
Elle réalise :

- l'alignement des exposants,
- l'opération sur les mantisses,
- la normalisation du résultat,
- l'arrondi selon la norme IEEE.

Ces opérations sont plus coûteuses que celles sur les entiers, mais entièrement gérées par le matériel.

Instructions vectorielles (SIMD)

Les processeurs modernes disposent d'unités **vectorielles** capables d'appliquer une même opération sur plusieurs données simultanément.
Principe :

- une instruction,
- plusieurs opérandes traités en parallèle.

C'est une extension directe des opérations arithmétiques de base, utilisée pour :

- l'informatique graphique,
- le traitement de signaux,
- le calcul scientifique.

Ordonnancement et pipeline

Les opérations arithmétiques ne sont pas exécutées isolément :

- elles sont **pipelinées**,
- réordonnées,
- exécutées en parallèle lorsque possible.

Ainsi :

- plusieurs additions peuvent être en cours simultanément,
- tant que les dépendances de données sont respectées.

Coût réel d'une opération

Dans un programme réel :

- le **temps d'accès à la mémoire** est souvent dominant,
- le calcul arithmétique pur est rarement le goulot d'étranglement.

Optimiser les performances revient souvent à :

- réduire les accès mémoire,
- améliorer la localité des données,
- exploiter le parallélisme.

Idée clé à retenir

> Les opérations arithmétiques sont des **briques matérielles élémentaires**, orchestrées par le processeur via l'ALU, la FPU et les unités vectorielles. Le code C++ exprime des calculs logiques, mais leur exécution repose sur l'ordonnancement, le parallélisme et l'accès efficace aux données.

7.4 Notion de cache mémoire

Problème fondamental : la latence mémoire

Accéder à la mémoire principale (RAM) est :

- **beaucoup plus lent** que d'accéder aux registres ou aux unités de calcul.

Ordre de grandeur :

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

Sans mécanisme intermédiaire, le CPU passerait son temps à **attendre la mémoire**.

Principe du cache

Le **cache mémoire** est une mémoire intermédiaire :

- plus petite que la RAM,
- beaucoup plus rapide,
- intégrée au processeur.

Il stocke des **copies de blocs de mémoire récemment utilisés**.

Hiérarchie de cache

On distingue généralement :

- **L1** : très petit, très rapide, par cœur,
- **L2** : plus grand, un peu plus lent,
- **L3** : partagé entre cœurs.

Chaque niveau agit comme un tampon vers le niveau inférieur.

Localité spatiale et temporelle

Le cache repose sur deux principes fondamentaux :

- **Localité temporelle** Une donnée utilisée récemment a de fortes chances d'être réutilisée.
- **Localité spatiale** Si on accède à une adresse mémoire, les adresses voisines ont de fortes chances d'être utilisées.

C'est pourquoi :

- les tableaux contigus,
- les `std::vector`,
- les parcours séquentiels,

sont **beaucoup plus performants** que des accès aléatoires.

Lien avec la programmation C++

Exemples de code favorables au cache :

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

Exemples défavorables :

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

En informatique graphique, ce point est **crucial** :

- traitement de sommets,
- simulation de particules,
- parcours de buffers GPU/CPU.

8 Méthodologies de développement et bonnes pratiques

Ce chapitre présente les **principes méthodologiques fondamentaux** permettant de produire du code C++ :

- lisible,
- robuste,
- testable,
- maintenable,

tout en respectant les contraintes de performance et de bas niveau propres au langage.

Ces principes s'appliquent aussi bien à de petits programmes qu'à des projets complexes (simulation, moteur graphique, calcul parallèle).

8.1 Qualité de code : objectifs concrets

La qualité de code ne se mesure pas à l'élégance perçue, mais à des critères pratiques :

- **Lisibilité** : le code est compréhensible sans effort excessif.
- **Localité** : comprendre une fonction ne nécessite pas d'explorer tout le projet.
- **Robustesse** : les erreurs sont détectées et traitées explicitement.
- **Testabilité** : le code peut être validé automatiquement.
- **Évolutivité** : les modifications futures sont possibles sans réécriture massive.
- **Performance maîtrisée** : optimisation guidée par des mesures, pas par intuition.

Notons que lorsque l'on travaille à plusieurs, la **lisibilité du code** doit être la priorité. Un code lisible :

- facilite la relecture et les revues de code,
- réduit les erreurs lors des modifications,
- accélère l'intégration de nouveaux contributeurs,
- permet de raisonner et de tester plus facilement.

Dans la plupart des cas, il faut privilégier la lisibilité et la simplicité plutôt que des optimisations micro-performantes prématurées. L'efficacité peut être recherchée ensuite, de manière ciblée et mesurée, quand un goulot de performance est avéré.

Bonnes pratiques pour la lisibilité : noms explicites, fonctions courtes, commentaires quand le code n'est pas auto-documenté, formatage cohérent, et revues de code systématiques.

8.2 Principes généraux : KISS, DRY, YAGNI

KISS – *Keep It Simple, Stupid*

Un code simple est plus fiable qu'un code complexe.

- Préférer une implémentation directe à une abstraction prématurée.
- Éviter les constructions "astucieuses" difficiles à expliquer.
- Une fonction devrait idéalement tenir sur un écran.

Exemple (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<int>& g, size_t w, size_t x, size_t y) { return g[y * w + x]; }

int count_neighbors(const std::vector<int>& grid, size_t w, size_t h,
                    size_t x, size_t y)
{
    int c = 0;
    size_t y0 = (y > 0) ? y - 1 : 0;
    size_t y1 = (y + 1 < h) ? y + 1 : h - 1;
    size_t x0 = (x > 0) ? x - 1 : 0;
    size_t x1 = (x + 1 < w) ? x + 1 : w - 1;

    for (size_t yy = y0; yy <= y1; ++yy) {
        for (size_t xx = x0; xx <= x1; ++xx) {
            if (xx == x && yy == y) continue; // ignorer la cellule centrale
            c += at(grid, w, xx, yy);
        }
    }
    return c;
}
```

DRY – *Don’t Repeat Yourself*

Une logique ne doit exister qu’à un seul endroit.

Attention :

éliminer toute duplication peut mener à des abstractions inutiles.

Une duplication locale et simple est parfois préférable à une généralisation complexe.

Exemple (DRY) :

```
// Duplication (moins bon) : deux fonctions très similaires
double average_int(const std::vector<int>& v) {
    if (v.empty()) return 0.0;
    long sum = 0;
    for (int x : v) sum += x;
    return double(sum) / v.size();
}

double average_double(const std::vector<double>& v) {
    if (v.empty()) return 0.0;
    double sum = 0;
    for (double x : v) sum += x;
    return sum / v.size();
}

// Refactorisation (DRY) : une implémentation générique évite la duplication
template<typename T>
double average(const std::vector<T>& v) {
    if (v.empty()) return 0.0;
    long double sum = 0;
    for (T x : v) sum += x;
    return double(sum / v.size());
}

// Usage :
// std::vector<int> vi = {1,2,3};
// std::vector<double> vd = {1.0,2.0,3.0};
// double a1 = average(vi); // fonctionne pour int
// double a2 = average(vd); // fonctionne pour double
```

YAGNI – *You Aren't Gonna Need It*

Ne pas implémenter des fonctionnalités "au cas où" si elles ne sont pas nécessaires.

Ce principe est particulièrement important en C++, où : - les templates, - la généricité, - et la méta-programmation peuvent encourager une complexité excessive trop tôt.

Exemple (YAGNI) :

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

// Version simple et suffisante pour l'usage courant
struct vec3 { float x, y, z; };
```

8.3 Invariants, assertions et contrat de fonction

Un programme robuste ne se contente pas de "fonctionner dans les cas normaux" : il **exprime explicitement ses hypothèses** et vérifie qu'elles sont respectées.

Ces hypothèses constituent ce que l'on appelle le **contrat** du code.

Pourquoi parler de contrat ?

Lorsqu'une fonction est appelée, deux points de vue existent :

- **le point de vue de l'appelant** "Qu'ai-je le droit de passer à cette fonction ?"
- **le point de vue de la fonction** "Qu'est-ce que je garantis en retour ?"

Si ces règles sont implicites ou seulement "dans la tête du développeur", le code devient fragile :

- erreurs silencieuses,
- comportements indéterminés,
- bugs difficiles à diagnostiquer.

Le contrat permet de formaliser ces règles. L'ensemble de ces règles constitue ce que l'on appelle la programmation par contrat.

Les trois notions clés du contrat

On distingue trois types de règles complémentaires.

1. Préconditions

Une **précondition** est une condition qui **doit être vraie avant l'appel** d'une fonction.

- Elle décrit ce que la fonction **attend**.
- Elle est de la responsabilité de l'appelant.

Exemples :

- un index doit être valide,
- un pointeur ne doit pas être nul,
- un diviseur doit être non nul.

2. Postconditions

Une **postcondition** est une condition qui **doit être vraie après l'exécution** de la fonction.

- Elle décrit ce que la fonction **garantit**.
- Elle est de la responsabilité de la fonction.

Exemples :

- la taille d'un conteneur a augmenté,
- une valeur retournée respecte un intervalle,
- un état interne a été mis à jour correctement.

3. Invariants

Un **invariant** est une propriété qui doit être **toujours vraie** pour un objet valide.

- Il est établi par le constructeur.
- Il doit être préservé par **toutes les méthodes publiques**.

Exemples :

- $0 \leq \text{size} \leq \text{capacity}$,
- un rayon est toujours strictement positif,
- deux pointeurs membres sont soit tous valides, soit tous nuls.

Illustration conceptuelle : pile (stack)

Avant de voir du C++, voici une vue conceptuelle du contrat d'une pile.

```
Entité : Pile (Stack)

Invariant :
    0 <= size <= capacity

Constructeur(capacity):
    établit l'invariant
    size := 0
    capacity := capacity

push(value):
    précondition : size < capacity
    postcondition : top == value, size augmenté de 1

pop():
    précondition : size > 0
    postcondition : size diminué de 1
```

L'invariant doit être vrai **après chaque appel public**, quelle que soit la séquence d'opérations.

Assertions à l'exécution (`assert`)

Les **assertions** permettent de vérifier ces règles **pendant l'exécution**, principalement en phase de développement.
En C++, on utilise `assert` pour détecter des **erreurs de programmation**.

```cpp
#include <cassert>

float safe_div(float a, float b)
{
    assert(b != 0.0f && "Division par zero");
    return a / b;
}
```

Ici :

- `b != 0.0f` est une **précondition**,
- l'assertion documente et vérifie cette hypothèse.

À quoi servent les `assert` ?

Les assertions permettent de :

- documenter les hypothèses internes du code,
- détecter rapidement des erreurs de logique,
- arrêter le programme **au point exact du problème** en debug.

Elles sont donc un **outil de développement**, pas un mécanisme de gestion d'erreurs utilisateur.

Bonnes pratiques avec `assert`

- utiliser `assert` pour des **erreurs de programmation**. Les `assert` sont théoriquement "inutile" au bon fonctionnement du programme, ils ne servent qu'à faciliter la programmation en détectant des cas inattendues/non prévus qui ne devraient jamais arriver.
- ne pas utiliser `assert` pour :
  - fichiers absents,
  - entrées utilisateur invalides,
  - erreurs récupérables
- ne jamais écrire d'effets de bord :

```
assert(++i < 10); // interdit
// Ici la valeur de i est modifié après l'exécution de assert.
// Lors d'une compilation en mode "release", l'assertion n'est pas exécuté, et la valeur de i sera
     différente dans le programme.
```

- fournir un message explicite :

```
assert(ptr && "ptr ne doit pas être nul");
```

Mode debug vs release

- En **debug** : les `assert` sont actives
- En **release** : elles sont supprimées (`NDEBUG`)

Note: Le programme ne doit **jamais dépendre** des assertions pour fonctionner correctement.

Assertions à la compilation (`static_assert`)

Certaines règles peuvent être vérifiées **avant même l'exécution**, à la compilation.
C'est le rôle de `static_assert`.

```
#include <type_traits>

template <typename T>
T square(T x)
{
    static_assert(std::is_arithmetic_v<T>,
                  "square attend un type arithmetique");
    return x * x;
}
```

Ici :

- la contrainte est vérifiée **à la compilation**,
- une mauvaise utilisation empêche la génération de l'exécutable.

Quand utiliser `static_assert` ?

- tailles connues à la compilation,
- contraintes sur des types templates,
- hypothèses structurelles impossibles à vérifier à l'exécution.

Règle générale : **préférer les vérifications à la compilation quand c'est possible**.

Exemple complet : pile avec invariant et 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 doit être positive");
    }

    void push(int v)
    {
        // précondition
        assert(data.size() < capacity && "push: pile pleine");

        data.push_back(v);

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

    int pop()
    {
        // précondition
        assert(!data.empty() && "pop: pile vide");

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

        // invariant toujours valide
        assert(data.size() <= capacity && "invariant violé");

        return v;
    }
};
```

Résumé

- Un **contrat** décrit ce que le code attend et garantit.
- Les **préconditions** sont la responsabilité de l'appelant.
- Les **postconditions** sont la responsabilité de la fonction.
- Les **invariants** définissent les états valides d'un objet.
- `assert` vérifie le contrat à l'exécution (debug).
- `static_assert` vérifie le contrat à la compilation.
- Utilisés correctement, ils rendent le code :
  - plus sûr,

  - plus lisible,
  - et plus facile à maintenir.

Alternatives à asserts

La fonction `assert` reste assez limité en terme de fonctionalité. Des outils alternatifs peuvent aider à exprimer et vérifier des contrats de façon plus lisible, sûre et maintenable pour des codes de grande envergure :

- **GSL (Guideline Support Library)** : fournit `Expects()` / `Ensures()` (macros ou fonctions) pour documenter pré/postconditions, ainsi que `not_null<T>` et `span<T>` pour des pointeurs et vues sûres.
- **Types résultat (expected/Outcome)** : utiliser `tl::expected` / `Outcome` ou `std::expected` quand disponible pour représenter explicitement les erreurs récupérables au lieu d'exceptions ou codes magiques.
- **Concepts & `static_assert` / `constexpr`** : remonter les vérifications au moment de la compilation quand c'est possible (templates, contraintes de types), réduisant le besoin d'assertions runtime.
- **Bibliothèques de contrat** : `Boost.Contract` et autres frameworks offrent des annotations require/ensure/invariant plus riches (contrats activables/désactivables, diagnostics centralisés).
- **Annotations légères (Expects/Ensures)** : définir des wrappers `Expects(condition)` permet d'uniformiser les messages et d'activer des comportements différents selon la configuration (throw, abort, log).
- **Outils complémentaires** : sanitizeurs (ASan/UBSan/TSan) et analyse statique (clang-tidy, cppcheck) détectent des classes d'erreurs que les assertions seules ne couvrent pas.

8.4 Tests et Test-Driven Development (TDD)

Un programme peut sembler correct sur quelques exemples simples et pourtant être faux dans des cas limites ou après une modification ultérieure.
Les **tests** permettent de vérifier automatiquement que le code respecte son comportement attendu, et surtout que ce comportement **reste correct dans le temps**.

Tester ne consiste pas à prouver que le programme est parfait, mais à **réduire le risque d'erreur** et à détecter les problèmes le plus tôt possible.

Pourquoi écrire des tests ?

Les tests sont utiles lorsqu'ils permettent de :

- détecter une erreur avant l'utilisateur final,
- éviter les régressions lors d'une modification ou d'un refactoring,
- documenter le comportement attendu du code de manière exécutable,
- faciliter l'évolution du code en toute confiance.

Dans un projet réel, les tests sont souvent exécutés automatiquement à chaque modification (intégration continue).

Qu'est-ce qu'un bon test ?

Un bon test est :

- **déterministe** : il produit toujours le même résultat dans les mêmes conditions,
- **rapide** : il doit pouvoir être exécuté fréquemment,
- **isolé** : il ne dépend pas d'un état global caché,
- **clair** : on comprend facilement ce qui est testé et pourquoi,
- **localisé** : en cas d'échec, la cause est identifiable rapidement.

Grandes catégories de tests

Tests unitaires

Un **test unitaire** vérifie une fonction ou une classe en isolation.

- sans I/O,

- sans accès réseau,
- sans dépendance matérielle.

Ils sont rapides et très précis.
Ils sont idéaux pour tester : - fonctions mathématiques, - algorithmes, - structures de données.

Tests d'intégration

Un **test d'intégration** vérifie l'interaction entre plusieurs composants :

- lecture de fichiers,
- chargement de ressources,
- threads,
- communication entre modules.

Ils sont plus lents mais plus proches du comportement réel.

Tests de non-régression

Un **test de non-régression** est ajouté après la correction d'un bug.

- il reproduit un cas qui a déjà échoué,
- il garantit que ce bug ne réapparaîtra pas.

Ces tests sont extrêmement précieux sur le long terme.

Structure d'un test : Arrange / Act / Assert

Un test lisible suit généralement la structure suivante :

1. **Arrange** : préparation des données,
2. **Act** : appel du code testé,
3. **Assert** : vérification du résultat.

Exemple :

```
// Arrange
float x = -1.0f;

// Act
float y = clamp(x, 0.0f, 1.0f);

// Assert
assert(y == 0.0f);
```

Cette structure améliore la lisibilité et la maintenance des tests.

Quels cas faut-il tester ?

Pour une fonction donnée, il est recommandé de tester :

1. le **cas nominal** (utilisation normale),
2. les **cas limites** (bornes, tailles 0 ou 1, valeurs extrêmes),
3. les **cas d'erreur** (préconditions violées, entrées invalides).

Tester uniquement le cas nominal est rarement suffisant.

Outil de test minimaliste (sans framework)

On peut écrire des tests avec `assert`, mais il est souvent utile d'avoir des messages plus explicites, notamment pour les flottants.

```
#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 Exemple guidé : tests unitaires pour `clamp`

Spécification attendue

La fonction `clamp(x, a, b)` :

- retourne `a` si `x < a`,
- retourne `b` si `x > b`,
- retourne `x` sinon.

Précondition : `a <= b`.

Tests

```
#include <cassert>

float clamp(float x, float a, float b);

int main()
{
    // cas nominal
    assert(clamp(0.5f, 0.0f, 1.0f) == 0.5f);

    // cas limites
    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 de précondition (doit échouer en debug)
    // clamp(0.0f, 1.0f, 0.0f);
}
```

Implémentation :

```
#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;
}
```

La précondition relève ici du **contrat** : sa violation est une erreur de programmation.

8.6 Test-Driven Development (TDD)

Le **TDD** est une méthodologie dans laquelle le code est écrit **en réponse à des tests**. Elle vise à transformer le besoin fonctionnel en comportement vérifiable.

Boucle TDD : Red -> Green -> Refactor

1. **Red** : écrire un test qui échoue,
2. **Green** : écrire le code minimal pour faire passer le test,
3. **Refactor** : améliorer le code sans casser les tests.

Cette boucle est répétée fréquemment.

Intérêts du TDD

Le TDD :

- force à clarifier l'API dès le départ,
- encourage des fonctions courtes et testables,
- limite la sur-ingénierie (YAGNI),
- rend les refactorisations beaucoup plus sûres.

8.7 Exemple TDD : normalisation d'un vecteur 3D

Spécification

- si `v` est non nul, `normalize(v)` retourne un vecteur de norme 1,
- la direction est conservée,
- précondition : `norm(v) > 0`.

Étape 1 : test (Red)

```
#include <cassert>
#include <cmath>

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

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

vec3 normalize(vec3 const& v);

int main()
{
    vec3 v{3.0f, 0.0f, 4.0f};
    vec3 u = normalize(v);

    assert(std::abs(norm(u) - 1.0f) < 1e-6f);

    float dot = v.x*u.x + v.y*u.y + v.z*u.z;
    assert(dot > 0.0f);
}
```

Étape 2 : implémentation minimale (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};
}
```

Étape 3 : refactor (Refactor)

Ensuite, on peut :

- factoriser `norm2`,
- améliorer les performances,
- ajouter des tests de non-régression.

Conclusion sur les tests et le TDD

Les tests constituent une **vérification automatique du contrat** d'une fonction. Le TDD fournit une méthodologie simple pour écrire du code :

> définir le comportement -> le vérifier automatiquement -> améliorer l'implémentation en confiance.

Utilisés correctement, les tests rendent le code plus fiable, plus lisible et plus facile à faire évoluer.

Test des cas invalides

Tester uniquement les cas valides est insuffisant : un code robuste doit également **détecter correctement les usages invalides**. Il est donc essentiel d'écrire des tests qui vérifient que :

- les **préconditions violées** sont bien détectées (assertion, exception, erreur retournée),
- les **entrées invalides** ne produisent pas de résultats silencieusement incorrects,
- le programme **échoue de manière contrôlée et explicite**, plutôt que de continuer dans un état incohérent.

Ces tests négatifs permettent de s'assurer que le **contrat du code est réellement respecté**, et pas seulement dans les cas idéaux. Ils sont particulièrement importants lors des refactorisations : un changement interne ne doit jamais transformer une erreur détectée en comportement silencieux.

Selon la politique de gestion d'erreurs choisie, un test peut vérifier :

- qu'une assertion échoue en mode debug,
- qu'une exception est levée,
- ou qu'un type résultat signale explicitement une erreur.

En pratique, **tester les cas invalides est souvent aussi important que tester les cas valides**, car c'est précisément dans ces situations que les bugs les plus coûteux apparaissent.

Très bien. Voici **un cas d'exemple concret**, **pédagogique**, et **cohérent avec ton chapitre**, montrant **comment tester un cas invalide**. Tu peux l'insérer **juste après le paragraphe "Tester aussi les cas invalides"**.

Je te propose **deux variantes complémentaires** :

1. cas invalide détecté par `assert` (erreur de programmation),
2. cas invalide détecté par retour d'erreur (erreur d'usage).

Exemple : tester un cas invalide détecté par `assert`

On reprend la fonction `normalize(v)` vue précédemment. Sa **précondition** est que le vecteur ne soit pas nul.

```
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};
}
```

Il est important de vérifier que cette précondition est **effectivement détectée**.

```
// Test négatif : violation de précondition (doit échouer en debug)
int main()
{
    vec3 zero{0.0f, 0.0f, 0.0f};

    // Ce test n'est pas destiné à "passer" :
    // en mode debug, l'assertion doit se déclencher.
    // normalize(zero);
}
```

Remarque :

- ce test est volontairement **commenté** dans un binaire de test classique,
- il est souvent activé séparément ou vérifié manuellement,
- son rôle est de documenter explicitement le **comportement attendu en cas d'usage invalide**.

Exemple : tester un cas invalide avec gestion d'erreur explicite

Si l'on souhaite gérer les entrées invalides sans faire échouer le programme, on peut utiliser un type résultat.

```
#include <optional>

std::optional<vec3> normalize_safe(vec3 const& v)
{
    float n = norm(v);
    if (n <= 0.0f)
        return std::nullopt;

    return vec3{v.x / n, v.y / n, v.z / n};
}
```

Test correspondant :

```
#include <cassert>

int main()
{
    vec3 zero{0.0f, 0.0f, 0.0f};

    auto r = normalize_safe(zero);
    assert(!r.has_value()); // le cas invalide est bien détecté
}
```

Ici, le test vérifie explicitement que :

- l'entrée invalide est reconnue,
- aucune valeur incorrecte n'est produite.

Création des tests

La création de tests exhaustifs est souvent une tâche **répétitive** et **chronophage**. Pour une fonction ou une API non triviale, il faut généralement couvrir :

- les cas nominaux,

- les cas limites,
- les entrées invalides,
- et parfois de nombreuses combinaisons de paramètres.

De plus, lorsque le code évolue (refactorisation, changement d'API, ajout de paramètres), les tests doivent être **mis à jour** afin de rester cohérents avec le nouveau contrat. Cette phase de maintenance peut représenter une part importante du temps de développement.

Dans ce contexte, les outils de **génération de code assistée par IA** peuvent être utilisés pour **accélérer et faciliter** la mise en place de batteries de tests. Ils sont particulièrement utiles pour :

- générer rapidement des tests unitaires systématiques à partir d'une spécification claire,
- proposer des tests de cas limites ou négatifs souvent oubliés,
- aider à adapter ou régénérer des tests après une modification du code,
- explorer automatiquement différentes combinaisons d'entrées.

8.8 Gestion des erreurs : principes et méthodologie

Un programme robuste ne se contente pas de détecter les erreurs : il doit **les classer**, **les signaler correctement**, et **permettre à l'appelant de réagir** de manière appropriée.

La gestion des erreurs fait partie intégrante du **design** du code et de son **API**.

Pourquoi une gestion explicite des erreurs ?

Sans stratégie claire de gestion des erreurs, on obtient :

- des erreurs silencieuses,
- des comportements indéfinis,
- des états internes incohérents,
- des bugs difficiles à reproduire.

Une bonne gestion des erreurs permet :

- de rendre les échecs **visibles et compréhensibles**,
- de séparer le code nominal du code d'erreur,
- de tester explicitement les comportements invalides,
- de renforcer le contrat entre appelant et fonction.

Deux grandes catégories d'erreurs

La première étape consiste à **distinguer la nature de l'erreur**.

1. Erreurs de programmation (bugs)

Ce sont des situations qui **ne devraient jamais arriver** si le code est correctement utilisé.
Exemples :

- violation d'un invariant,
- index hors limites,
- pointeur nul inattendu,
- précondition non respectée.

Ces erreurs indiquent un **bug**.
Traitement recommandé :

- `assert`,
- `static_assert`,
- ou arrêt immédiat du programme.

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

Ces erreurs ne sont généralement **pas récupérables**.

2. Erreurs d'usage ou d'environnement

Ce sont des situations **prévisibles**, même si le code est correct.
Exemples :

- fichier absent,
- données mal formées,
- entrée utilisateur invalide,
- ressource matérielle indisponible.

Ces erreurs doivent être **signalées à l'appelant**.
Traitement recommandé :

- exceptions,
- codes de retour,
- types résultats (`optional`, `expected`, `Result`).

Stratégies de gestion des erreurs en C++

Le choix d'une stratégie dépend :

- du type d'erreur,
- du contexte (bibliothèque, application, temps réel),
- des contraintes de performance et de lisibilité.

1. Exceptions

Les exceptions permettent de **séparer clairement le code nominal du code d'erreur**.

```
float parse_float(std::string const& s)
{
    return std::stof(s); // peut lever une exception
}
```

Avantages :

- code nominal lisible,
- propagation automatique de l'erreur,
- adaptées aux erreurs rares.

Inconvénients :

- coût potentiel (selon contexte),
- contrôle du flux moins explicite,
- parfois interdites en bas niveau / temps réel.

À utiliser avec **discipline**, et à documenter clairement.

2. Codes de retour

Approche historique et explicite.

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

Avantages :

- simple,
- pas d'exception,
- contrôle explicite.

Inconvénients :

- facile à oublier de vérifier,
- peu expressif sans structure associée.

3. Types résultats (`optional`, `expected`, `Result`)

Approche moderne et expressive.

```
std::optional<float> parse_float_safe(std::string const& s);
```

Ou avec information d'erreur :

```
std::expected<float, ParseError> parse_float(std::string const& s);
```

Avantages :

- rend l'erreur explicite dans le type,
- force l'appelant à la traiter,
- très testable.

Souvent le meilleur compromis pour les API modernes.

Exemple complet : API robuste avec type résultat

```
#include <fstream>
#include <optional>
#include <string>
#include <vector>

struct ReadError {
    enum class Code { FileNotFound, ParseError };
    Code code;
    std::string message;
    int line = -1;
};

template <typename T>
struct Result {
    std::optional<T> value;
    std::optional<ReadError> error;

    static Result ok(T v) { return {std::move(v), std::nullopt}; }
    static Result fail(ReadError e) { return {std::nullopt, std::move(e)}; }
};
```

Lecture d'un fichier contenant un flottant par ligne :

```
Result<std::vector<float>> read_floats(std::string const& filename)
{
    std::ifstream file(filename);
    if (!file.is_open()) {
        return Result<std::vector<float>>::fail(
            {ReadError::Code::FileNotFound, "Impossible d'ouvrir le fichier"});
    }

    std::vector<float> values;
    std::string line;
    int line_id = 0;

    while (std::getline(file, line)) {
        ++line_id;
        try {
            values.push_back(std::stof(line));
        } catch (...) {
            return Result<std::vector<float>>::fail(
                {ReadError::Code::ParseError, "Erreur de parsing", line_id});
        }
    }

    return Result<std::vector<float>>::ok(std::move(values));
}
```

Test minimal :

```
auto r = read_floats("data.txt");
assert(r.value.has_value() || r.error.has_value());
```

Lien avec le contrat et les tests

- les **assertions** vérifient les erreurs de programmation,
- les **types résultats / exceptions** gèrent les erreurs récupérables,
- les **tests négatifs** vérifient que les erreurs sont bien détectées,
- le **contrat** documente ce qui relève de l'un ou de l'autre.

Voici une **version enrichie et pédagogique** de ta section **Bonnes pratiques pour la conception d'API**, avec **des exemples concrets "mauvais / meilleur"** pour chaque principe, tout en restant cohérente avec le reste de `08-methodology.md`.

Tu peux **remplacer intégralement** ta section actuelle par celle-ci.

8.9 Bonnes pratiques pour la conception d'API

Une **API** (*Application Programming Interface*) est l'**interface de communication** entre un morceau de code et ses utilisateurs (autres fonctions, autres modules, ou autres développeurs). Elle décrit **comment utiliser le code**, quelles opérations sont disponibles, quels paramètres sont attendus, et quels résultats ou erreurs peuvent être produits.

En C++, une **API** correspond le plus souvent à l'**ensemble des déclarations visibles dans les fichiers d'en-tête** (`.hpp`).
Ces fichiers décrivent **ce que le code permet de faire**, sans exposer **comment il le fait**.

Concrètement, une API C++ est constituée de : - fonctions et leurs signatures, - classes et leurs méthodes publiques, - types (structures, énumérations, alias), - constantes et namespaces exposés.

L'utilisateur de l'API n'a besoin de lire que les fichiers d'en-tête pour comprendre : - comment appeler une fonction, - quels paramètres fournir, - quelles valeurs ou erreurs attendre, - et quelles règles (préconditions) doivent être respectées.

Les fichiers source (`.cpp`) contiennent l'implémentation interne et peuvent évoluer librement tant que l'API, définie par les en-têtes, reste inchangée.
Ainsi, en C++, **concevoir une bonne API revient essentiellement à concevoir de bons fichiers d'en-tête** : clairs, cohérents, et difficiles à mal utiliser.

Objectifs d'une bonne API

Une API bien conçue doit être :

- **claire** : difficile à mal utiliser,
- **prévisible** : comportements cohérents dans des situations similaires,
- **documentée par le type** : les types expriment les contraintes,
- **testable** : facile à utiliser dans des tests unitaires,
- **stable** : les changements ne cassent pas inutilement le code existant.

Rendre les erreurs explicites dans l'API

Une API doit indiquer clairement **comment les erreurs sont signalées**.

Mauvais exemple (erreur silencieuse)

```
float normalize(vec3 const& v); // que se passe-t-il si v est nul ?
```

Ici :

- le contrat est implicite,
- l'utilisateur peut appeler la fonction sans savoir qu'elle est invalide,
- le comportement en cas d'erreur est ambigu.

Exemple avec type résultat explicite

```
std::optional<vec3> normalize(vec3 const& v);
```

Utilisation :

```
auto r = normalize(v);
if (!r) {
    // cas invalide : v est nul
}
```

L'erreur fait partie de l'API : elle **ne peut pas être ignorée accidentellement**.

Exemple avec précondition explicite (erreur de programmation)

```
vec3 normalize(vec3 const& v); // précondition : norm(v) > 0
```

Ici :

- l'appelant est responsable,
- la violation est une **erreur de programmation**,
- elle peut être détectée via `assert`.

Choisir explicitement si l'erreur est récupérable ou non.

Préférer des types expressifs

Les types doivent porter le sens, pas seulement les valeurs.

À éviter : paramètres ambigus

```
void load(int mode); // que signifie mode ?
```

L'API permet des valeurs invalides (`mode = 42`).

Préférer : types forts et explicites

```
enum class LoadMode { Fast, Safe };
void load(LoadMode mode);
```

Utilisation :

```
load(LoadMode::Fast);
```

Avantages :

- impossible de passer une valeur invalide,
- l'intention est claire,
- les erreurs sont détectées à la compilation.

Autre exemple : bool ambigu vs type dédié

```
void draw(bool wireframe); // que signifie true ?
```

Meilleur design :

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

Limiter les états invalides

Une bonne API rend les états invalides **impossibles ou difficiles à représenter**.

Exemple problématique : état partiellement valide

```
struct Image {
    unsigned char* data;
    int width;
    int height;
};
```

Ici, rien n'empêche :

- `data == nullptr`,
- `width <= 0`,
- incohérences internes.

Meilleur exemple : invariant établi par le constructeur

```
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;
};
```

Avantages :

- l'objet est toujours valide après construction,
- les invariants sont centralisés,
- l'utilisateur ne peut pas créer un état incohérent.

Séparer interface et implémentation

L'API doit exposer **ce que fait le code**, pas **comment il le fait**.

Header (`.hpp`) : interface

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

Source (`.cpp`) : implémentation

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

void Image::clear()
```

```
{
    // détails internes invisibles pour l'utilisateur
}
```

Avantages :

- liberté de changer l'implémentation,
- compilation plus rapide,
- API plus stable.

Éviter les effets de bord cachés

Une fonction ne doit pas modifier des états globaux de manière inattendue.

Mauvais exemple

```
void render()
{
    global_state.counter++; // effet de bord caché
}
```

Meilleur exemple

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

Les dépendances sont explicites et testables.

Règles pratiques de conception d'API

- documenter clairement les **préconditions** et **postconditions**,
- rendre les erreurs visibles dans le type ou le comportement,
- éviter les paramètres ambigus (`bool`, `int` non documentés),
- préférer des fonctions petites et orthogonales,
- tester l'API comme si on était un **utilisateur externe**,
- considérer que l'API est plus difficile à modifier que l'implémentation.

Idée clé à retenir

> **Une bonne API empêche les erreurs avant même l'exécution du programme.**

Elle guide l'utilisateur vers le bon usage, rend les erreurs explicites, et facilite les tests, la maintenance et l'évolution du code.