Chapitre 1. Le langage C++ - introduction

Objectifs pédagogiques

À la fin de ce chapitre, l'étudiant devra être capable de :

- Expliquer l'historique et la philosophie du langage C++.
- Comparer le C++ à d'autres langages de programmation.
- Identifier les domaines d'application industrielle du C++.
- Connaître les différentes **normes du langage (C++98 à C++20)**.
- Comprendre les principales **nouveautés introduites à chaque version**.

1. Historique du langage C++

- **Créateur :** Bjarne Stroustrup (Bell Labs, 1979).
- **Motivation :** ajouter la **programmation orientée objet** (POO) au langage C.
- **Nom initial :** "C with Classes", car il étendait le langage C avec des classes.

Évolution

Année	Version	Caractéristiques principales
1979–1983	C with Classes	Classes, encapsulation, héritage
1985	C++ (1ère édition)	Nom officiel, surcharge d'opérateurs, fonctions inline
1990	C++ 2.0	Héritage multiple, classes abstraites, templates
1998	**C++98**	Première norme ISO
2003	**C++03**	Corrections et précisions de C++98
2011	**C++11**	Révolution du langage (lambda, auto, smart pointers, threads)
2014	**C++14**	Améliorations mineures de C++11
2017	**C++17**	std::optional, std::variant, filesystem, structured bindings
2020	**C++20**	Concepts, ranges, coroutines, modules

2. Comparaison avec d'autres langages

Langage	Paradigme	Performance	Sécurité	Facilité	Domaines
C	Procédural	⋆ ⋆ ⋆ ⋆	⚠	⋆ ⋆	Systèmes, OS, embarqué

C++	Multi-paradigme (procédural, objet, générique, fonctionnel)	⭐ ⭐ ⭐ ⭐ ⭐	⭐ ⭐	⭐ ⭐	Industrie, IA, jeux, temps réel
Java	Orienté objet	⭐ ⭐	⭐ ⭐ ⭐ ⭐	⭐ ⭐ ⭐	Web, mobile, entreprises
Python	Interprété, objet	⭐	⭐ ⭐ ⭐ ⭐	⭐ ⭐ ⭐ ⭐ ⭐	IA, data, scripting
Rust	Sûreté mémoire, compilé	⭐ ⭐ ⭐ ⭐	⭐ ⭐ ⭐ ⭐ ⭐	⭐ ⭐	Systèmes, sécurité, embarqué

C++ combine :

- **La performance du C**,
- **La modularité de la POO**,
- Et **la puissance des templates et de la métaprogrammation**.

Il reste **le langage de référence** pour les applications critiques en performance.

3. Domaines d'utilisation dans l'industrie

Exemples concrets :

- **Systèmes embarqués et temps réel :** automobile, spatial, défense.
- **Jeux vidéo et moteurs 3D :** Unreal Engine, Unity (backend).
- **Finance et haute fréquence :** trading algorithmique, calcul intensif.
- **IA et traitement du signal :** bibliothèques comme TensorRT, OpenCV.
- **Systèmes d'exploitation :** Windows, macOS, Linux (parties écrites en C/C++).
- **Simulation scientifique :** calcul haute performance (HPC).

4. Normes et versions du langage

Le langage **C++** a évolué à travers plusieurs normes officielles définies par l'ISO (International Organization for Standardization). Chaque version introduit des fonctionnalités nouvelles, corrige des imperfections et modernise le langage pour répondre aux besoins des développeurs modernes.

4.1.C++98 / C++03

Avant 1998, chaque compilateur (Borland, Microsoft, GNU, etc.) avait ses propres variantes du C++. En 1998, le langage est enfin normalisé (ISO/IEC 14882:1998) : c'est le C++98, la première version officielle.

Nouveautés clés :

- **STL (Standard Template Library)** : grande avancée du C++.
  Elle introduit :

- Des **conteneurs génériques** (std::vector, std::list, std::map),
- Des **itérateurs** pour parcourir ces conteneurs,
- Des **algorithmes génériques** (sort, find, accumulate, etc.).

- **C++03** : version révisée du C++98, publiée en 2003.
  - Corrige des incohérences du standard précédent.
  - Améliore la compatibilité entre compilateurs.
  - Raffine la gestion des **templates** et des **exceptions**.

But : stabiliser le langage et définir une base commune.

4.2.C++11 — La "modernisation"

Cette version transforme le langage pour le rendre **plus sûr, plus simple et plus expressif**, tout en restant performant.

Principales nouveautés :

1. **Mot-clé auto :** Déduction automatique du type :

```
auto x = 5;          // int
auto y = 3.14;       // double
auto it = v.begin(); // type d'itérateur
```

2. nullptr

Remplace le vieux NULL du C :

```
int* p = nullptr;  // plus sûr que int* p = NULL;
```

3. constexpr

Permet d'évaluer une expression **à la compilation** :

```
constexpr int carre(int x) { return x * x; }
int a[carre(3)]; // tableau de taille 9, connu à la compilation
```

4. Fonctions lambda

Fonctions anonymes, idéales pour les callbacks :

```
auto somme = [](int a, int b) { return a + b; };
cout << somme(2, 3); // affiche 5
```

5. Boucles for simplifiées

```
std::vector<int> v = {1, 2, 3};
for (auto x : v)
   std::cout << x << " ";
```

6. Pointeurs intelligents

Gestion automatique de la mémoire :

```
#include <memory>
std::unique_ptr<int> p1 = std::make_unique<int>(10);
std::shared_ptr<int> p2 = std::make_shared<int>(20);
```

7. Threads et synchronization

```
#include <thread>
void tache() { std::cout << "Bonjour\n"; }
std::thread t(tache);
t.join();
```

But **:** moderniser le langage et éviter les erreurs de gestion mémoire.

4.3. C++14 — Améliorations et simplifications

C++14 consolide C++11 avec des ajustements ergonomiques et de légères extensions.

Nouveautés :

1. Type de retour automatique pour les fonctions :

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

2. Lambdas généralisées :

- Meilleure capture des variables locales.
- Retour implicite sans écrire ->.

3. Amélioration des templates et constexpr.

But : fluidifier la syntaxe et simplifier la vie du programmeur.

4.4. C++17 — Vers un langage plus expressif

C++17 continue la modernisation du langage avec des fonctionnalités de haut niveau.

Nouveautés majeures :

1. std::optional, std::variant, std::any

Manipulation sécurisée des valeurs optionnelles :

```
#include <optional>
std::optional<int> getVal(bool ok) {
   if (ok) return 42;
   return std::nullopt;
}
```

2. if constexpr

Permet des conditions **à la compilation** :

```
template<typename T>
void f(T x) {
   if constexpr (std::is_integral_v<T>)
      std::cout << "Entier\n";
   else
      std::cout << "Autre type\n";
}
```

3. Destructuring **:**

```
auto [a, b] = std::pair<int, int>(1, 2);
```

4. Filesystem intégré :

Manipulation simple des fichiers :

```
#include <filesystem>
namespace fs = std::filesystem;
for (auto& p : fs::directory_iterator("."))
    std::cout << p.path() << "\n";
```

But : introduire des abstractions plus puissantes et sûres.

4.5. C++20 — Une révolution moderne

C++20 est une refonte profonde du langage. Il introduit des **paradigmes de programmation avancés** : concepts, ranges, modules et coroutines.

Nouveautés clés :

1. Concepts

Contraintes sur les templates → code plus lisible et sûr :

```
template <typename T>
concept Entier = std::is_integral_v<T>;

Entier auto somme(Entier auto a, Entier auto b) {
    return a + b;
}
```

2. **Ranges**

Nouvelle bibliothèque pour manipuler les séquences :

```
#include <ranges>
std::vector<int> v = {1, 2, 3, 4, 5};
for (int x : v | std::ranges::views::filter([](int n){ return n%2==0; }))
    std::cout << x << " "; // affiche 2 4
```

3. **Modules**

Remplacent les #include → compilations plus rapides, dépendances mieux gérées :

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

4. **Coroutines**

Pour écrire du code asynchrone lisible :

```
#include <coroutine>
task<int> f() {
    co_return 42;
```

```
}
```

5. Opérateur de comparaison <=> (Spaceship operator)

Simplifie la définition des comparaisons :

```
auto operator<=>(const Point&) const = default;
```

6. Lambdas et constexpr améliorés

- Lambdas peuvent être constexpr.
- Plus de fonctions utilisables à la compilation.

But : rendre C++ plus **expressif**, **modulaire** et **orienté vers la programmation moderne**.

5. Synthèse : évolution du C++

Version	Année	Objectif principal	Mots-clés emblématiques
C++98 / 03	1998–2003	Standardisation et STL	template, std::vector
C++11	2011	Modernisation	auto, nullptr, lambda, unique_ptr
C++14	2014	Simplification	auto (retour), lambdas généralisées
C++17	2017	Expressivité et sécurité	optional, variant, filesystem
C++20	2020	Révolution conceptuelle	concept, ranges, modules, coroutines

6. Philosophie du C++

Bjarne Stroustrup résume ainsi la philosophie du langage :

"C++ est conçu pour permettre une programmation efficace et abstraite, tout en laissant le contrôle au programmeur."

En pratique :

- Contrôle total sur la mémoire et les performances.
- Possibilité d'écrire du code de haut niveau sans sacrifier l'efficacité bas niveau.
- Favorise la **parcimonie** et la **modularité**, deux notions essentielles en ingénierie logicielle.