Introduction au C++

Intérêt du C++

Performance : compilation native, optimisations fines, accès direct au matériel.

Contrôle mémoire : allocation explicite, mais options modernes (`smart pointers`) réduisent les erreurs.

Polyvalence : supporte paradigmes procédural, orienté-objet, générique et fonctionnel.

Écosystème : bibliothèques matures pour 2D/3D, calcul scientifique, GPU et systèmes.

Domaines d'application

Moteurs de jeu, rendu 3D

Simulations et calcul scientifique

Logiciels embarqués et systèmes

Bibliothèques hautes performances (vision, DL)

La norme C++ a évolué régulièrement pour améliorer le langage et la bibliothèque standard.

C++98 / C++03 : normalisation initiale du langage et de la bibliothèque.

C++11 : « C++ moderne » — `auto`, lambdas, pointeurs intelligents (`std::unique_ptr`, `std::shared_ptr`), boucles étendues.

C++14 / C++17 : améliorations syntaxiques et bibliothèques (`filesystem`, structured bindings, parallélisme partiel).

C++20 : concepts, coroutines, ranges — étapes importantes vers une programmation plus expressive.

C++23 : peaufinage des bibliothèques et simplifications d'usage.

Premier programme C++

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

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

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

- `#include` : inclusion de la bibliothèque standard pour les entrées/sorties.
- `int main() { ... }`: point d'entrée du programme.
- `std::cout << ...`: affichage du texte "Hello, world!" + retour à la ligne.
- `return 0;`: indique que le programme s'est terminé correctement.
- Chaque instruction se termine par un point-virgule `;`.
- Commentaires: // ou /* ... */ pour expliquer le code.
- Indentation, saut de lignes: uniquement pour la lisibilité.

Compilation d'un programme C++

Le code C++ doit être compilé

Compilation = traduction du code source en code machine exécutable par l'ordinateur.

Code exécutable = fichier pouvant être lancé directement par le système d'exploitation.

Rem. Différence avec Python

- Python = code source interprété à la volée par un programme.

Nécessite l'exécution via un interpréteur (ex: `python mon_programme.py`).

- C++ = code source compilé en code machine avant exécution.

Une fois compilé : le programme s'exécute directement par le système d'exploitation/CPU.

Sous Linux/MacOS, compilation via le compilateur `g++` ou `clang`.

```
g++ mon_programme.cpp -o mon_programme
```

mon_programme.cpp: fichier source C++ à compiler.

-o mon_programme: option pour nommer le fichier exécutable généré

Exécution:

Sous Linux/MacOS: `./mon_programme`

Sous Windows: `mon_programme.exe`

Cas spéciaux

Variable = espace mémoire nommé pour stocker une valeur. Types fondamentaux courants en C++ :

- `int`: entier (ex: 42, -7)
- `float`: nombre à virgule simple précision (ex: 3.14f, -0.01f)
- `double`: nombre à virgule double précision (ex: 3.1415926535, -0.0001)
- `char`: caractère (ex: 'a', 'Z', '0')
- `bool`: booléen (valeurs: `true` ou `false`)

```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 << "Âge : " << age << std::endl;
    std::cout << "Taille : " << taille << " m" << std::endl;
    std::cout << "Valeur de pi : " << pi << std::endl;

    return 0;
}
```

Variables : Cas spéciaux

Division entière

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

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

Pour obtenir un résultat décimal, au moins un opérande doit être flottant

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

Mot-clé auto

Le mot-clé `auto` permet au compilateur de déduire le type d'une variable à partir de l'initialisation.

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

Utile pour les types complexes. Mais: Préférez les types explicites pour les types simples.

Variables: Cas spéciaux

Variables non initialisées

Variables fondamentales non initialisées par défaut

Valeur indéfinie avant leur affectation.

```
int a; // contient une valeur indéfinie
// Utiliser 'a' avant affectation => comportement indéfini
```

Bonne pratique : toujours initialiser les variables. `int a = 0;`

Variables constantes

Une variable `const` doit être initialisée lors de sa déclaration et ne peut plus être modifiée ensuite.

```
const float pi = 3.14159f;
// pi = 3.14f; // ERREUR : impossible de modifier une constante
```

Avantages :

Sécurité : protège contre les modifications accidentelles.

Lisibilité : exprime l'intention.

Optimisation : le compilateur peut effectuer des optimisations.

Affichage / lecture formatés : printf et scanf

Affichage formaté via les fonctions C : printf et scanf

(en plus de std::cout et std::cin)

```cpp
#include <cstdio>

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

    // affichage formaté
    printf("Age : %d ans, taille : %.2f m\n", age, taille);

    // lecture formatée
    int x;
    printf("Entrez un entier : ");
    if (scanf("%d", &x) == 1) {
        printf("Vous avez entré : %d\n", x);
    }
    return 0;
}
```

- Affichage via **spécificateurs de format** (`%d`, `%f`, `%s`, ...)

Conteneurs contigus: Tableaux C classiques

Les tableaux C (`T NomTableau[N]`)

Elements stockés dans zone mémoire appelée pile (stack).

```cpp
int tab[5] = {10, 20, 30, 40, 50};
// Indices de 0 à N-1
for (int i = 0; i < 5; i++) {
    std::cout << "tab[" << i << "] = " << tab[i] << std::endl;
}
```

Présentent des limitations et pièges :

Taille fixe, devant être connue à la compilation.

Pas de mise en mémoire de la taille du tableau.

Conversion automatique en pointeur

```cpp
void afficher(int* arr) {
    // Impossible de connaître la taille ici !
    std::cout << "sizeof(arr) = " << sizeof(arr) << std::endl;
    // Taille du pointeur (8 octets)
}

int main() {
    int tab[5] = {1, 2, 3, 4, 5};
    std::cout << "sizeof(tab) = " << sizeof(tab) << std::endl;
    // 5 * sizeof(int) = 20 octets
    afficher(tab);
    // Conversion implicite en int*
    return 0;
}
```

Conteneurs contigus : std::array

std::array<T,N> conteneur à taille fixe. Interface et sécurité d'un conteneur STL.

Inclusion : #include <array>

Taille connue à la compilation : utile pour des petites tailles et lorsque la taille est fixe.

Méthodes : size(), begin()/end(), fill(), at().

```
std::array<int,3> a = {1,2,3};
for (auto &x : a)
    std::cout << x << " ";
std::cout << "\nsize=" << a.size() << std::endl;
a.fill(0);
```

```
// La taille est conservée lors du passage en argument
void afficher(const std::array<int,3>& arr) {
    std::cout << "size=" << arr.size() << std::endl;
    for (const auto& x : arr)
        std::cout << x << " ";
}

int main() {
    std::array<int,3> a = {10, 20, 30};
    afficher(a);  // size=3
}
```

Avantages : interface sûre (méthodes STL), pas de conversion implicite en pointeur comme les tableaux C.

Conteneurs contigus : std::vector

`std::vector` tableau dynamique : sa taille peut croître à l'exécution.

- Classe template std::vector<T> pour un vecteur d'éléments de type T.
- Ajout en fin de tableau : .push_back(val)
- Opérations principales : push_back(), size(), clear(), resize().
- Accès via v[i] ou .at(i) (sécurisé en cas de dépassement de taille).
- Elements stockés dans zone mémoire appelée tas (heap).

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

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    for (auto x : v)
        std::cout << x << " ";
    std::cout << "\nsize=" << v.size()  << std::endl;
    return 0;
}
```

Conditionnelles et boucles

Structure générale :

```
if (condition) {
    // instructions si vrai
} else if (autre_condition) {
    // instructions si autre_condition vraie
} else {
    // instructions par défaut
}
```

{} optionnelles si une seule instruction.

Comparaison : ==, !=, <, >, <=, >=

Logiques : && (et), || (ou), ! (non)

```
int note = 15;

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

Boucle while : répète tant que la condition est vraie

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

Boucle for : initialisation, condition, incrément

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

Boucle for étendue (C++11)

```
std::vector<int> v = {1, 2, 3, 4, 5};
for (int x : v)
    std::cout << x << " ";
// Avec référence (modifie le vecteur)
for (int& x : v)
    x *= 2;
```

Conteneur associatif: std::map

std::map dictionnaire de paires clé/valeur triées par clé

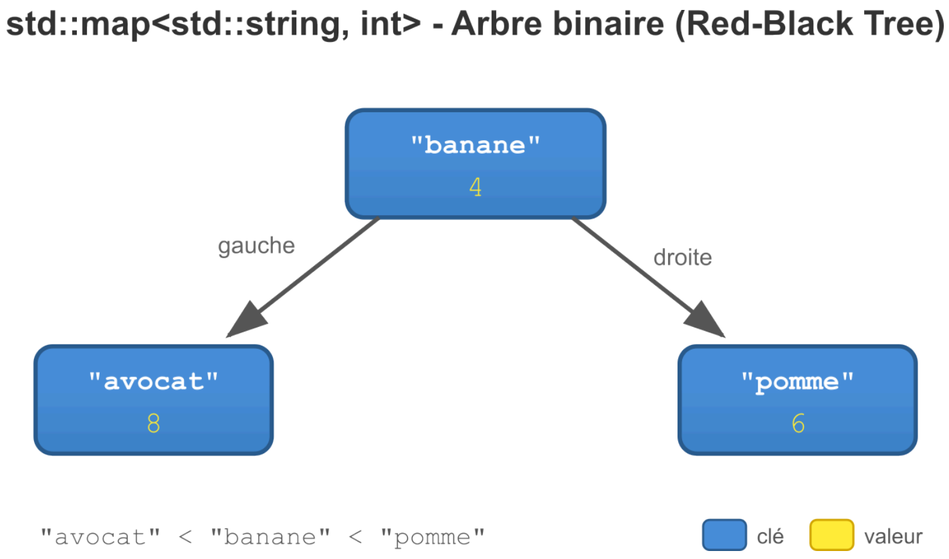
- Tri automatique via l'ordre défini par l'opérateur '<'
- Inclusion : #include <map>
- Recherche en O(log n).

```cpp
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;
}

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



Durée de vie des variables

Durée de vie (scope) déterminée par le bloc d'instructions dans lequel elle est déclarée.

Un bloc est défini par des accolades { ... }.

Variable existe depuis sa déclaration jusqu'à l'accolade fermante } du bloc.

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

```cpp
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 jusqu'à la fin de main()
```

Possibilité de déclarer un variable du même nom dans un sous-bloc :

```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
```

Fonctions

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

On appelle:

Signature d'une fonction = nom + types des arguments + type de retour

Corps de la fonction = ensemble des instructions entre les accolades { ... }

Pas retour: type 'void'

La signature d'une fonction doit toujours être déclarée avant son utilisation.

```
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;
}
```

Fonctions: Surcharge

Possibilité de "surcharger" une fonction

= Plusieurs fonctions avec le même nom, mais arguments différents.

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

// 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;
}
```

Passage d'arguments : par copie

Les arguments des fonctions ont la valeur copiée du paramètre.

On parle de "passage par copie" ou "passage par valeur" ("pass by value")

[+] Les modifications restent locales à la fonction

[-] Pour les gros objets, la copie peut être coûteuse

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

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

```cpp
void multiply(std::vector<float> vec, float s) {
    for (int k = 0; k < vec.size(); ++k)
        vec[k] *= s;
    // Modifie la copie locale
}

int main() {
    std::vector<float> v = {1, 2, 3};
    multiply(v, 2.0f);
    // v est inchangé : {1, 2, 3}
}
```

Passage d'arguments : par référence

En C++, le symbole & permet de passer par **référence** (accès direct, pas de copie)

- Permet de modifier l'argument d'origine
- Évite la copie pour les gros objets

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

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

Dans le cas où on souhaite éviter le cout de la copie sans modifier l'argument: **Référence constante** (`const&`)

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

Bonne pratique : utiliser `const&` pour les gros objets non modifiés.

Rem. const T& arg, ou T const& arg : même signification.

Classes : principes

Une **classe** (ou **struct**) regroupe :
- des **attributs** (données membres)
- des **méthodes** (fonctions membres)

Un **objet** est une instance de la classe.

```
struct vec3 {
    float x, y, z;   // attributs
};

int main() {
    vec3 p1;                              // non initialisé
    vec3 p2 = {1.0f, 2.0f, 5.0f};     // initialisé

    p2.y = -4.0f;                         // modification
    std::cout << p2.x << "," << p2.y << "," << p2.z;
}
```

Rem. Deux mots clés **class** et **struct** pour définir une classe.
- Comportement similaire.
- Par défaut, les membres d'une **struct** sont publics.
- Par défaut, les membres d'une **class** sont privés.

```
struct vec3 {
    float x, y, z;  // public par défaut
};

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

Classes : méthodes

Une classe peut définir des **méthodes** qui manipulent ses attributs.

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

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

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

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

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

    std::cout << p.norm();  // appel méthode

    p.normalize();
}
```

const après méthode : ne modifie pas l'objet

Syntaxe implémentation : *Classe::methode()*

Classes : constructeurs et destructeur

Constructeur : initialise l'objet à sa création

Destructeur : exécuté à la destruction de l'objet

En général: rien à faire sur des objets simples

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

    vec3();               // constructeur par défaut
    vec3(float v);        // constructeur personnalisé
    ~vec3();              // destructeur
};

vec3::vec3() : x(0), y(0), z(0) { }
vec3::vec3(float v) : x(v), y(v), z(v) { }
vec3::~vec3() { std::cout << "Goodbye"; }
```

```
int main() {
    vec3 a;          // appelle vec3()
    vec3 b(1.0f);    // appelle vec3(float)
}   // appelle ~vec3() pour a et b
```

= default : génère l'implémentation automatique

```
struct vec3 {
    float x, y, z;
    vec3() = default;
    ~vec3() = default;
};
```

Fonctions membres vs fonctions externes

Le choix entre **méthode** et **fonction externe** est libre.

```cpp
// Fonction membre (méthode)
struct vec3 {
    float x, y, z;
    float norm() const {
        return std::sqrt(x*x + y*y + z*z);
    }
};

vec3 p = {1, 2, 3};
float n = p.norm();   // appel via objet
```

```cpp
// Fonction externe (non-membre)
struct vec3 {
    float x, y, z;
};

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

vec3 p = {1, 2, 3};
float n = norm(p);    // appel comme fonction
```

Lecture / écriture de fichiers

Bibliothèque `<fstream>` pour manipuler des fichiers :

- `std::ifstream` : lecture (input)
- `std::ofstream` : écriture (output)

```cpp
#include <fstream>

// Écriture
std::ofstream file("data.txt");
if (file.is_open()) {
    file << "Hello" << std::endl;
    file << 1.5f << " " << 2.0f << std::endl;
    file.close();
}
```

```cpp
#include <fstream>

// Lecture
std::ifstream file("data.txt");
if (file) {
    std::string line;
    std::getline(file, line);  // lit une ligne
    float x, y;
    file >> x >> y;            // lit valeurs
    file.close();
}
```

Modes file("data.txt", mode) :

- `std::ios::app` (ajout)
- `std::ios::binary` (binaire)

Organisation du code

Trois types de fichiers :

.hpp Déclarations, signatures de fonctions partagés avec d'autres fichiers

.cpp Implémentation des fonctions et classes décrites dans .hpp

main.cpp Point d'entrée du programme, utilisation des fonctions et classes

```cpp
// vec3.hpp
#pragma once
struct vec3 {
    float x, y, z;
    float norm() const;
};
```

```cpp
// vec3.cpp
#include "vec3.hpp"
#include <cmath>
float vec3::norm() const {
    return std::sqrt(x*x+y*y+z*z);
}
```

```cpp
// main.cpp
#include "vec3.hpp"
int main() {
    vec3 v = {1, 2, 3};
    std::cout << v.norm();
}
```

#pragma once : évite les inclusions multiples

#include "file.hpp" : copie-colle le contenu du fichier

Compilation

Compilation : `.cpp` → compilateur → `.o` → linker → exécutable

```
# Un seul fichier
g++ main.cpp -o programme

# Plusieurs fichiers
g++ -c main.cpp   # main.o
g++ -c vec3.cpp   # vec3.o
g++ main.o vec3.o -o prog
```

CMake : génère Makefile (Linux) ou projet VS (Windows)

```
mkdir build && cd build
cmake ..
make   # Linux/MacOS
```

Linux/MacOS : `g++` ou `clang++`
Windows : Visual Studio (MSVC)