Utilisation des outils de base de la bibliothèque standard du C++

C++ L3 maths

28 février 2025

Introduction

La bibliothèque standard du C++ offre de nombreux outils facilitant le développement :

- **Containers** : `vector`, `list`, `map`, `set`, etc.
- **Pointeurs** : gestion des pointeurs intelligents.
- **Streams** : gestion des entrées/sorties, notamment pour la console et les fichiers.
- **Algorithmes** : tri, recherche, manipulation des containers.
- **Strings** : manipulation des chaînes de caractères avec `std::string`.

Les Containers

Voici un exemple d'utilisation de `std::vector` :

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    for (auto v : vec) {
        std::cout << v << "␣";
    }
    return 0;
}
```

Explication du code C++

Le code suivant :

```
for (auto v : vec) {
    std::cout << v << "␣"; // Affiche chaque élément suivi d'un espace
}
return 0;
```

- **Boucle range-based :** La syntaxe `for (auto v : vec)` permet d'itérer automatiquement sur chaque élément du conteneur `vec`. Le mot-clé `auto` permet au compilateur de déduire le type de `v` en fonction des éléments stockés dans `vec`, ici `int`.

Les Pointeurs

Utilisation d'un pointeur intelligent `unique_ptr` :

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

int main() {
    std::unique_ptr<int> uptr = std::make_unique<int>(20);
    std::cout << *uptr << std::endl;
    return 0;
}
```

Les unique_ptr : Utilisation et règles

Les `unique_ptr` servent à gérer la durée de vie d'un objet alloué dynamiquement tout en assurant qu'un seul pointeur possède cet objet à un moment donné.

Exemple accepté (déplacement) :

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

int main() {
    std::unique_ptr<int> p1 = std::make_unique<int>(42);
    // Déplacement du unique_ptr vers p2
    std::unique_ptr<int> p2 = std::move(p1);
    std::cout << *p2 << std::endl;  // OK, affiche 42
    return 0;
}
```

Exemple non accepté (copie interdite) :

```
#include <memory>

int main() {
    std::unique_ptr<int> p1 = std::make_unique<int>(42);
    // Tentative de copie (non compilable)
    std::unique_ptr<int> p2 = p1;  // Erreur de compilation !
    return 0;
}
```

Les Pointeurs Partagés

Exemple d'utilisation d'un pointeur partagé `shared_ptr` :

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

int main() {
    std::shared_ptr<int> sptr1 = std::make_shared<int>(30);
    std::shared_ptr<int> sptr2 = sptr1; // sptr1 et sptr2 partagent le meme
        objet
    std::cout << *sptr1 << "␣" << *sptr2 << std::endl;
    return 0;
}
```

Les Streams

Exemple de lecture et écriture via les streams :

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

int main() {
    // Sortie sur la console
    std::cout << "Hello, world!" << std::endl;

    // Écriture dans un fichier
    std::ofstream fichier("exemple.txt");
    if (fichier) {
        fichier << "Ligne de texte dans le fichier" << std::endl;
        fichier.close();
    }
    return 0;
}
```

Les Chaînes de Caractères

Manipulation des chaînes avec `std::string` :

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

int main() {
    std::string s = "Bonjour";
    s += "␣le␣monde!";
    std::cout << s << std::endl;
    return 0;
}
```

Les Algorithmes

Utilisation des algorithmes de la STL, par exemple le tri :

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

int main() {
    std::vector<int> vec = {5, 2, 8, 1, 9};
    std::sort(vec.begin(), vec.end());
    for (int v : vec) {
        std::cout << v << "␣";
    }
    return 0;
}
```

Lecture d'un Fichier

Exemple de lecture d'un fichier ligne par ligne :

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

int main() {
    std::ifstream fichier("exemple.txt");
    std::string ligne;
    while (std::getline(fichier, ligne)) {
        std::cout << ligne << std::endl;
    }
    return 0;
}
```

Combinaison : Containers et Algorithmes

Lire des données depuis un fichier et les stocker dans un container pour les trier :

```
#include <fstream>
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>

int main() {
    std::ifstream fichier("nombres.txt");
    std::vector<int> nombres;
    int n;
    while (fichier >> n) {
        nombres.push_back(n);
    }
    std::sort(nombres.begin(), nombres.end());
    for (auto nombre : nombres)
        std::cout << nombre << "␣";
    return 0;
}
```

Hashmaps : Introduction

Les hashmaps, implémentées par `std::unordered_map`, permettent de stocker des paires clé-valeur avec un accès en temps constant en moyenne. Elles sont idéales pour :

- La recherche rapide
- L'insertion et la suppression efficaces
- La gestion dynamique des données associatives

Hashmaps : Exemple d'Utilisation

Voici un exemple d'utilisation de `std::unordered_map` :

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

int main() {
    std::unordered_map<std::string, int> map;
    map["un"] = 1;
    map["deux"] = 2;
    map["trois"] = 3;
    for (const auto &pair : map) {
        std::cout << pair.first << " : " << pair.second << std::endl;
    }
    return 0;
}
```

Recherche dans une unordered_map

Exemple de recherche d'une clé dans une `std::unordered_map` :

```
int main() {
    std::unordered_map<std::string, int> map;
    map["un"] = 1;
    map["deux"] = 2;
    map["trois"] = 3;

    // Recherche de la clé "deux"
    auto it = map.find("deux");
    if (it != map.end()) {
        std::cout << "Trouvé␣:␣" << it->first << "␣->␣" << it->second << std::
            endl;
    } else {
        std::cout << "Clé␣non␣trouvée" << std::endl;
    }
    return 0;
}
```

- La méthode `find` retourne un itérateur pointant sur l'élément recherché.
- Si la clé n'existe pas dans la hashmap, `find` renvoie `map.end()`.
- L'itérateur permet d'accéder à la clé avec `it->first` et à la valeur associée avec `it->second`.

Utilisation de la bibliothèque Chrono

La bibliothèque <chrono> permet de mesurer le temps d'exécution d'un morceau de code.

```
#include <chrono>
int main() {
    auto start = std::chrono::high_resolution_clock::now();
    // Code à mesurer ici
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> diff = end - start;
    std::cout << "Durée d'exécution : "
              << diff.count() << " secondes."
              << std::endl;
    return 0;
}
```

- **Mesure du temps :** `std::chrono::high_resolution_clock` fournit l'heure précise pour débuter et terminer la mesure.
- **Calcul de la durée :** La différence entre les deux instants est stockée dans un objet `std::chrono::duration<double>` qui représente la durée en secondes.
- **Affichage :** La méthode `diff.count()` renvoie la durée écoulée, ici affichée en secondes.

Utilisation de std : :optional

La classe `std::optional` permet de représenter une valeur qui peut être présente ou non.

```
#include <optional>
std::optional<int> findValue(bool condition) {
    if (condition)
        return 42;          // Valeur présente
    else
        return std::nullopt; // Aucune valeur
}

int main() {
    auto val = findValue(true);
    if (val.has_value()) {
        std::cout << "La valeur est : " << val.value() << std::endl;
    } else {
        std::cout << "Aucune valeur n'a été retournée." << std::endl
             ;
    }
    return 0;
}
```

- **std : :optional<T>** encapsule une valeur de type `T` ou indique son absence.
- Permet d'éviter l'utilisation de pointeurs nuls ou de valeurs sentinelles.
- Méthodes utiles : `has_value()` et `value()`.

Utilisation de std : :variant

La classe `std::variant` est un type discriminé (union sécurisée) qui peut contenir une valeur parmi un ensemble de types prédéfinis.

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

int main() {
    std::variant<int, std::string> data;

    data = 10; // data contient un int
    std::cout << "Valeur␣int␣:␣" << std::get<int>(data) << std::endl
         ;

    data = "Bonjour"; // data contient maintenant un std::string
    std::cout << "Valeur␣string␣:␣" << std::get<std::string>(data)
         << std::endl;

    return 0;
}
```

- Permet de stocker des valeurs de types différents dans une même variable.
- Accès sécurisé grâce à `std::get<T>` qui vérifie le type actuellement stocké.
- Idéal pour les cas où une variable peut représenter plusieurs types possibles.

Tester le type contenu dans std : :variant I

Pour vérifier le type actuellement stocké dans un `std::variant`, vous pouvez utiliser :

- **std : :holds_alternative<T>(variant)** : renvoie `true` si le type T est stocké.
- **std : :visit** : permet d'appliquer une fonction à la valeur contenue, quel que soit son type.

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

int main() {
    std::variant<int, std::string> data;
    data = "Bonjour";

    // Utilisation de std::holds_alternative
    if (std::holds_alternative<int>(data)) {
        std::cout << "data␣contient␣un␣int." << std::endl;
    } else if (std::holds_alternative<std::string>(data)) {
        std::cout << "data␣contient␣un␣std::string." << std::endl;
    }

    // Utilisation de std::visit avec un lambda
    std::visit([](auto&& arg) {
        using T = std::decay_t<decltype(arg)>;
```

Tester le type contenu dans std : :variant II

```
        if constexpr (std::is_same_v<T, int>)
            std::cout << "data est un int : " << arg << std::endl;
        else if constexpr (std::is_same_v<T, std::string>)
            std::cout << "data est un std::string : " << arg << std::endl;
    }, data);

    return 0;
}
```

Introduction aux lambdas en C++

Les lambdas sont des fonctions anonymes que l'on peut définir directement sur place. Elles sont particulièrement utiles pour passer des fonctions comme arguments aux algorithmes.

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // Utilisation d'une lambda pour afficher chaque élément du vecteur
    std::for_each(vec.begin(), vec.end(), [](int x) {
        std::cout << x << " ";
    });
    std::cout << std::endl;
    return 0;
}
```

- Syntaxe générale : `[captures](params) -> retour { corps }`
- Permet de définir des fonctions courtes et locales sans nom.

Utilisation des captures dans les lambdas

Les lambdas peuvent capturer des variables de leur environnement.
La capture peut se faire par valeur ou par référence.

```
int main() {
    int a = 5;
    int b = 10;

    // Capture par valeur : copie de a et b dans la lambda
    auto somme = [a, b]() {
        return a + b;
    };
    std::cout << "Somme (par valeur) : " << somme() << std::endl;

    // Capture par référence : la lambda peut modifier a
    auto incremente = [&a]() {
        a++;
    };
    incremente();
    std::cout << "a après incrémentation (par référence) : " << a << std::endl;

    return 0;
}
```

- [a, b] capture par valeur, les variables sont copiées.
- [&a] capture par référence, permettant de modifier la variable originale.
- On peut combiner les captures selon les besoins.

Introduction aux lambdas en C++

Les lambdas sont des fonctions anonymes que l'on peut définir directement sur place. Elles sont particulièrement utiles pour passer des fonctions comme arguments aux algorithmes.

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // Utilisation d'une lambda pour afficher chaque élément du vecteur
    std::for_each(vec.begin(), vec.end(), [](int x) {
        std::cout << x << " ";
    });
    std::cout << std::endl;
    return 0;
}
```

- Syntaxe générale : `[captures](params) -> retour { corps }`
- Permet de définir des fonctions courtes et locales sans nom.

Utilisation des captures dans les lambdas I

Les lambdas peuvent capturer des variables de leur environnement. La capture peut se faire par valeur ou par référence.

```
int main() {
    int a = 5;
    int b = 10;
    // Capture par valeur : copie de a et b dans la lambda
    auto somme = [a, b]() {
        return a + b;
    };
    std::cout << "Somme (par valeur) : " << somme() << std::endl;
    // Capture par référence : la lambda peut modifier a
    auto incremente = [&a]() {
        a++;
    };
    incremente();
    std::cout << "a après incrémentation (par référence) : " << a << std::
         endl;
    return 0;
}
```

- [a, b] capture par valeur, les variables sont copiées.
- [&a] capture par référence, permettant de modifier la variable originale.
- On peut combiner les captures selon les besoins.