Utilisation des itérateurs en C++ et parallélisation

L3 Informatique S6

14 mars 2025

Plan

- Rappel sur la syntaxe des boucles `for` classiques
- Les principaux conteneurs de la bibliothèque standard
- Introduction aux itérateurs en C++
- Parallélisation avec les itérateurs (C++17 et plus)
- Exemples pratiques et perspectives

Rappel : Boucles `for` classiques (1)

- Syntaxe générale : `for (initialisation; condition; incrémentation) { ... }`
- Permet d'itérer sur une séquence de valeurs.
- L'initialisation définit la variable de boucle.
- La condition est vérifiée à chaque itération.
- L'incrémentation permet d'évoluer la variable de boucle.

Rappel : Boucles `for` classiques (2) – Exemple

```
#include <iostream>
using namespace std;

int main() {
    // Affiche les nombres de 0 à9
    for (int i = 0; i < 10; ++i) {
        cout << "i␣=␣" << i << "\n";
    }
    return 0;
}
```

Exemple de bug : Dépassement de tableau

```
#include <iostream>
using namespace std;

int main() {
    int tab[5] = {0, 1, 2, 3, 4};
    // Boucle incorrecte : l'indice i parcourt de 0 à5 inclus,
    // ce qui entraîne un accès hors bornes lorsque i vaut 5.
    for (int i = 0; i <= 5; ++i) {
        cout << "tab[" << i << "] = " << tab[i] << "\n";
    }
    return 0;
}
```

Les conteneurs et les itérateurs permettent d'éviter ce type de bug.

Les principaux conteneurs de la STL (1)

- **Conteneurs séquentiels** :
  - `std::vector` : tableau dynamique.
  - `std::list` : liste doublement chaînée.
  - `std::deque` : file à double entrée.
- **Conteneurs associatifs** :
  - `std::set` et `std::map` : conteneurs triés.

Les principaux conteneurs de la STL (2) – Exemple

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

int main() {
    vector<int> vec = {5, 1, 2, 3, 4, 5};
    set<int> s = {5, 3, 5, 1, 4, 2};

    cout << "Vector:␣";
    for (auto n : vec)
        cout << n << "␣";
    cout << "\nSet:␣";
    for (auto n : s)
        cout << n << "␣";
    // output: Vector: 5 1 2 3 4 5
    // Set: 1 2 3 4 5
    return 0;
}
```

Exemple avancé : std::set

```
#include <iostream>
#include <set>
using namespace std;

int main() {
    set<int> s;
    s.insert(42);
    s.insert(17);
    s.insert(23);
    s.insert(17); // Tentative d'insertion d'un doublon
    s.insert(89);

    cout << "Elements␣dans␣le␣set␣:␣";
    for (int x : s)
        cout << x << "␣";
    cout << "\n";

    cout << "En␣ordre␣inverse␣:␣";
    for (auto it = s.rbegin(); it != s.rend(); ++it)
        cout << *it << "␣";
    cout << "\n";

    return 0;
}
```

Exemple : std::unordered_set

```
#include <iostream>
#include <unordered_set>
using namespace std;

int main() {
    // Création d'un unordered_set avec quelques valeurs
    unordered_set<int> uset = {5, 1, 3, 9, 7};

    cout << "Elements dans l'unordered_set : ";
    // L'ordre d'affichage n'est pas garanti
    for (int x : uset)
        cout << x << " ";
    cout << "\n";

    return 0;
}
```

Exemple : std::list I

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main() {
    list<string> fruits = {"pomme", "banane", "cerise"};

    // Ajout en tête et en queue
    fruits.push_front("orange");
    fruits.push_back("kiwi");

    cout << "Liste␣de␣fruits␣:␣";
    for (auto it = fruits.begin(); it != fruits.end(); ++it)
        cout << *it << "␣";
    cout << "\n";

    // Suppression d'un élément
    fruits.remove("banane");

    cout << "Après␣suppression␣de␣'banane'␣:␣";
    for (const auto &fruit : fruits)
```

Exemple : std::list II

```
            cout << fruit << "␣";
        cout << "\n";

        return 0;

    }
}
```

Exemple : std::map I

```
#include <map>
#include <string>
int main() {
    map<string, int> ages;
    ages["Alice"] = 30;
    ages["Bob"] = 25;
    ages["Charlie"] = 35;

    // Mise àjour de l'âge d'Alice
    ages["Alice"] = 31;

    cout << "Âges␣des␣personnes␣:\n";
    for (const auto &pair : ages)
        cout << pair.first << "␣a␣" << pair.second << "␣ans\n";

    // Recherche d'un élément
    auto it = ages.find("Bob");
    if (it != ages.end())
         cout << "Bob␣a␣été␣trouvé␣avec␣" << it->second << "␣ans.\n";

    return 0;
}
```

Introduction aux itérateurs

- Les itérateurs sont des objets permettant de parcourir des conteneurs.
- Pointeurs généralisés ;
- ou outils de parcours de plus haut niveau (par exemple, fonctionnels).

Exemple simple

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    for(auto it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << "␣";
    }
    // même boucle avec la syntaxe range-based for
    for(int i : vec) {
        cout << i << "␣";
    }
    return 0;
}
```

Itérateurs et algorithmes parallèles (C++17)

- C++17 introduit des politiques d'exécution pour paralléliser les algorithmes.
- Exemples de politiques : `std::execution::par` et `std::execution::par_unseq`.
- Ces politiques permettent d'utiliser les itérateurs dans un contexte parallèle.

Exemple : `std::for_each` parallèle

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    for_each(execution::par, vec.begin(), vec.end(), [](int &n) {
        n *= 2;
    });
    for(auto n : vec)
        cout << n << "␣";
    return 0;
}
```

Utilisations avancées

- Combinaison avec d'autres algorithmes de la STL (ex. `std::transform`).
- Utilisation d'adaptateurs d'itérateurs comme `reverse_iterator` ou `insert_iterator`.
- Les itérateurs parallèles améliorent les performances sur de grands jeux de données.

Exemple : `std::transform` parallèle

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>
using namespace std;

int main() {
    vector<int> src = {1, 2, 3, 4, 5};
    vector<int> dest(src.size());
    transform(execution::par, src.begin(), src.end(), dest.begin(), [](
         int n) {
        return n * n;
    });
    for(auto n : dest)
        cout << n << "␣";
    return 0;
}
```

Avantages et limites

- **Avantages :**
  - Amélioration des performances sur de grands conteneurs.
  - Code concis et expressif.
- **Limites :**
  - Dépendance à l'architecture matérielle (nombre de cœurs).
  - Complexité accrue dans le débogage.

Exemple : Itérateur parallèle avec bug non détecté par le compilateur I

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>
using namespace std;

int main() {
    //vector<int> vec = {1, 2, 3, 4, 5};
    // vecteur contenant les 100 premiers entiers
    vector<int> vec(100);
    std::iota(vec.begin(), vec.end(), 0);
    // affichage
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << "␣";
    }
    int sum = 0;
    // BUG : accès concurrent non synchronisé à'sum'
    // La modification de 'sum' dans une exécution parallèle crée une
          condition de course
    for_each(execution::par, vec.begin(), vec.end(), [&](int n) {
         sum += n; // Race condition non détectée par le compilateur
```

Exemple : Itérateur parallèle avec bug non détecté par le compilateur II

```
    });
    cout << "Somme:␣" << sum << "\n";
    return 0;
}
```

Conclusion et perspectives

- Les itérateurs simplifient le parcours des conteneurs et facilitent l'utilisation des algorithmes.
- Les politiques d'exécution de C++17 offrent de nouvelles possibilités pour la parallélisation.
- Il faut tester et expérimenter pour bien comprendre...