Itérateurs et quelques usages

14 novembre 2018

Swiss Olympiad in Informatics

Itérateurs

Itérateur - exemple

Avec `a` qui est du type `vector<int>` :

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

Itérateur - exemple

Avec `a` qui est du type `vector<int>` :

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

Itérateur - définition

Définition : Un itérateur (*iterator* en anglais) est un objet qui pointe sur un élément d'une collection qui permet de traverser (*iterate* en anglais) cette collection.

Itérateur - définition

Définition : Un itérateur (*iterator* en anglais) est un objet qui pointe sur un élément d'une collection qui permet de traverser (*iterate* en anglais) cette collection.

Traverser la collection est fait à l'aide de :

- L'opérateur d'incrémentation (`++`, *increment operator*) pour passer à l'itérateur qui pointe sur le prochain élément de la collection
- L'opérateur de déréférencement (`*`, *indirection operator*) qui permet d'accéder à l'élément pointé

Itérateur - exemple

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

Itérateur - exemple

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

- `a.begin()` correspond à l'itérateur qui pointe sur `a[0]`

Itérateur - exemple

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

- `a.begin()` correspond à l'itérateur qui pointe sur `a[0]`
- `++it` avance l'itérateur `it` de `a[i]` à `a[i+1]`

Itérateur - exemple

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

- `a.begin()` correspond à l'itérateur qui pointe sur `a[0]`
- `++it` avance l'itérateur `it` de `a[i]` à `a[i+1]`
- `a.end()` correspond à l'itérateur qui pointe à une position après la dernière valeur du vector

Itérateur - exemple

```
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it << "\n";
}
```

```
for (unsigned int i = 0; i < a.size(); ++i) {
    cout << a[i] << "\n";
}
```

- `a.begin()` correspond à l'itérateur qui pointe sur `a[0]`
- `++it` avance l'itérateur `it` de `a[i]` à `a[i+1]`
- `a.end()` correspond à l'itérateur qui pointe à une position après la dernière valeur du vector
- `*it` permet d'accéder l'élément sur lequel l'itérateur pointe

Itérateur pour un vector

Pour un vector, un itérateur offre des fonctionalités similaires aux indices.

Itérateur pour un vector

Pour un vector, un itérateur offre des fonctionalités similaires aux indices.

```
vector<int>::iterator it = a.begin() + 5;
```

Itérateur pour un vector

Pour un vector, un itérateur offre des fonctionalités similaires aux indices.

```
vector<int>::iterator it = a.begin() + 5;
```

`it` pointe sur `a[5]`

Itérateur pour un vector

Pour un vector, un itérateur offre des fonctionalités similaires aux indices.

```
vector<int>::iterator it = a.begin() + 5;
```

`it` pointe sur `a[5]`

```
--it;
```

Itérateur pour un vector

Pour un vector, un itérateur offre des fonctionalités similaires aux indices.

```
vector<int>::iterator it = a.begin() + 5;
```

`it` pointe sur `a[5]`

```
--it;
```

`it` pointe sur `a[4]`

Attention !

Avec `a` qui est de type `vector<vector<int> >` :

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it.size() << "\n";
}
```

Attention !

Avec `a` qui est de type `vector<vector<int> >` :

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it.size() << "\n";
}
```

`*it.size()` est interprété comme `*(it.size())`, cependant, on cherche à obtenir `(*it).size()`

Attention !

Avec `a` qui est de type `vector<vector<int> >` :

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it.size() << "\n";
}
```

`*it.size()` est interprété comme `*(it.size())`, cependant, on cherche à obtenir `(*it).size()`

Comme pour l'addition et la multiplication, il y a des priorités !

Attention !

Avec `a` qui est de type `vector<vector<int> >` :

```
for (vector<vector<int> >::iterator it = a.begin(); it != a.end(); ++it) {
    cout << *it.size() << "\n";
}
```

`*it.size()` est interprété comme `*(it.size())`, cependant, on cherche à obtenir `(*it).size()`

Comme pour l'addition et la multiplication, il y a des priorités !

`(*a).x` est équivalent à `a->x`

Trouver l'erreur

Avec `a` qui est de type `vector<vector<int> >` :

```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // on désire avoir 6, 6, 6, 6, 6, 6
```

Quelle est l'erreur ?

Trouver l'erreur

Avec `a` qui est de type `vector<vector<int> >` :

```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // on désire avoir 6, 6, 6, 6, 6, 6
```

Quelle est l'erreur ? L'itérateur `it` pointe seulement sur le premier 5 dans le vector. Si on désire changer la valeur pointée, il faut déréférencer l'itérateur, donc faire `*it = 4;`.

Trouver l'erreur

Avec `a` qui est de type `vector<vector<int> >` :

```
vector<int> a{1, 4, 5, 5, 2, 5};
for (vector<int>::iterator it = a.begin(); it != a.end(); ++it) {
    it = 6;
} // on désire avoir 6, 6, 6, 6, 6, 6
```

Quelle est l'erreur ? L'itérateur `it` pointe seulement sur le premier 5 dans le vector. Si on désire changer la valeur pointée, il faut déréférencer l'itérateur, donc faire `*it = 4;`. `ita = itb;` change la valeur de l'itérateur `ita` à l'itérateur `itb`

À quoi servent les itérateurs ?

- Pour l'instant, ils font ce que les indices font aussi
- Une syntaxe pas très élégante

… Mais :

À quoi servent les itérateurs ?

- Pour l'instant, ils font ce que les indices font aussi
- Une syntaxe pas très élégante

… Mais :

- Permettent de spécifier un ensemble de valeurs dans la collection

À quoi servent les itérateurs ?

- Pour l'instant, ils font ce que les indices font aussi
- Une syntaxe pas très élégante

… Mais :

- Permettent de spécifier un ensemble de valeurs dans la collection
- La notation `a[i]` ne fait pas de sens dans certains types de collections

Quelques usages

Quels usages ?

À quoi peut servir de pouvoir donner un ensemble de valeurs d'une collection en paramètre ?

Trier un vector

La fonction **`sort`** trie les élements d'un vector en ordre croissant :

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // a devient 1, 2, 4, 5, 5, 5
```

Trier un vector

La fonction `sort` trie les élements d'un vector en ordre croissant :

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // a devient 1, 2, 4, 5, 5, 5
```

De manière générale,

```
sort(x, y);
```

trie les éléments entre l'élément pointé par **x** (inclus) et l'élément pointé par **y** (exclu).

Trier un vector

La fonction `sort` trie les élements d'un vector en ordre croissant :

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // a devient 1, 2, 4, 5, 5, 5
```

De manière générale,

```
sort(x, y);
```

trie les éléments entre l'élément pointé par **x** (inclus) et l'élément pointé par **y** (exclu).
Comment trier tous les éléments d'un vector sauf le premier et les deux derniers ?

Trier un vector

La fonction `sort` trie les élements d'un vector en ordre croissant :

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // a devient 1, 2, 4, 5, 5, 5
```

De manière générale,

```
sort(x, y);
```

trie les éléments entre l'élément pointé par **x** (inclus) et l'élément pointé par **y** (exclu).
Comment trier tous les éléments d'un vector sauf le premier et les deux derniers ?

```
vector<int> a{1, 4, 2, 5, 2, 5};
sort(a.begin()+1, a.end()-2); // a devient 1, 2, 4, 5, 2, 5
```

Trouver l'élément minimal

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    cout << "n'a pas de minimum\n"; // la collection est vide
} else {
    cout << "minimum : " << *it << "\n";
}
```

Trouver l'élément minimal

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = min_element(a.begin(), a.end());
if (it == a.end()) {
    cout << "n'a pas de minimum\n"; // la collection est vide
} else {
    cout << "minimum : " << *it << "\n";
}
```

Attention à bien gérer le cas où la collection est vide !

Compter un élément

```
vector<int> a{1, 4, 5, 5, 2, 5};
cout << "il␣y␣a␣" << count(a.begin(), a.end(), 5) << "␣fois␣5␣dans␣a\n";
// affiche "il y a 3 fois 5 dans a\n"
```

Trouver la position d'un élément

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    cout << "l'element␣n'existe␣pas\n";
} else {
    cout << "l'element␣" << *it << "␣existe␣dans␣la␣collection\n";
}
```

Trouver la position d'un élément

```
vector<int> a{1, 4, 5, 5, 2, 5};
vector<int>::iterator it = find(a.begin(), a.end(), 5);
if (it == a.end()) {
    cout << "l'element n'existe pas\n";
} else {
    cout << "l'element " << *it << " existe dans la collection\n";
}
```

Attention à bien penser au cas où l'élément ne se trouve pas dans la collection !

Remplir une collection

```
vector<int> a{1, 4, 5, 5, 2, 5};
fill(a.begin(), a.end, 0); // a : 0, 0, 0, 0, 0, 0
```

Remplir une collection

```
vector<int> a{1, 4, 5, 5, 2, 5};
fill(a.begin(), a.end, 0); // a : 0, 0, 0, 0, 0, 0
```

```
vector<int> a{1, 4, 5, 5, 2, 5};
iota(a.begin(), a.end(), -5); // a : -5, -4, -3, -2, -1, 0
```

Enlever des éléments

```
vector<int> a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
a.erase(a.end() - 2); // a : 0, 1, 2, 3, 4, 5, 6, 7, 9
a.erase(a.begin() + 3, a.begin() + 5); // a : 0, 1, 2, 5, 6, 7, 9
a.erase(a.begin(), a.end()); // a est vide
```

Enlever des éléments

```
vector<int> a{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
a.erase(a.end() - 2); // a : 0, 1, 2, 3, 4, 5, 6, 7, 9
a.erase(a.begin() + 3, a.begin() + 5); // a : 0, 1, 2, 5, 6, 7, 9
a.erase(a.begin(), a.end()); // a est vide
```

Pourquoi la syntaxe est différente des autres fonctions ?

Obtenir tous les éléments uniques d'un vector

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // trie le vector
// a : 1, 2, 4, 5, 5, 5
a.erase( unique(a.begin(), a.end()), a.end());
// a : 1, 2, 4, 5
```

Que fait `unique(a.begin(), a.end())` ?

Obtenir tous les éléments uniques d'un vector

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // trie le vector
// a : 1, 2, 4, 5, 5, 5
a.erase( unique(a.begin(), a.end()), a.end());
// a : 1, 2, 4, 5
```

Que fait `unique(a.begin(), a.end())` ?

- Met toutes les copies à la fin du vector

Obtenir tous les éléments uniques d'un vector

```
vector<int> a{1, 4, 5, 5, 2, 5};
sort(a.begin(), a.end()); // trie le vector
// a : 1, 2, 4, 5, 5, 5
a.erase( unique(a.begin(), a.end()), a.end());
// a : 1, 2, 4, 5
```

Que fait `unique(a.begin(), a.end())` ?

- Met toutes les copies à la fin du vector
- Retourne l'itérateur qui pointe sur la première copie

Des fonctions comme argument

Motivation

Pourquoi réinventer la roue si on désire trier un vector en ordre inverse ?

Trier en ordre décroissant

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

using namespace std;

bool comparateur(int lhs, int rhs) {
    return (lhs > rhs);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    sort(a.begin(), a.end(), comparateur);
    // a : 8 8 7 6 5 4 3 2 2
}
```

Trouver si un élément satisfait une contrainte

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

using namespace std;

bool impair(int n) {
    return (n % 2 == 1);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = find_if(a.begin(), a.end(), impair);
    if (it != a.end()) {
        cout << "trouve " << *it << "\n";
    } else {
        cout << "pas trouve\n";
    }
}
```

Enlever tous les éléments qui satisfont une contrainte

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

using namespace std;

bool impair(int n) {
    return (n % 2 == 1);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = remove_if(a.begin(), a.end(), impair);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- Déplace tous les éléments pairs au début du vector (aucune garantie pour la fin !)

Enlever tous les éléments qui satisfont une contrainte

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

using namespace std;

bool impair(int n) {
    return (n % 2 == 1);
}

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    vector<int>::iterator it = remove_if(a.begin(), a.end(), impair);
    // a : 2 6 8 4 2 8 2 7 8
    a.erase(it, a.end());
    // a : 2 6 8 4 2 8
}
```

- Déplace tous les éléments pairs au début du vector (aucune garantie pour la fin !)
- Retourne un itérateur qui pointe après les éléments pairs

Lambdas

La fonction `impair` n'est que utilisée une seule fois, pourquoi lui donner un nom ?

Lambdas

La fonction **`impair`** n'est que utilisée une seule fois, pourquoi lui donner un nom ?

Le code précédent s'écrit aussi

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

using namespace std;

int main() {
    vector<int> a{2, 6, 3, 5, 8, 4, 2, 7, 8};
    a.erase(remove_if(a.begin(), a.end(),
         [](int i){ return (i % 2) == 1; }
    ));
    // a : 2 6 8 4 2 8
}
```

Questions

Des questions ?