1 Itérateurs

1.1 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

1.2 Exemple

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

est équivalent à :

```
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

Attention à ces erreurs :

- Ne pas confondre `it` et `*it`
- `*it.x` est interprété comme `*(it.x)`, et pas comme `(*it).x`. `it->x` est équivalent à `(*it).x`

2 Quelques usages

2.1 Trier un vector, inclure `algorithm`

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

La borne gauche est incluse, la borne droite exclue.

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

2.2 Trouver le minimum, inclure `algorithm`

```
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";
}
```

2.3 Compter un élément, inclure `algorithm`

```
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"
```

2.4 Trouver un élément, inclure `algorithm`

```
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";
}
```

2.5 Remplir un vector, inclure `algorithm`

```
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
```

2.6 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
```

2.7 Obtenir tous les éléments uniques, inclure `algorithm`

```
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
```

`unique` met toutes les copies à la fin et retourne l'itérateur sur la première copie

3 Des fonctions comme arguments, exemples

3.1 Trier en ordre inverse

```
#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
}
```

3.2 Trouver un élément impair

```
#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";
    }
}
```

3.3 Enlever tous les éléments qui satisfont une contrainte

```
#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
}
```

`remove_if` :

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

4 Lambdas

Les lambdas permettent de créer une fonction sans lui donner un nom. Cela permet de créer la fonction directement lors de l'appel d'une fonction par exemple.