

Chapitre 17

La bibliothèque de modèles standard

(STL: Standard Template Library)

1. Problématique

Une application est constituée de:

- Noms de variables ayant un type donné T: ex. `int`, `double`, `char`, etc.,
- Conteneurs gérant les données décrites dans cette application: ex. `vecteur`, `liste`, etc.,
- Et, finalement, des algorithmes appliqués sur ces données: ex. `sort`, `find`, etc.,

Supposons que l'on a `i` types, `j` conteneurs et `k` algorithmes, pour concevoir l'application, on devrait écrire:

`i * j * k` versions de code!!!

Avec l'apport des fonctions (ou classes) génériques (template), `i = 1`, car un seul type T est traité, ce nombre est réduit à:

`k * j`

Avec des algorithmes capables de travailler sur n'importe quel type de conteneurs: ex. `sort` pour vecteur, liste etc. ce nombre tombe à :

`k + j`

Le but des STL est d'uniformiser le traitement, afin de réduire la prolifération de versions de code.

2. Définitions

STL est un ensemble de classes qui, tout en collaborant, permettent de réaliser dans une catégorie de logiciels des conceptions réutilisables.

Les STL :

- * utilisent les templates,
- * n'utilisent pas l'héritage et les fonctions virtuelles pour des raisons d'efficacité.

3. Composants

Les STL contiennent 6 catégories principales:

3.1. Algorithmes

Ils sont utilisés pour traiter les éléments d'un ensemble de données. Ils définissent une procédure informatique, ex: *find*, *sort*, *copy*, etc.

3.2. Conteneurs

Ils sont utilisés pour gérer une collection d'objets d'un type donné. Ils gèrent donc les structures de données ou bien ce sont des objets pour stocker d'autres objets. Les conteneurs peuvent être implantés comme des tableaux, listes chaînées, ou bien comme une clé spéciale pour chacun des éléments.

3.3. Itérateurs

Ils fournissent aux algorithmes un moyen pour parcourir un conteneur (généralisation de la notion de pointeurs).

3.4. Fonctions objets

Des classes qui redéfinissent l'opérateur d'appel de fonction (`()`).

3.5. Adaptateurs

C'est l'encapsulation d'un autre composant pour fournir une autre interface. Par exemple: d'un tableau peut dériver une pile mais avec une interface réduite.

3.6. Allocateurs

Ils encapsulent le modèle de gestion de la mémoire. C'est une sorte de surdéfinition des opérateurs *new* & *delete*.

4. Conteneurs

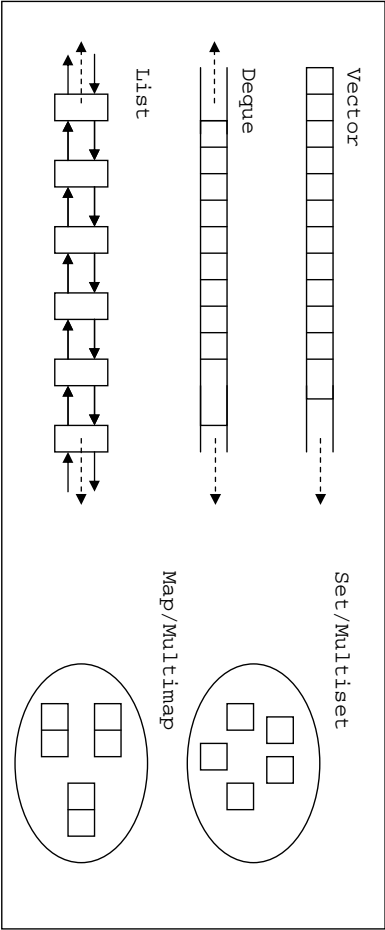
Trois catégories de conteneurs:

Séquentiels: l'accès est linéaire, c'est l'ordre qui compte.

Associatifs: l'accès par clés.

Adaptateurs: l'accès est réglementé.

4.1. Conteneurs séquentiels



* `vector`

Un vecteur gère ses éléments dans un tableau dynamique. Il permet d'accéder directement à un élément à partir d'un index.

Ajouter ou enlever des éléments à partir de la fin est une opération très rapide à exécuter.

Par contre, insérer un élément au début ou bien au milieu du vecteur, prend un certain temps, car les éléments doivent être déplacés un par un pour créer une place au nouveau arrivant.

Il faut inclure:

```
#include <vector>
```

```
using namespace std;
```

À titre d'exemple:

```
int main() {  
  
    vector<int> v1; // conteneur vide.  
    vector<double> v2(6); // conteneur de 6 éléments égaux à zéro.  
    vector<int> v3(10, 999); // 10 éléments égaux à 999.  
  
    int tab[4] = { 1, 2, 3, 4};  
  
    // Nouveau conteneur vector par recopie des éléments du tableau tab.  
    vector<int> v4(tab, tab+4); // 1 2 3 4
```

```
// La taille du conteneur v4.
cout << v4.size() << endl; // 4

// Échanger les éléments: v4 devient v3, et vice versa.
v4.swap(v3); // v4: 999 999 999 999 999 999 999 999
// v3: 1 2 3 4

// Destructeur de v4 est appelé. Maintenant size de v4 = 0.
v4.clear ();

cout << v4.size() << endl; // 0

// Conversion illégale: v2 (double) alors que v3 (int) .
// v2 = v3;

// assign disponible uniquement dans les versions récentes de gcc! Le vecteur v3 a une
// nouvelle taille = 2, et ses valeurs sont initialisées à 5.
// v3.assign(2,5);

// On insère au début, deux éléments identiques! Opération à éviter, car prend "trop" de temps!
v2.insert(v2.begin(),2,0.89); // 0.89 0.89 0 0 0 0 0

// On insère à partir de la 3e position du début, le contenu de du tableau tab des éléments de
// la position 1 à 3, donc que 2 éléments. Opération à éviter, car prend "trop" de temps!
v2.insert(v2.begin()+2,tab+1,tab+3); // 0.89 0.89 2 3 0 0 0 0 0

// On insère à partir de la fin, opération idéale à faire!
v2.insert(v2.end(),2,28.99); // 0.89 0.89 2 3 0 0 0 0 28.99 28.99
```

```
// Une autre façon pour insérer un élément, à la manière des piles!
v2.push_back(55.23); // 0.89 0.89 2 3 0 0 0 0 28.99 28.99 55.23

// On retire le 2e élément du vecteur, le premier étant indexé par 0.
v2.erase(v2.begin()+1); // 0.89 2 3 0 0 0 0 28.99 28.99 55.23

// On retire le dernier élément dans v4, un pop à la pile ...
v2.pop_back(); // 0.89 2 3 0 0 0 0 28.99 28.99

return 0;

}
```

*** deque (DoubleEndedQueue)**

C'est un tableau dynamique qui peut grossir dans les deux directions, une file bilatérale.

Insérer des éléments au début ou à la fin sont deux opérations rapides à exécuter, par contre insérer un élément au milieu va prendre un certain temps, car il faudra déplacer les éléments.

Il faut inclure:

```
#include <deque>

using namespace std;
```

À titre d'exemple:

```
int main() {  
    // conteneur deque contenant 4 éléments identiques, valant 8.  
    deque<int> v1(4, 8);  
    return 0;  
}
```

Les fonctions disponibles pour la classe `vector` se retrouvent en majeure partie dans la classe `deque`. Par contre, la classe `deque` est enrichie de deux fonctions, `push_front()` et `pop_front()`, vu que c'est une file bilatérale. Ces deux fonctions se comportent de la même manière que `push_back()` et `pop_back()` expliquées dans l'exemple du paragraphe 4.1.

* list

C'est une liste doublement chaînée.

Chaque élément de la liste a son propre segment de mémoire, et pointe son prédécesseur et son successeur. Les listes ne permettent pas un accès aléatoire, c.-à-d. à un index donné, comme le font les conteneurs `vector` et `deque`. Pour accéder au N^{ième} élément de la liste, il faut passer par les N-1 prédécesseurs. L'accès à un élément donné prend un temps linéaire nettement supérieur que si cet élément était dans un `vector` ou `deque`.

L'avantage des listes est que l'insertion ou le retrait d'un élément se fait rapidement. Il suffit de modifier les adresses des pointeurs des prédécesseurs et successeurs.

Il faut inclure:

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

À titre d'exemple:

```
int main() {  
    // Conteneur list contenant 4 éléments identiques du type int, et = 8.  
    list<int> l1(4, 8);  
    l1.push_front(20); // 20 8 8 8 8  
    l1.push_back(56); // 20 8 8 8 8 56  
    l1.reverse(); // 56 8 8 8 8 20  
    l1.erase(l1.begin()); // 8 8 8 8 20  
    return 0;  
}
```

4.2. Conteneurs associatifs

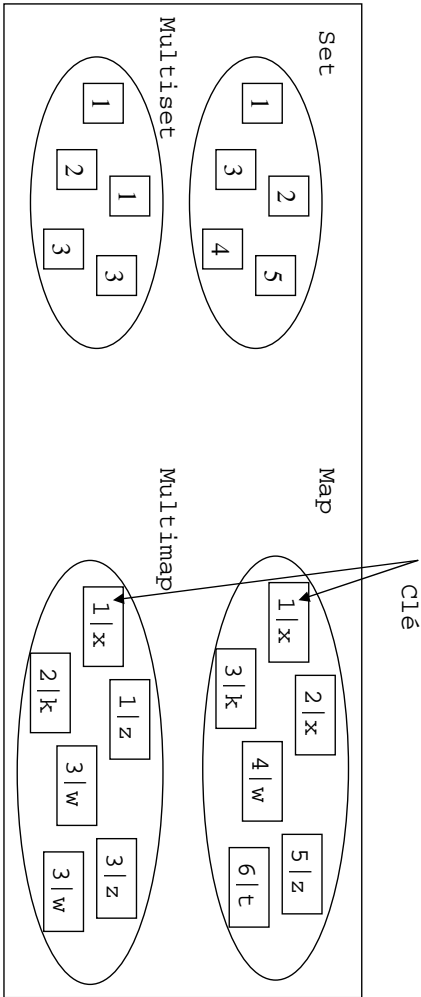
Les éléments sont triés suivant un certain critère. Ce critère est une sorte de fonction qui compare suivant la valeur (le cas de `set/multiset`) ou bien la clé associée à la valeur (le cas de `map/multimap`). Par défaut, cette fonction

utilise l'opérateur de comparaison <, cependant, une autre fonction de comparaison, se basant sur d'autres critères peut-être définie pour réaliser cette opération.

Nous avons donc deux catégories de conteneurs:

set/multiset: ne contiennent que la valeur, ex: {jobs, gates, ellison}

map/multimap: une paire contenant une clé et une valeur associée à cette clé, ex: {(jobs,apple),(gates,microsoft),(ellison,oracle)}



La différence entre set et multiset est que la valeur peut être dupliquée. La différence entre map et multimap est que la clé peut être dupliquée.

*** set**

il faut inclure:

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

À titre d'exemple:

```
int main() {
    // Définit un conteneur set avec ordre ascendant de ses éléments (configuration par défaut).
    typedef set<int> IntSet;

    // Définit un type de conteneur avec ordre descendant de ses éléments. Ici on a utilisé un autre
    // critère de tri, on pouvait définir aussi notre propre fonction et l'inclure à la place
    // de greater<int>.
    typedef set<int,greater<int>> IntGreatSet;

    IntSet coll1;
    // On insert des éléments dans coll1.
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
```

```
coll1.insert(2);
// Pas grave si on duplique, la valeur ne sera pas prise en compte.
coll1.insert(5);
IntSet::iterator pos;

// Affichage des éléments de coll1 // 1 2 3 4 5 6
for (pos=coll1.begin(); pos != coll1.end(); ++pos) {
    cout << *pos << ' ';
}
cout << endl;

// Création par recopie de coll2 à partir de coll1, avec tri descendant.
IntGreatSet coll2(coll1.begin(), coll1.end());

// Copie vers le flux de sortie le contenu de coll2 // 6 5 4 3 2 1
copy (coll2.begin(), coll2.end(), ostream_iterator<int>(cout, " "));
cout << endl;

// Enlève tous les éléments ayant la valeur 5.
int num;
num = coll2.erase(5); // 1
cout << "élément(s) retiré(s) du conteneur: " << num << endl;
// Efface tous les éléments jusqu'à l'élément ayant une valeur = 3.
coll2.erase (coll2.begin(), coll2.find(3)); // 3 2 1

// Copie vers le flux de sortie le contenu de coll2 // 3 2 1
copy (coll2.begin(), coll2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
return 0;
}
```

```
Sortie:
1 2 3 4 5 6
6 5 4 3 2 1
élément(s) retiré(s) du conteneur: 1
3 2 1
```

On peut tester aussi si un élément se trouve déjà dans le conteneur. On utilise pour cela la notion de `pair` définie dans le fichier `<utility>` ... c'est une autre histoire ...

*** multiset**

Ce qui va différer avec l'exemple précédent c'est la déclaration de `multiset` au lieu de `set` et la sortie du programme.

```
il faut inclure:

#include <set>
using namespace std;

À titre d'exemple:
```

Dans l'exemple utilisant les `set`, vous remplacez les lignes suivantes:

```
typedef set<inc> IntSet;
typedef set<int, greater<int>> > IntGreatSet;
```

```
par
    typedef multiset<int> IntSet;
    typedef multiset<int,greater<int> > IntGreatSet;

Sortie:
1 2 3 4 5 5 6
6 5 5 4 3 2 1
élément(s) retiré(s) du conteneur: 2
3 2 1

* map

Il faut inclure:

#include <map>
using namespace std;

À titre d'exemple:
```

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {

    /*
    Crée un map / tableau associatif
    La clé est une string, représentant le nom du compositeur
```

```
*/
    La valeur une string, représentant une de ses œuvres

    typedef map<string,string> StringStringMap;
    StringStringMap compositeurs;

    // Insérer quelques éléments.

    compositeurs["Vivaldi"] = "Les 4 saisons";
    compositeurs["Mozart"] = "La flûte enchantée";
    compositeurs["Beethoven"] = "9";
    compositeurs["Bach"] = "La passion selon saint Matthieu"; // J.S.B
    compositeurs["Puccini"] = "Madama Butterfly";

    StringStringMap::iterator pos;

    // On peut accéder à chaque composante d'une paire clé/valeur par:
    // first: pour avoir la clé; second: pour avoir la valeur associée.
    for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
        cout << "nom du compositeur: " << pos->first << "\t"
            << "un échantillon: " << pos->second << endl;
    }
    cout << endl;

    // On copie directement les éléments de Bach vers "lui-même" (JS: Jean Sébastien)
    compositeurs["Bach_js"] = compositeurs["Bach"];
    // On peut enlever un élément ...
    compositeurs.erase("Bach");
```



```
for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
    cout << "nom du compositeur: " << pos->first << "\t"
    << "un échantillon: " << pos->second << endl;
}
// On va chercher si un compositeur est dans la liste ...
pos = compositeurs.find("Berlioz");

// Si on arrive à la fin sans l'avoir, cela veut dire qu'il n'a pas été inclus dans la liste.
if (pos==compositeurs.end()){
    cout << "Berlioz n'est pas dans la liste\n";
} else {
    cout << "nom du compositeur: Berlioz\tun échantillon: " \
    << compositeurs["Berlioz"] << endl;
}
return 0;
}
```

Sortie:

```
nom du compositeur: Bach          un échantillon: La passion selon saint Matthieu
nom du compositeur: Beethoven    un échantillon: 9
nom du compositeur: Mozart       un échantillon: La flûte enchantée
nom du compositeur: Puccini      un échantillon: Madama Butterfly
nom du compositeur: Vivaldi      un échantillon: Les 4 saisons

nom du compositeur: Bach_js      un échantillon: La passion selon saint Matthieu
nom du compositeur: Beethoven    un échantillon: 9
nom du compositeur: Mozart       un échantillon: La flûte enchantée
nom du compositeur: Puccini      un échantillon: Madama Butterfly
nom du compositeur: Vivaldi      un échantillon: Les 4 saisons

Berlioz n'est pas dans la liste
```

* multimap

Dans le cas des multimap, la clé peut-être dupliquée. Ainsi par exemple, pour la clé "mozart", avec les multimap, on peut associer à cette clé d'autres noms de compositions.

Il y a plusieurs manières pour insérer des éléments. Nous avons montré dans l'exemple précédent une insertion du style tableau associatif:

```
compositeurs["Vivaldi"] = "Les 4 saisons";
```

nous pouvions aussi insérer des éléments comme suit:

- value_type (utilisée pour éviter les conversions implicites)

compositeurs.insert(StringStringMap::value_type("Vivaldi", "Les 4 saisons"));

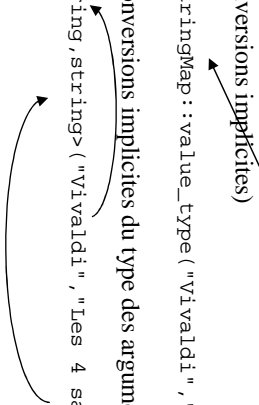
- pair<> (il faudrait faire attention aux conversions implicites du type des arguments)

compositeurs.insert(pair<string, string>("Vivaldi", "Les 4 saisons"));

- make_pair() la plus simple à écrire, cependant faire attention aux conversions de type.

compositeurs.insert(make_pair("Vivaldi", "Les 4 saisons"));

map<string, string>



Il faut inclure:

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

À titre d'exemple:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
```

```
    /*
    Crée un multimap
    La clé est une string, représentant le nom du compositeur.
    La valeur est une string, représentant une de ses œuvres.
    */
```

```
typedef multimap<string, string> StringStringMap;
StringStringMap compositeurs;
```

// Insérer quelques éléments.

```
compositeurs.insert(make_pair("Vivaldi", "Les 4 saisons"));
compositeurs.insert(make_pair("Mozart", "La flûte enchantée"));
compositeurs.insert(make_pair("Beethoven", "9"));
compositeurs.insert(make_pair("Bach", "La passion selon saint Matthieu"));
compositeurs.insert(make_pair("Puccini", "Madama Butterfly"));
```

```
compositeurs.insert(make_pair("Mozart", "Don Giovanni"));
compositeurs.insert(make_pair("Beethoven", "Pour Elise"));
compositeurs.insert(make_pair("Puccini", "La Bohème"));

// On déclare un itérateur.
StringStringMap::iterator pos;
```

// On peut accéder à chaque composante d'une paire, clé/valeur par:
// first: pour avoir la clé ; second: pour avoir la valeur associée.

```
for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
    cout << "nom du compositeur: " << pos->first << "\t" \
        << "un échantillon: " << pos->second << endl;
}
```

// Afficher toutes les valeurs associées à la clé Beethoven

```
string composer("Beethoven");
cout << composer << ": " << endl;

// lower_bound: retourne la première position où un élément avec la clé composer
// doit être inséré.
// upper_bound: retourne la dernière position où un élément avec la clé composer
// doit être inséré.
```

```
for (pos = compositeurs.lower_bound(composer);
    pos != compositeurs.upper_bound(composer); ++pos) {
    cout << "\t" << pos->second << endl;
}
```

```
// Afficher toutes les clés associées à la composition no 9
composer = ("9");
cout << composer << " : " << endl;

// On cherche dans toutes les valeurs de la position begin à end toutes les clés associées
// à cette valeur.
for (pos = compositeurs.begin(); pos != compositeurs.end(); ++pos) {
    if (pos->second == composer) {
        cout << "\t" << pos->first << endl;
    }
}
return 0;
}

Sortie:
nom du compositeur: Bach          un échantillon: La passion selon saint Matthieu
nom du compositeur: Beethoven    un échantillon: 9
nom du compositeur: Beethoven    un échantillon: Pour Elise
nom du compositeur: Mozart       un échantillon: La flûte enchantée
nom du compositeur: Mozart       un échantillon: Don Giovanni
nom du compositeur: Puccini       un échantillon: Madama Butterfly
nom du compositeur: Puccini       un échantillon: La Bohème
nom du compositeur: Vivaldi       un échantillon: Les 4 saisons

Beethoven:
9
    Pour Elise
Beethoven
```

4.3. Adaptateurs de conteneurs séquentiels

Ces conteneurs adaptent les conteneurs séquentiels afin de remplir des missions précises. Parmi ces adaptateurs, nous citerons: `stack` (pile), `queues` (file) et les `priority queues` (les files prioritaires). Nous allons donner un exemple uniquement dans le cas d'une pile.

La classe `stack` contient une pile du type `LIFO` (LastInFirstOut: dernier arrivé, premier servi).

```
push() insère un élément dans la pile,
pop() retire un élément de la pile,
top() retourne le prochain élément dans la pile.
```

Pour utiliser cette classe, il faut inclure: `#include <stack>`

À titre d'exemple:

```
int main() {
    // Une pile contenant des entiers.
    stack<int> st;

    // On insère 3 éléments dans la pile.
    st.push(1);
    st.push(2);
    st.push(3);
}
```

```
// On affiche l'élément se trouvant au sommet de la pile:
cout << st.top() << ' ' ; // 3

// On retire l'élément se trouvant au sommet de la pile.
st.pop(); // 3

// On affiche l'élément se trouvant au sommet de la pile.
cout << st.top() << ' ' ; // 2

// On retire l'élément se trouvant au sommet de la pile.
st.pop(); // 2

// On insère au sommet de la pile la valeur 77, on écrase ainsi la valeur 1.
st.top() = 77;
// On insère deux éléments dans la pile.
st.push(4);
st.push(5);
// On retire l'élément se trouvant au sommet de la pile: 5.
st.pop(); // 5

// Tant que la pile n'est pas vide, on affiche son contenu.
while (!st.empty()) {
    cout << st.top() << ' ' ;
    st.pop();
}
cout << endl;
return 0;
}
```

Sortie: 3 2 4 77

On peut aussi créer une pile à partir d'un conteneur, par exemple:

```
vector<int> v;

stack<int,v> st;
```

On crée par recopie (élément par élément) la pile st à partir du conteneur vecteur v.

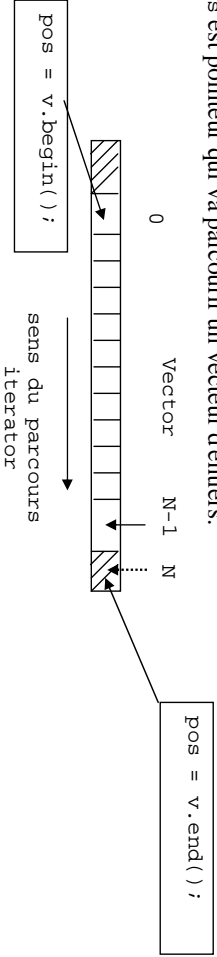
5. Itérateurs

Les conteneurs ont des fonctions membres begin() et end() qui retournent un itérateur. Un itérateur est une sorte de pointeur permettant de parcourir un conteneur.

```
vector<int> v; // taille N

vector<int>::iterator pos;
```

pos est pointeur qui va parcourir un vecteur d'entiers.

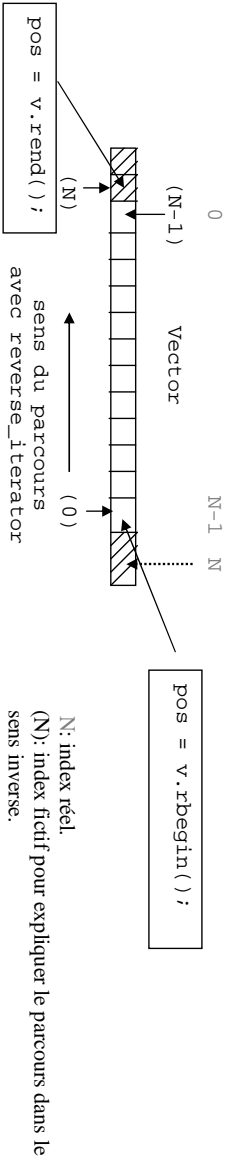


On peut parcourir aussi un conteneur à l'envers:

```
vector<int>::reverse_iterator inverse;
```

Il est associé dans ce cas. Les fonctions permettant d'obtenir les positions dans le cas d'un parcours à l'envers,

```
rbegin() et rend()
```



*** Particularités générales des itérateurs**

Si `it` est un itérateur, on peut toujours faire:

- `it++` pour pointer l'élément suivant,
- déréférencer un itérateur, `(*it)` est l'élément pointé,
- comparer les itérateurs par `== (ou) !=`

Certains conteneurs permettent aussi à des itérateurs de faire:

- `it--` pour pointer l'élément précédent,
- `it+n` accès direct
- `>` `<` des tests de comparaison
- si `ita` et `itb` sont deux itérateurs sur le même conteneur, et on peut atteindre "`itb` de `ita`" alors l'expression `ita, itb` désigne l'intervalle `[ita, itb)`.

6. Algorithmes

Les algorithmes peuvent être classés en fonction de la tâche à réaliser. On distingue principalement les 3 divisions suivantes:

Algorithmes non mutants: ne modifient pas les données (ordre ou valeur), par exemple: `find`, `find_if`, `count`, etc.

Algorithmes mutants: modifient les données, par exemple `reverse`, `swap`, etc.

Algorithmes de tris: `sort`, `heap`, etc.

```
Il faut inclure:
#include <algorithm>
(ou bien: <algo.h>)

using namespace std;
```

À titre d'exemple:

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

using namespace std;

// Pour afficher les éléments.
void dump(int i){ cout << i << endl;}

// Pour détecter les nombres impairs.
bool odd(int i) {return i%2 != 0;}

// Pour comparer deux éléments.
bool comp(const int& i1, const int& i2) { return i1>i2;}

int main() {

    // On initialise le générateur pseudo-aléatoire, pour éviter que rand() ne retourne
    // les mêmes valeurs après le même appel au programme.
    srand(time(NULL));
```

```
// Un vecteur de 10 éléments = 0
vector<int> v(10);

cout << "-----\n";

// On génère 10 valeurs aléatoires et on les stocke dans v.
generate(v.begin(), v.end(), rand);

// On affiche le contenu du vecteur
for_each(v.begin(), v.end(), dump);

cout << "-----\n";

// On remplace tous les éléments impairs par 0.
replace_if(v.begin(), v.end(), odd, 0);

// On trie les éléments par ordre décroissant.
sort(v.begin(), v.end(), comp);

// On affiche le contenu du vecteur.
for_each(v.begin(), v.end(), dump);

cout << "-----\n";

return 0;

}
```

Sortie:

6841991

```
1496271338
1410782963
226586264
1303392378
225013071
1734294671
327806992
138900820
207610059
-----
1496271338
1303392378
327806992
226586264
138900820
0
0
0
0
0
-----
```

7. Fonctions objets

Une fonction objet est une instance d'une classe qui définit l'opérateur parenthèse ().

Chaque fois qu'une fonction objet est utilisée comme une fonction parenthèse, c'est son opérateur () qui est appelé.

Il faut inclure: **#include <functional>**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
class PlusGrandQue:public unary_function <int , bool > {
public:
    // par défaut la variable valeur = 5
    PlusGrandQue(int x=5):valeur(x) {}
    bool operator()(int val) const {return val > valeur;}
};

int main() {
    int tab[10] = {1,2,2,4,5,8,7,9,4,10};
    vector<int> v(tab,tab+10);
    vector<int>::iterator premier;
    premier = find_if(v.begin(),v.end(),PlusGrandQue());
    cout << *premier << endl; // 8 est la 1ère valeur > 5 (valeur par défaut)
    // not1 : l'inverse du critère.
    premier = find_if(v.begin(),v.end(),not1(PlusGrandQue()));
    cout << *premier << endl; // 1 est la première valeur <= 5
    return 0;
}
```

Sortie: 8
1