LANGAGE C++ STL

Définitions

STL = Standard Template Library

Il s’agit d’une bibliothèque générique qui fournit des solutions pour gérer un ensemble de données en utilisant des algorithmes efficaces

Du point de vue du programmeur, la STL fournit un groupe de classes répondant à divers besoins.

Tous les composants de la STL sont des templates.

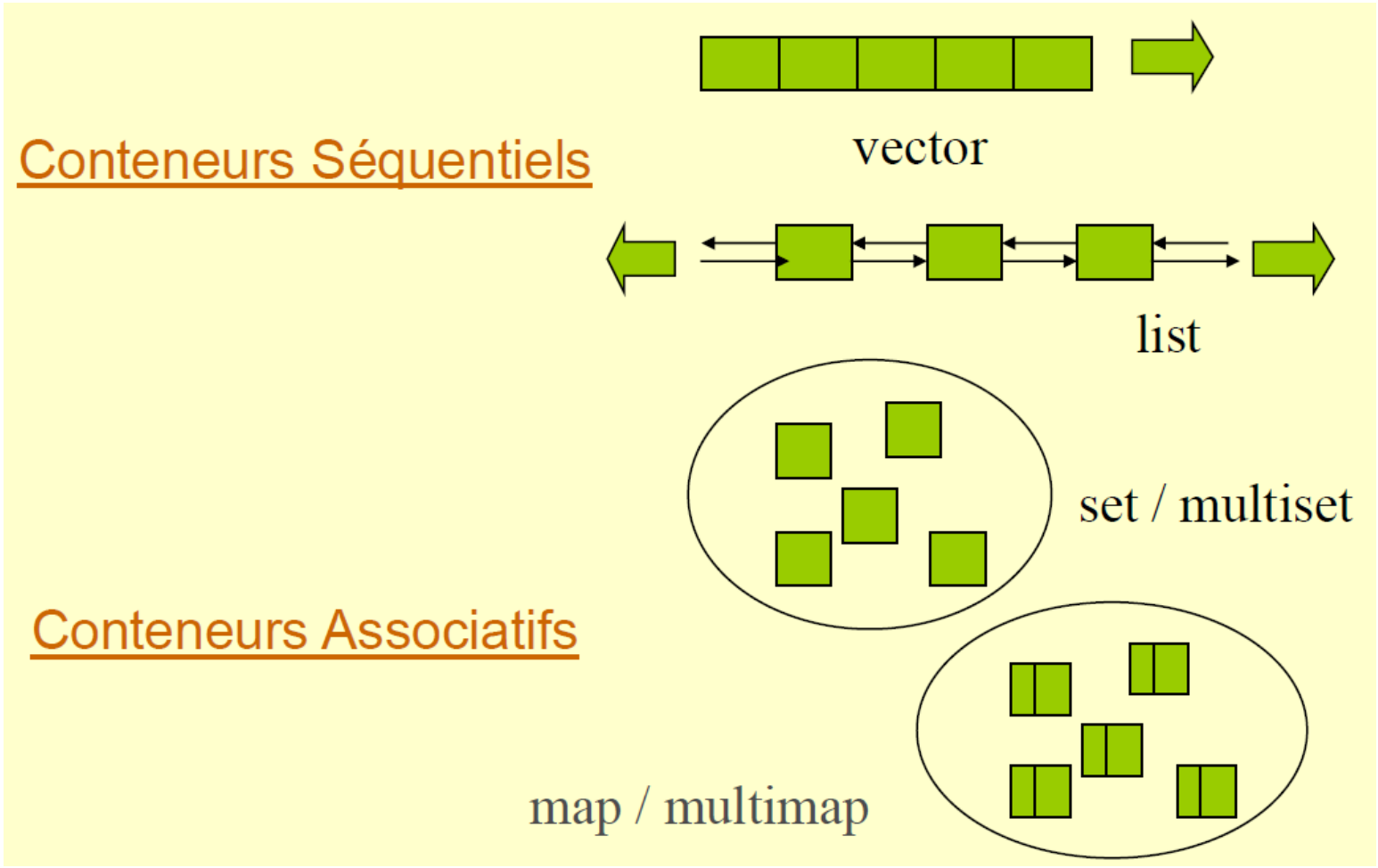
Composants STL

conteneurs : Ils sont utilisés pour manipuler des objets d'un même type. On parle donc d'une collection d'objets La STL fournit différentes sortes de conteneurs pour combler différents besoins

Itérateurs : Les itérateurs sont des objets qui permettent de naviguer parmi les éléments d'un conteneur Un itérateur est un pointeur « intelligent » L'itérateur fait le lien entre les conteneurs et les algorithmes

Algorithmes : Les algorithmes de la STL offrent des services fondamentaux tels que recherche, tri, copie, modification des éléments des conteneurs. Les algorithmes sont des fonctions globales qui opèrent avec des itérateurs

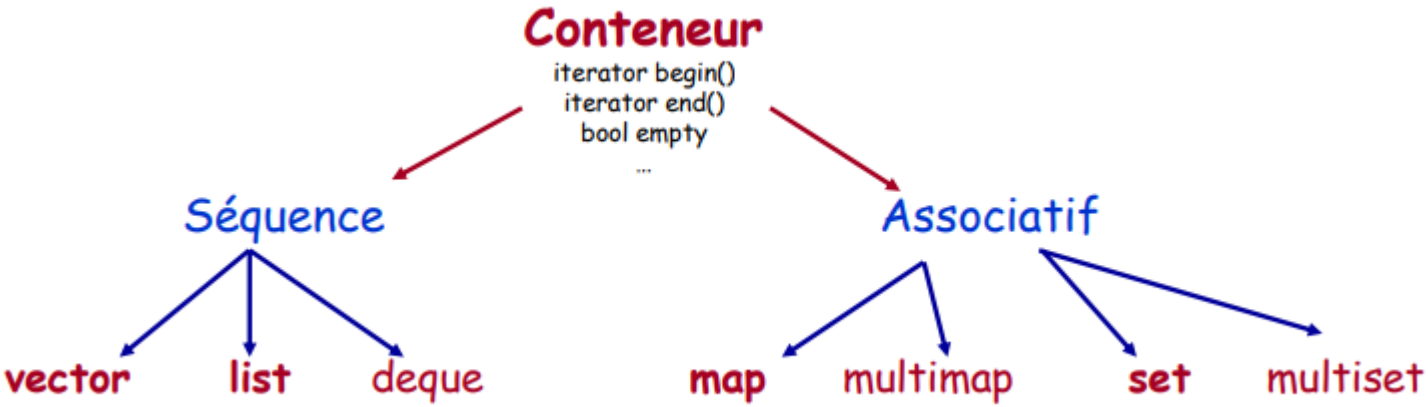
Les conteneurs



Les conteneurs séquentiels

Les éléments du conteneur ont une position qui dépend du temps et de l'endroit de l'insertion ,

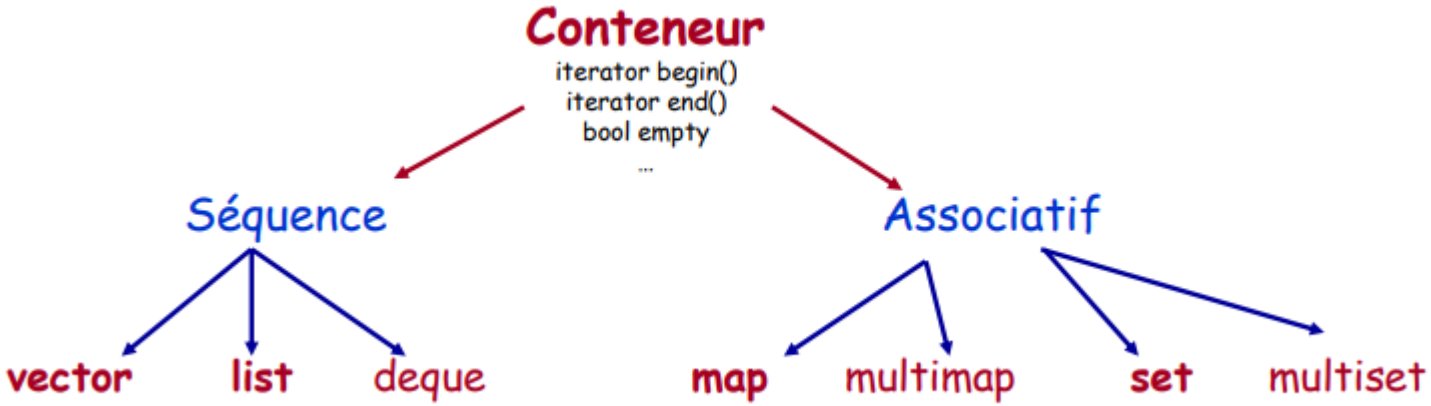
La position est donc indépendante de la valeur de l'élément vector, list et deque sont des conteneurs séquentiels



Les conteneurs associatifs

Les éléments du conteneur sont automatiquement triés lors de leur insertion selon un critère précis.

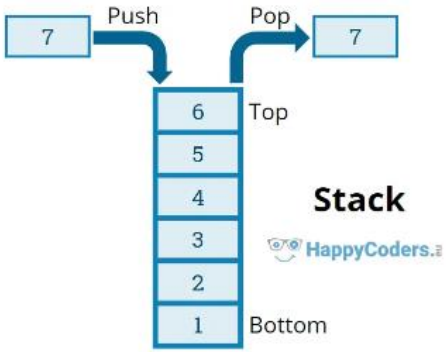
- Le critère de tri prend la forme d'une comparaison de la valeur de l'élément ou d'une clé spéciale définie pour l'élément.
- Les conteneurs associatifs sont généralement représentés sous forme d'arbre binaire
- set, multiset, map et multimap sont des conteneurs associatifs



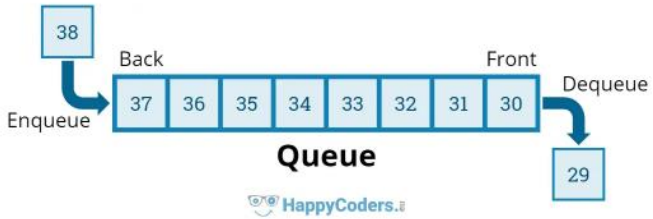
Les conteneurs dérivés

Le standard C++ fournit quelques conteneurs supplémentaires:

- **Piles** (stacks): conteneur manipulant ses éléments selon la politique LIFO (last-in-first-out)



- **Files** (queues): conteneur manipulant ses éléments selon la politique FIFO (first-in-first-out)



Choix du conteneur

Les **vector** et les **deque** permettent d'accéder rapidement à un élément de la liste (opérateur []) contrairement aux **list**

- Un élément peut être inséré
  - ✓ n'importe où pour une **list**
  - ✓ à la fin ou au début pour un **deque**
  - ✓ à la fin seulement pour un **vector**

- Utiliser un **vector** est un bon choix lorsque:
  - ✓ on a besoin d'insérer/supprimer des éléments seulement à la fin
  - ✓ le conteneur doit être compatible au tableau C standard

Choix du conteneur (suite)

- Utiliser un **deque** est un bon choix lorsque:
  - ✓ on a besoin d'insérer/supprimer des éléments seulement à la fin ou au début
  - ✓ le conteneur ne doit pas être compatible avec un tableau C standard
  - ✓ la taille maximum requise du conteneur n'est pas connue

- Utiliser une **list** est un bon choix lorsque:

  - ✓ on a besoin d'insérer/supprimer des éléments au milieu du conteneur
  - ✓ le conteneur ne doit pas être compatible avec un tableau C standard
  - ✓ la taille maximum requise du conteneur n'est pas connue

Fonctions membres des conteneurs

Tous les conteneurs offrent les mêmes fonctions de base permettant aux itérateurs d'accéder aux éléments

- **begin()** : Retourne un itérateur représentant le début des éléments dans le conteneur
- **end()** : Retourne un itérateur représentant la fin des éléments dans le conteneur
- **size()** : Retourne le nombre d'éléments présents dans le conteneur

Tableau dynamique (vector)

Un tableau dynamique est un objet vector.
Il faut ajouter la ligne #include <vector> pour utiliser ces tableaux.

On utilise la syntaxe suivante pour le déclarer : vector<type> nom(taille);

```
int main()
{
   int const nombreNotes(5);
   vector<int> tableau(nombreNotes, 3);  //Crée un tableau de 5 entiers
valant tous 3

   for(int i(0); i<nombreNotes; i++)
       cout << tableau[i] << endl;
}
```

Les éléments sont insérés à la fin du tableau

Pour changer la taille d'un tableau de façon dynamique, il faut utiliser la fonction **push_back()** ,

Tableau dynamique (vector)

Pour changer la taille d'un tableau de façon dynamique, il faut utiliser la fonction push_back() :

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

int main()
{
   vector<double> notes;//Un tableau vide

   notes.push_back(12.5);//On ajoute des valeurs
   notes.push_back(19.5);
   notes.push_back(6);

   double moyenne(0);
   for(int i(0); i<notes.size(); ++i)
   //On utilise notes.size() pour la limite de notre boucle
   {
       moyenne += notes[i];   //On additionne toutes les notes
   }
   moyenne /= notes.size();
   //On utilise à nouveau notes.size() pour obtenir le nombre de notes
   cout << "La moyenne est : " << moyenne << endl;
   return 0;
}
```

Tableau dynamique (vector)

Pour des raisons d'efficacité, on préférera passer un tableau par référence à une fonction :

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

//Une fonction recevant un tableau d'entiers en argument
void fonction(vector<int>& tab)
{
    //…
}

int main()
{
   vector<int> tableau(3,2);   //On crée un tableau de 3 entiers valant 2
   fonction(tableau);          //On passe le tableau à la fonction

return 0;
}
```

Tableau dynamique (vector)

Si ce tableau ne doit pas être modifié, on fera un passage par référence constante :

```
void fonction(vector<int> const& tableau)
```

Remarque : si vous utilisez une fonction prototype (définition d'une fonction dans un fichier d'en-tête .h), il faudra spécifier l'espace de nom.

```
void fonction(std::vector<int> const&);
```

Utilisation des itérateurs

Tous les conteneurs définissent deux types d'itérateur

- **container::iterator** :
  - permet de naviguer en mode lecture/écriture

- **container::const_iterator**
  - permet de naviguer en mode lecture seulement

```
#include <list>
#include <iterator>

list<char> collection; // création d'une liste de caractères
for (char c = 'a'; c<='z'; c++)
{
collection.push_back(c);
}

// impression de la liste à l'aide d'un itérateur
list<char>::const_iterator pos;
for(pos = collection.begin(); pos != collection.end(); pos++)
{
cout << *pos << ' ';
}
```

Tableau dynamique

assign() Affecte, depuis le début, une valeur à plusieurs objets

```
vec.assign(2, 77); // les 2 premiers reçoivent 77
//Affecte, depuis le début, la séquence d'un autre conteneur
vec.assign(tab, tab+5); // vec[0 à 5] <- tab[0 à 5]
```

at(index)=val Idem à [index] mais plus sur : Vérifie la type de val et les limites (prévoir Trait Error)

```
vec.at(1)=1000; // Affecte 100 au 2ème
```

back() Retourne la valeur du dernier objet.

begin() Retourne l'itérateur Premier : position du 1er objet.

capacity() Retourne la capacité, le nombre d'objets possibles de stocker dans le vector

clear() Efface tous les objets du vector

empty() Teste si le vector est vide (true si pas d'objet size()==0)

erase(itrPrem, itrDer) Efface les objets de la séquence définie par les itérateurs

```
vec.erase(vec.begin+1(), vec.end()-6);
//Efface l'objet pointé par l'itérateur
vec.erase(vec.begin+4());
```

end() Retourne l'itérateur Fin : position suivante après le dernier objet du vector

front() Retourne la valeur du 1er objet

Tableau dynamique

insert() Insérer un objet à une position

```
vec.insert(vec.begin()+2, 33); // *(Début+2) =33
//Insérer une série d'objets de même valeurs, à partir d'une position
vec.insert(vec.begin()+5, 3, 1789);
// A partir de Début+5, insère 3 objets de valeur 1789
//Insérer, à partir la position, une séquence d'un autre conteneur.
vec.insert(vec.begin()+7, vec3.begin()+2, vec3.end()-6 );
```

pop_back() Retire le dernier objet et retourne sa valeur

push_back(val) Ajoute un objet à la fin et lui affecte la valeur « val »

reserve(nbre) Réserve de la mémoire pour stocker « nbre » objets. Cette fonction permet de seulement augmenter la capacité avec éventuellement une réallocation mémoire du vector.

resize (nbre) La capacité est adaptée pour stocker « nbre » d'objets. La capacité du vector peut être diminuée avec cette méthode.

size() Retourne le nombre d'objets contenus dans le vector.

swap Echange des valeurs avec un autre conteneur (même type d'objets)

[ ] Accès direct indexé à la valeur d'un élément du vector (comme un tableau)

***** Opérateur d'indirection : *vec1.begin() retourne la valeur pointée par begin()

Exemple d'algorithmes

```
#include <vector>
#include <algorithm>    // std::max
vector<int> coll;
vector<int>::iterator pos;

// trouver le minimum et le maximum

pos = min_element(coll.begin(), coll.end());
cout << "min: " << *pos << endl;

pos = max_element(coll.begin(), coll.end());
cout << "max: " << *pos << endl;

// tri
sort(coll.begin(), coll.end());
```

Exemple

```
Personne p1("PROVOLO","Alain");
Personne p2("DELATTRE","Alain");
Personne p3("CHARLIER","Arnaud");
Personne p4("LEPLA","Stephane");

vector <Personne> Professeurs;

cout << "Taille de la collection " << Professeurs.size() << endl;

    cout << "Insertion " << endl;
Professeurs.push_back(p1);
Professeurs.push_back(p2);
Professeurs.push_back(p3);
Professeurs.push_back(p4);
cout << "Taille de la collection " << Professeurs.size() << endl;
cout << "Parcours notation tableau " << endl;
for(int i=0; i < Professeurs.size();i++)
{
  cout << Professeurs[i].getNom() << endl;

}
```

Exemple cours (suite)

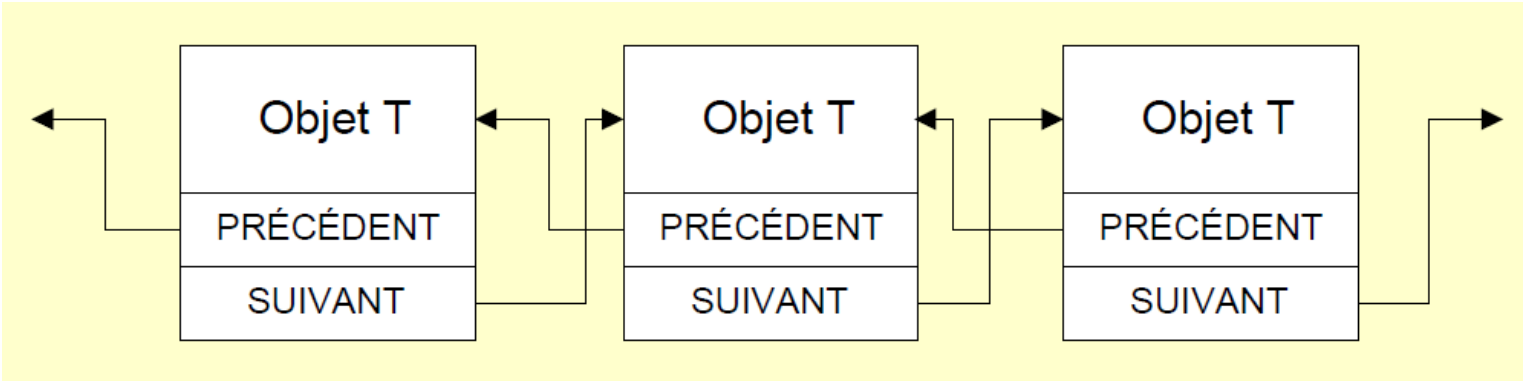
```
Personne p = Professeurs.back() ;
Professeurs.pop_back();
cout <<  endl;

 cout << "Parcours iterateur " << endl;
 vector<Personne>::iterator it;

 for(it = Professeurs.begin(); it != Professeurs.end();it++)
 {

 cout << (*it).getNom()<< endl;
 }
cout << " " << endl;


getch();
return 0;
}
```

```
class Personne
{
private:
string nom;
string prenom;
public :
Personne(){};
Personne(string n,string p){nom = n; prenom=p;}
~Personne(){};
string  getNom(){return nom;}
};
```

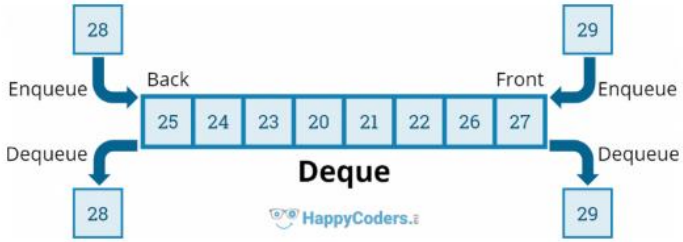
List

Liste doublement chaînée d'éléments

- Un élément pointe sur son prédécesseur et son successeur dans la liste
- Les éléments sont insérés n'importe où dans la liste



Deque



Deque = double-ended queue

- Tableau dynamique
- Les éléments sont insérés à la fin ou au début du tableau

```
#include <deque>

deque<int> coll;
for(int i=1; i<=6; i++)
{
    coll.push_back(i);
    coll.push_front(i);
}
```

Le tableau associatif Set

Les éléments sont triés selon leur valeur. Chacun des éléments ne peut exister qu'une fois

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

int main()
 {
    set<string> Heros;
    set<string>::iterator H;
    set<int> Nbre;
    set<int>::iterator it;

    Heros.insert("Tintin");
    Heros.insert("Asterix");
    Heros.insert("Obelix");
    Heros.insert("Asterix");
    Heros.insert("Titeuf");
    Heros.insert("Obelix");

    for (H= Heros.begin(); H != Heros.end(); H++)
    {
           cout<< *H << endl;
    }
    Nbre.insert(9); Nbre.insert(3); Nbre.insert(6);
    for (it= Nbre.begin(); it != Nbre.end(); it++)
    {
         cout<< *it << endl;
    }
    return 0;
}
```

Affichage :

Asterix
Obelix
Tintin
Titeuf

3
6
9

Le tableau associatif Multiset

Les éléments sont triés selon leur valeur. Chacun des éléments peut exister plus d'une fois

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

using namespace std;

int main() {
    multiset<string> Heros;
    multiset<string>::iterator H;
    set<int> Nbre;
    set<int>::iterator it;
    Heros.insert("Tintin");
    Heros.insert("Asterix");
    Heros.insert("Obelix");
    Heros.insert("Asterix");
    Heros.insert("Titeuf");
    Heros.insert("Obelix");
    for (H= Heros.begin(); H != Heros.end(); H++)
        cout<< *H << endl;
    Nbre.insert(9); Nbre.insert(3); Nbre.insert(6);
    for (it= Nbre.begin(); it != Nbre.end(); it++)
        cout<< *it << endl;
    return 0;
}
```

Affichage :

```
Asterix
Asterix
Obelix
Obelix
Tintin
Titeuf
3
6
9
```

Map

Les éléments sont une paire formée d'une clé jumelée à une valeur.

- Les éléments sont triés selon leur clé.
- Chacune des clés ne peut exister qu'une fois

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

int main() {
    map<string, float> poids; // table qui associe le nom d'un animal à son poids
    map<string, float>::iterator it;
    //On ajoute les poids de quelques animaux // poids[K] = V ;
    poids["souris"] = 0.05;
    poids["tigre"] = 200;
    poids["chat"] = 3;
    poids["elephant"] = 10000;
    for(it=poids.begin(); it!=poids.end(); it++) {
    cout << it->first << " pese " << it->second << " kg." << endl;
    }
    return 0;
}
```

```
chat pese 3 kg.
elephant pese 10000 kg.
souris pese 0.05 kg.
tigre pese 200 kg.
```

Map (suite)

Autre exemple

```
int  main() {
    map<string,int> M; // Pour chaque clé correspondra une seule
valeur associée.
    map<string,int>::iterator it;

    M["Tintin"] = 2;
    M.insert(make_pair("Asterix",5));
    M.insert(make_pair("Obelix",9));
    M.insert(make_pair("Obelix",9));
    M.insert(make_pair("Obelix",7));

    for(it= M.begin(); it!=M.end(); it++) {
    cout << it->first << ":"  << it->second  << endl;
    }
    return 0;
}
```

M[Asterix] = 5
M[Obelix] = 9
M[Tintin] = 2

Multimap

Même chose que map excepté qu'une clé peut exister plus d'une fois.

- On peut donc retrouver des éléments possédant la même clé mais des valeurs différentes.
- Exemple : un dictionnaire

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

int  main() {
    multimap<string,int> M; // Pour chaque clé correspondra une seule valeur associée.
    multimap<string,int>::iterator it;

    M["Tintin"] = 2;
    M.insert(make_pair("Asterix",5));
    M.insert(make_pair("Obelix",9));
    M.insert(make_pair("Obelix",9));
    M.insert(make_pair("Obelix",7));

    for(it= M.begin(); it!=M.end(); it++) {
    cout << it->first << "="  << it->second  << endl;
    }
    return 0;
}
```

M[Asterix] = 5 et
M[Obelix] = 9 et
M[Obelix] = 9 et
M[Obelix] = 7 et