



PОО – Langage C++

Généralités sur STL

Standard Template Library

1^{ère} année ingénieur informatique

Mme Wiem Yaiche Elleuch

2018 - 2019



introduction

- La STL est une partie importante de la bibliothèque standard du C++
- <http://www.sgi.com/tech/stl/>
 - développée par Hewlett-Packard puis Silicon Graphics
 - 1994 : adoptée par le comité de standardisation du C++, pour être intégrée au langage
- La STL propose un grand nombre d'éléments:
 - des structures de données évoluées (conteneur/collection) (tableaux, listes chaînées, vecteurs, piles, ...),
 - des algorithmes efficaces sur ces structures de données (ajout, recherche, parcours, suppression, ...).

Définitions

- Les STL :
 - utilisent les templates,
 - n'utilisent pas l'héritage et les fonctions virtuelles pour des raisons d'efficacité.
- la STL propose principalement trois types d'éléments :
 - les conteneurs (containers)(collections): de base ou évolués (adaptators) pour stocker les données (vecteur, listes, ...)
 - les itérateurs (iterators) pour parcourir les conteneurs,
 - les algorithmes travaillent sur les conteneurs via les itérateurs.

Templates conteneurs collections

- `#include <algorithm > // algorithmes utiles (tri , ...)`
- `#include <deque > // tableaux dynamiques (extensibles a chaque extremite)`
- `#include <functional > // objets fonctions predefinis (unary_function , ...)`
- `#include <iterator > // iterateurs , fonctions , adaptateurs`
- `#include <list > // listes doublement chainees`
- `#include <map > // maps et multimaps`
- `#include <numeric > // fonctions numeriques : accumulation , ...`
- `#include <queue > // files`
- `#include <set > // ensembles`
- `#include <stack > // piles`
- `#include <string > // chaines de caracteres`
- `#include <utility > // divers utilitaires`
- `#include <vector > // vecteurs`

plan

- **Notions de conteneur, d'itérateur et d'algorithme**
 - ➔ Ces trois notions sont étroitement liées et, la plupart du temps, elles interviennent simultanément dans un programme utilisant des conteneurs
- Les différentes sortes de conteneurs
- Fonctions, prédicats et classes fonctions

Notion de conteneur

- La bibliothèque standard fournit un ensemble de **classes** dites **conteneurs**, permettant de représenter les structures de données les plus répandues telles que les **vecteurs**, les **listes**, les **ensembles** ou les tableaux associatifs.
- Il s'agit de **patrons de classes** par le type de leurs éléments.

```
list<int> l;    // liste vide d'éléments de type int
vector<double> v; // vecteur vide d'éléments de type double
list<point> lp; // liste vide d'éléments de type point
```

Notion d'itérateur

- Les itérateurs sont des **objets** fondamentaux de la STL, ils permettent de manipuler les conteneurs de manière transparente, plus facilement.
- Un itérateur permet :
 - de parcourir un conteneur,
 - d'accéder aux données contenues dans le conteneur,
 - et (éventuellement) de modifier les données.
- Un itérateur peut être vu comme un **pointeur** sur un élément du conteneur.
- **Iterator** ➔ parcours d'un conteneur du début à la fin, (ou d'une séquence (intervalle)).
- **reverse_iterator** ➔ parcours d'un conteneur en sens inverse (de la fin au début ou d'une séquence (intervalle))

Utilisation générale d'un itérateur

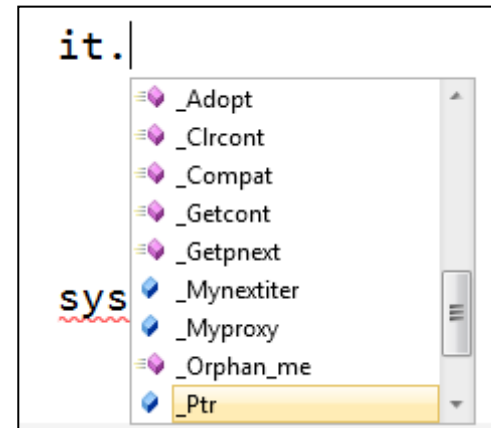
- Un itérateur peut être initialisé grâce aux méthodes :
 - `begin()` : retourne un itérateur “pointant” sur le premier élément du conteneur
 - `end()` retourne un itérateur “pointant” sur l’élément juste après le dernier élément du conteneur
 - ou toutes autres méthodes retournant un itérateur sur le conteneur : `find()`, ...
- De manière similaire, un `reverse_iterator` peut être initialisé en utilisant les méthodes : `rbegin()` / `rend()`, ...
- `rbegin()` pointe sur le dernier élément du conteneur
- `rend()` pointe juste avant le premier.
- Accéder à l’élément “pointé” par un itérateur `it`: `(*it)`

Notion d'itérateur

```
vector<int> v; // vecteur v  
vector<int>::iterator it; // itérateur it  
vector<int>::reverse_iterator rit; // reverse_iterator rit
```

Il existe 3 types d'itérateurs:

1. Itérateur unidirectionnel
2. Itérateur bidirectionnel
3. Itérateur à accès direct



Propriétés des itérateurs

- **itérateur unidirectionnel**

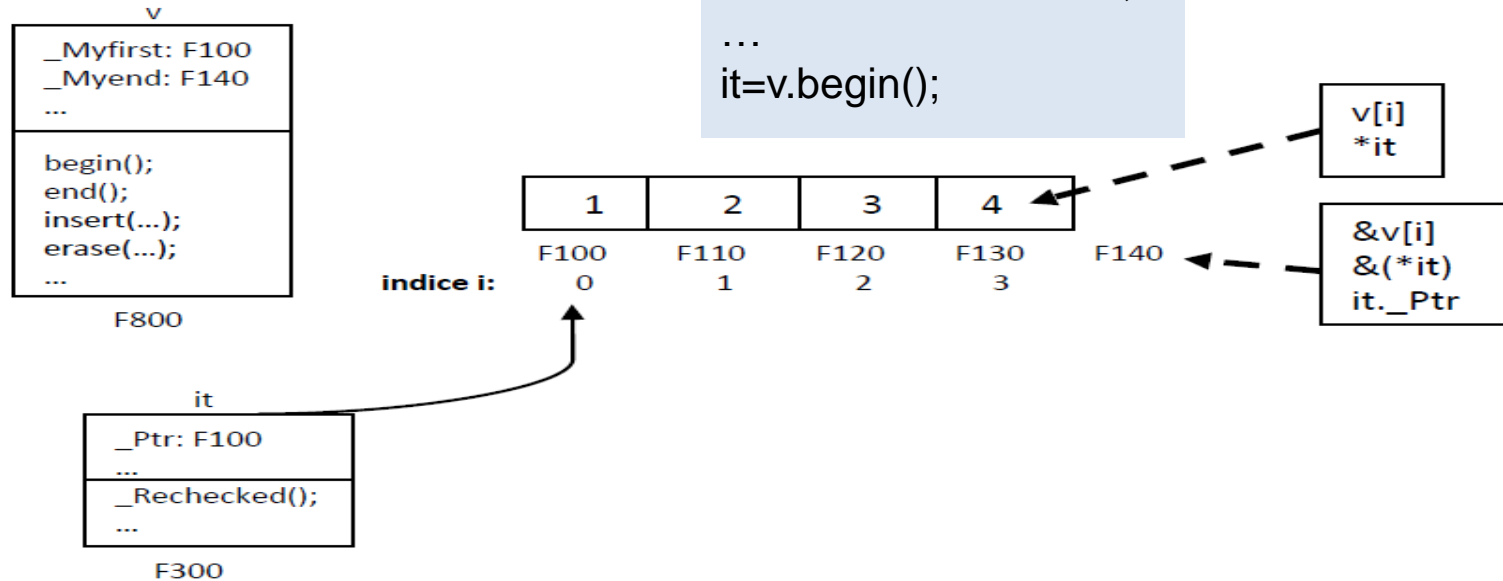
- A un instant donné, un itérateur possède une valeur qui désigne un élément donné d'un conteneur; (un itérateur **pointe** sur un élément d'un conteneur)
- un itérateur peut être incrémenté par l'opérateur **++**, de manière à pointer sur l'élément suivant du même conteneur;
- un itérateur peut être déréférencé, comme un pointeur, en utilisant **l'opérateur *** ; par exemple, si *it* est un itérateur sur une liste de points, **it* désigne un point de cette liste;
- deux itérateurs sur un même conteneur peuvent être comparés par égalité ou inégalité **==** ou **!=**.

- **itérateur bidirectionnel:** est un itérateur unidirectionnel qui en plus, peut être décrémenté par **l'opérateur --** ;

- **Itérateur à accès direct:** si *it* est un tel itérateur , ***it[i]*** est équivalent à ****(it+i)***. ***It+=n; it-=n; Comparaison <, <=, >, >=***

La classe vector

```
vector<int> v;
vector<int>::iterator it;
...
it=v.begin();
```



fonctions	description
<code>v.push_back(val);</code>	ajoute la valeur <code>val</code> à la fin du vecteur <code>v</code> .
<code>v.size();</code>	retourne le nombre d'éléments dans le vecteur <code>v</code> .
<code>v[i]</code>	l'élément d'indice <code>i</code> dans le vecteur <code>v</code> (1 ^{er} élément d'indice 0).
<code>v.begin()</code>	Retourne <u>un itérateur</u> qui pointe sur le premier élément du vecteur
<code>v.end()-1</code>	Retourne <u>un itérateur</u> qui pointe sur le dernier élément du vecteur
<code>v.erase(v.begin()+i);</code>	supprime l'élément d'indice <code>i</code>
<code>v.insert(v.begin()+i, val);</code>	insère la valeur <code>val</code> à la position d'indice <code>i</code>
<code>v.pop_back();</code>	supprime le dernier élément
<code>v.clear();</code>	supprime tous les éléments du vecteur (vider le vecteur)

```
void main ()
```

```
{
```

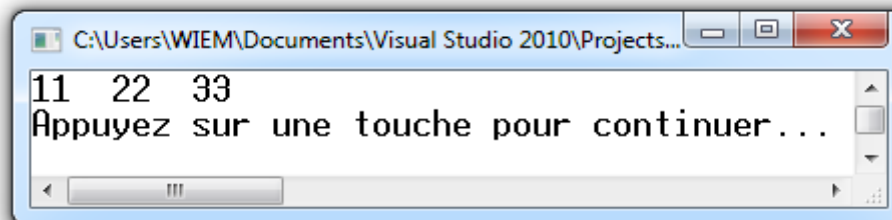
```
vector<int> v;  
vector<int>::iterator it;
```

Parcours d'un conteneur avec un itérateur

```
v.push_back(11);  
v.push_back(22);  
v.push_back(33);  
// v: 11 22 33
```

```
for(it=v.begin(); it!=v.end() ; it++)  
    cout<<*it<<" ";  
cout<<endl;  
system("PAUSE");
```

```
}
```



```
void main ()
```

```
{
```

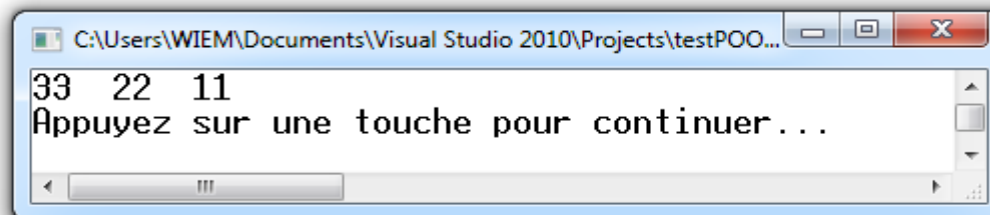
```
vector<int> v;  
vector<int>::reverse_iterator rit;
```

Parcours d'un conteneur avec un reverse_iterator

```
v.push_back(11);  
v.push_back(22);  
v.push_back(33);  
// v: 11 22 33
```

```
for(rit=v.rbegin(); rit!=v.rend() ; rit++)  
    cout<<*rit<<" ";  
cout<<endl;  
system("PAUSE");
```

```
}
```



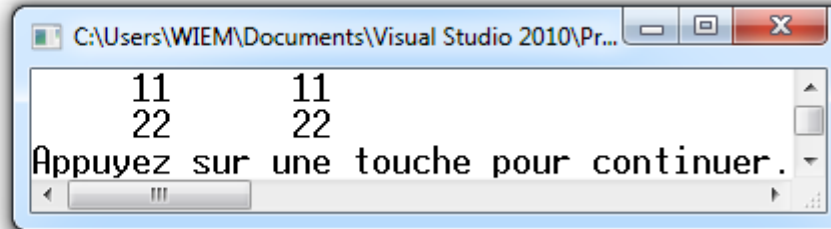
Parcours d'un conteneur de points avec un itérateur

```
#include<list>
void main()
{
    list<point> l;
    list<point>::iterator it;

    point a(11,11);
    point b(22,22);
    l.push_back(a);
    l.push_back(b);

    for(it=l.begin() ; it!=l.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



remarque

- la valeur de l'itérateur de fin consiste à pointer, non pas sur le dernier élément d'un conteneur, mais juste après.
- lorsqu'un conteneur est vide, *begin()* possède la même valeur que *end()*,
- Dans la boucle for (condition d'arrêt)
 - Puisque le conteneur est un vector: on aurait pu écrire `it<l.end();`
 - Si le conteneur est une liste, `it<l.end();` // **ERREUR**
 - ➔ l'opérateur `<` ne peut s'appliquer qu'à des itérateurs à accès direct.

Les algorithmes

- La notion d'algorithme se fonde sur le fait que, par le biais d'un itérateur, beaucoup d'opérations peuvent être appliquées à un conteneur, quels que soient sa nature et le type de ses éléments →
Les algorithmes opèrent sur les conteneurs uniquement via les itérateurs
- La STL fournit environs 70 algorithmes destinés à manipuler des conteneurs.
- 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**, etc.
- Il faudra inclure le fichier d'en-tête `#include <algorithm>`

Les algorithmes

- On distingue les algorithmes
 - **mutables, c'est à dire qui modifient les éléments du conteneur.**

<code>copy ()</code>	<code>partition()</code>	<code>replace_copy()</code>	<code>stable_partition()</code>
<code>copy_backward()</code>	<code>random_shuffle()</code>	<code>replace_copy_if()</code>	<code>swap()</code>
<code>fill()</code>	<code>remove()</code>	<code>replace_if()</code>	<code>swap_ranges()</code>
<code>fill_n()</code>	<code>remove_copy()</code>	<code>reverse()</code>	<code>transform()</code>
<code>generate()</code>	<code>remove_copy_if()</code>	<code>reverse_copy()</code>	<code>unique()</code>
<code>generate_n()</code>	<code>remove_if()</code>	<code>rotate()</code>	<code>unique_copy()</code>
<code>iter_swap()</code>	<code>replace()</code>	<code>rotate_copy()</code>	

- les algorithmes **non mutables**

<code>adjacent-find ()</code>	<code>equal()</code>	<code>find_end()</code>	<code>mismatch()</code>
<code>count()</code>	<code>find()</code>	<code>find_first_of()</code>	<code>search()</code>
<code>count_if()</code>	<code>find_each()</code>	<code>find_if()</code>	<code>search_n()</code>

- et les algorithmes numériques (fichier d'en-tête <numeric>)

<code>accumulate ()</code>	<code>inner_product()</code>	<code>partial_sum()</code>	<code>adjacent_difference()</code>
----------------------------	------------------------------	----------------------------	------------------------------------

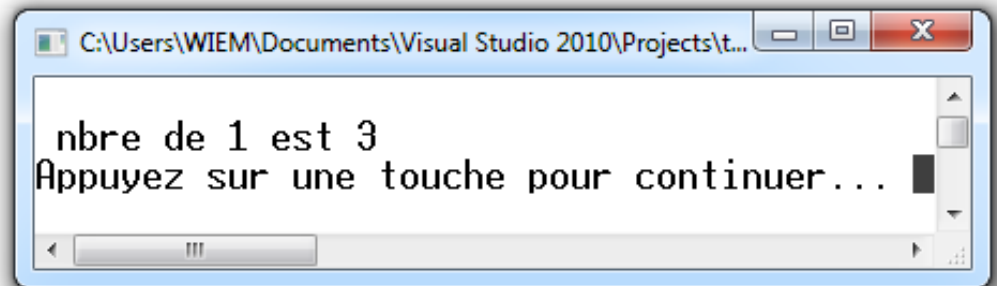
Algorithme count: compter le nombre d'éléments égaux à 1 dans un vecteur

```
#include<list>
#include<algorithm>
void main()
{
    vector<int> v;

    v.push_back(1);
    v.push_back(5);
    v.push_back(1);
    v.push_back(4);
    v.push_back(1);

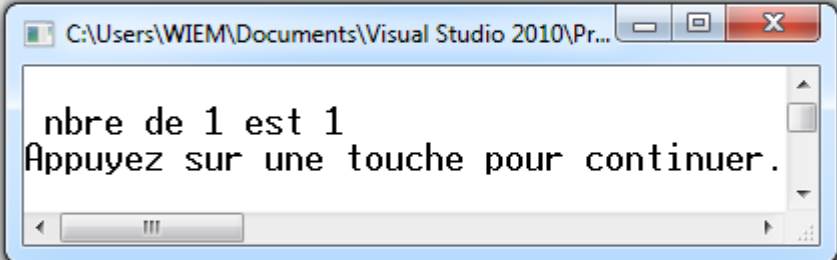
    int n=count(v.begin(), v.end(), 1);

    cout<<"\n nbre de 1 est "<<n<<endl;
    system("PAUSE");
}
```



Compter le nbre de 1 dans la **séquence** **(intervalle)** (v.begin()+1, v.end()-1)

```
vector scope
1 #include<list>
2 #include<algorithm>
3 void main()
4 {
5     vector<int> v;
6
7     v.push_back(1);
8     v.push_back(5);
9     v.push_back(1);
10    v.push_back(4);
11    v.push_back(1);
12
13    int n=count(v.begin()+1, v.end()-1, 1);
14
15    cout<<"\n nbre de 1 est "<<n<<endl;
16    system("PAUSE");
17 }
```



Les algorithmes

- D'une manière générale, les algorithmes s'appliquent, non pas à un conteneur, mais à **une séquence définie par un intervalle d'itérateur** ;
- Certains algorithmes permettront facilement de recopier des informations d'un conteneur d'un type donné vers un conteneur d'un autre type, à condition que ses éléments soient du même type que ceux du premier conteneur.

Copie d'une liste d'entiers dans un vecteur d'entiers

- Recopie l'intervalle [l.begin(), l.end()) à partir de l'emplacement pointé par v.begin()
- on fournit l'intervalle de départ, on ne mentionne que le début de celui d'arrivée

```
void afficher(vector<int> v)
{
    vector<int>::iterator it;
    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
}

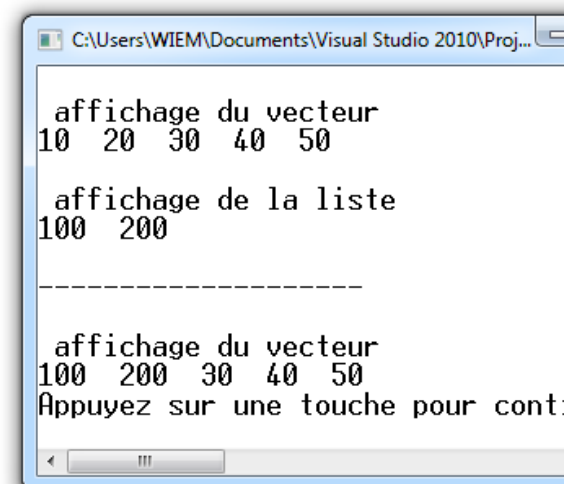
void afficher(list<int> l)
{
    list<int>::iterator it;
    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
}
```

```
void main()
{
    vector<int> v;
    list<int> l;

    for(int i=1; i<6; i++)
        v.push_back(10*i);
    afficher(v);

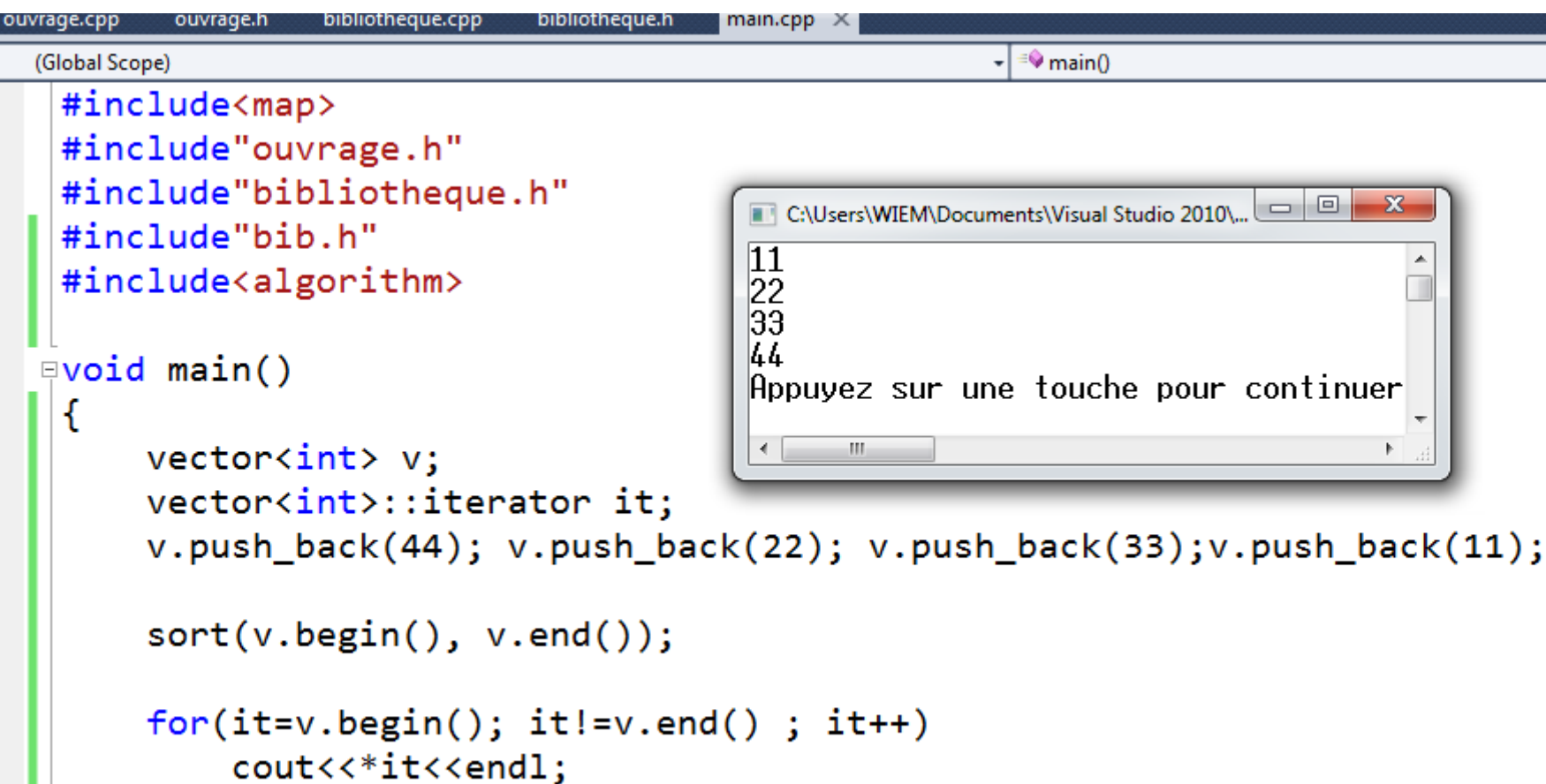
    for(int i=1; i<3; i++)
        l.push_back(100*i);
    afficher(l);

    copy(l.begin(), l.end(), v.begin());
    cout<<"\n-----"<<endl;
    afficher(v);
    system("PAUSE");
}
```



Quelques algorithmes STL

sort



The screenshot shows a Visual Studio 2010 IDE with a solution containing four files: `ouvrage.cpp`, `ouvrage.h`, `bibliotheque.cpp`, and `bibliotheque.h`. The `main.cpp` file is active, showing the following code:

```
#include<map>
#include"ouvrage.h"
#include"bibliotheque.h"
#include"bib.h"
#include<algorithm>

void main()
{
    vector<int> v;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(11);

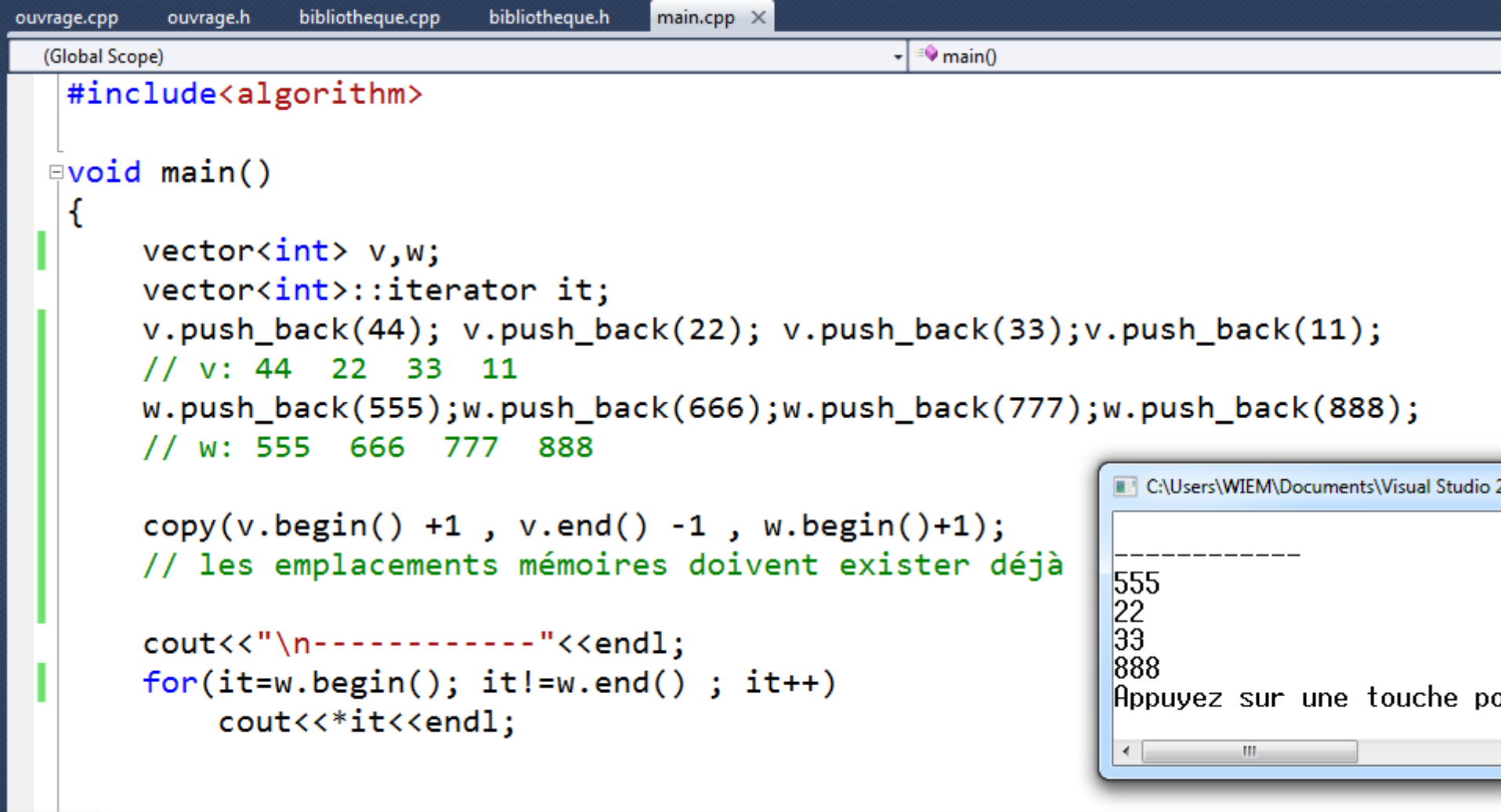
    sort(v.begin(), v.end());

    for(it=v.begin(); it!=v.end() ; it++)
        cout<<*it<<endl;
```

Overlaid on the code editor is a console window titled `C:\Users\WIEM\Documents\Visual Studio 2010\...`. The console displays the output of the program:

```
11
22
33
44
Appuyez sur une touche pour continuer
```

copy



```
ouvrage.cpp  ouvrage.h  bibliotheque.cpp  bibliotheque.h  main.cpp X
(Global Scope)  main()
#include<algorithm>

void main()
{
    vector<int> v,w;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(11);
    // v: 44  22  33  11
    w.push_back(555);w.push_back(666);w.push_back(777);w.push_back(888);
    // w: 555  666  777  888

    copy(v.begin() +1 , v.end() -1 , w.begin()+1);
    // les emplacements mémoires doivent exister déjà

    cout<<"\n-----"<<endl;
    for(it=w.begin(); it!=w.end() ; it++)
        cout<<*it<<endl;
}
```

C:\Users\WIEM\Documents\Visual Studio 2

555
22
33
888
Appuyez sur une touche po

copy

```
ouvrage.cpp  ouvrage.h  bibliotheque.cpp  bibliotheque.h  main.cpp* X
(Global Scope)  main()
#include<algorithm>

void main()
{
    vector<int> v,w;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(11);
    // v: 44  22  33  11
    w.push_back(555);w.push_back(666);
    // w: 555  666

    copy(v.begin() +1 , v.end() , w.begin()+1);
    // les emplacements memoires n'existent pas déjà

    cout<<"\n-----"<<endl;
    for(it=w.begin(); it!=w.end() ; it++)
        cout<<*it<<endl;
}
```

Microsoft Visual C++ Debug Library



Debug Assertion Failed!

Program: ...cuments\Visual Studio
2010\Projects\testPOO\Debug\testPOO.exe
File: c:\program files (x86)\microsoft visual studio
10.0\vc\include\vector
Line: 157

Expression: vector iterator + offset out of range

For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)

Abandonner

Recommencer

Ignorer

find

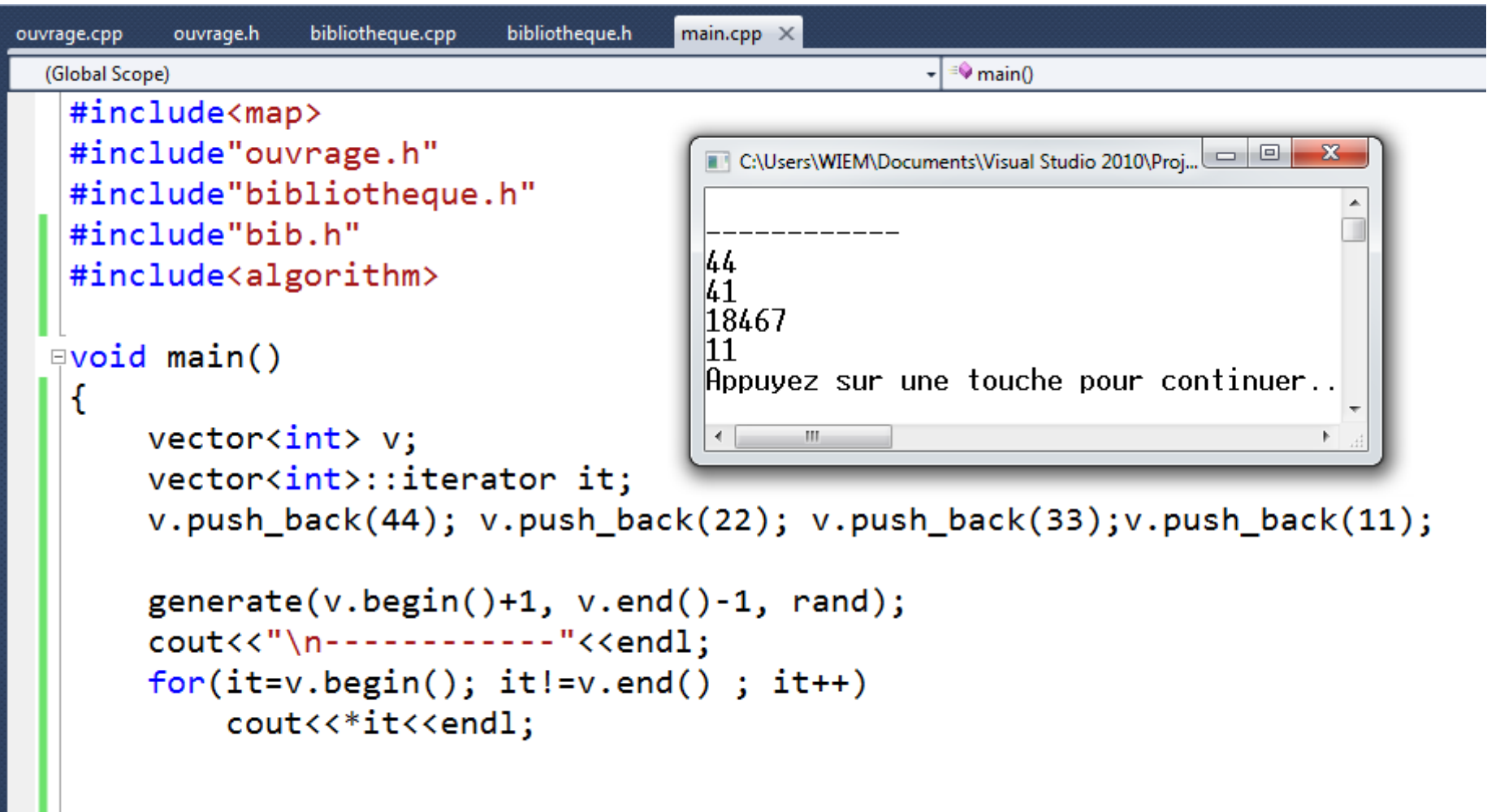
```
ouvrage.cpp  ouvrage.h  bibliotheque.cpp  bibliotheque.h  main.cpp X
(Global Scope)  main()

#include<algorithm>

void main()
{
    vector<int> v;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(22);

    it=find(v.begin(), v.end(), 66);
    if(it!=v.end()) cout<<*it<<endl;
    else cout<<"\n cette valeur n'existe pas "<<endl;
    cout<<"\n-----"<<endl;
    for(it=v.begin(); it!=v.end() ; it++)
        cout<<*it<<endl;
}
```

generate



The screenshot shows a Visual Studio 2010 IDE with the 'main.cpp' file open. The code defines a vector 'v' containing the values 44, 22, 33, and 11. It then uses the 'generate' function to fill the range from the second element to the third-to-last element with random values. The output window shows the resulting vector contents: 44, 41, 18467, and 11, followed by a prompt to press a key to continue.

```
ouvrage.cpp  ouvrage.h  bibliotheque.cpp  bibliotheque.h  main.cpp X
(Global Scope)  main()

#include<map>
#include"ouvrage.h"
#include"bibliotheque.h"
#include"bib.h"
#include<algorithm>

void main()
{
    vector<int> v;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(11);

    generate(v.begin()+1, v.end()-1, rand);
    cout<<"\n-----"<<endl;
    for(it=v.begin(); it!=v.end() ; it++)
        cout<<*it<<endl;
}
```

Output Window: C:\Users\WIEM\Documents\Visual Studio 2010\Proj...

44
41
18467
11
Appuyez sur une touche pour continuer..

count

```
ouvrage.cpp  ouvrage.h  bibliotheque.cpp  bibliotheque.h  main.cpp X
(Global Scope)
#include<map>
#include"ouvrage.h"
#include"bibliotheque.h"
#include"bib.h"
#include<algorithm>

void main()
{
    vector<int> v;
    vector<int>::iterator it;
    v.push_back(44); v.push_back(22); v.push_back(33);v.push_back(22);

    cout<<count(v.begin(), v.end(), 22)<<endl;
    cout<<"\n-----"<<endl;
    for(it=v.begin(); it!=v.end() ; it++)
        cout<<*it<<endl;
}
```

C:\Users\WIEM\Documents\Visual Studio 2010\Projects\testPOO\Debug\testPOO.exe

2

44

22

33

22

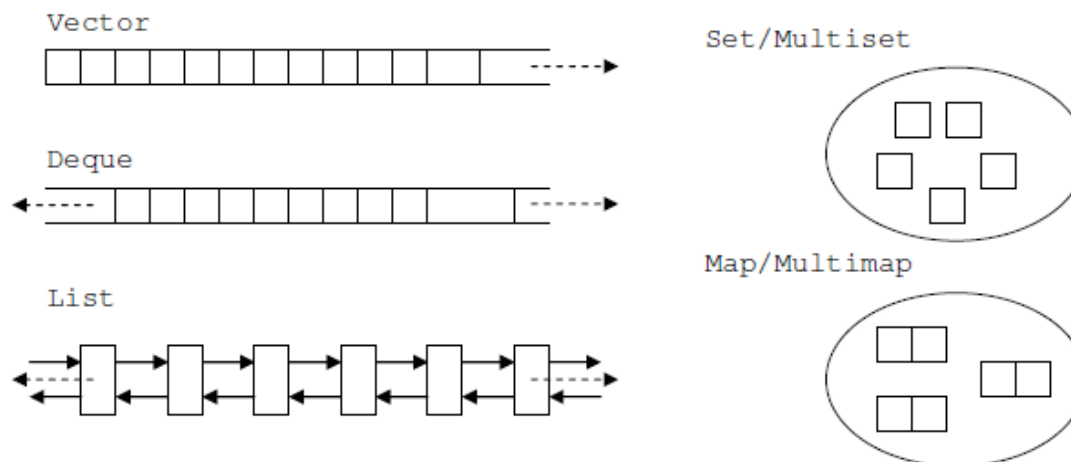
Appuyez sur une touche pour continuer...

plan

- Notions de conteneur, d'itérateur et d'algorithme
- **Les différentes sortes de conteneurs**
- Fonctions, prédicats et classes fonctions

Les différentes catégories de conteneurs

- La norme classe les différents conteneurs en deux catégories:
 - les conteneurs séquentiels
 - les conteneurs associatifs.
- La notion de conteneur en séquence correspond à des éléments qui sont ordonnés comme ceux d'un vecteur ou d'une liste.
- On peut parcourir le conteneur suivant cet ordre.
- Quand on insère ou qu'on supprime un élément, on le fait en un endroit qu'on a explicitement choisi.



Les différentes catégories de conteneurs

- La notion de conteneur associatif peut être illustrée par un répertoire téléphonique.
- Dans ce cas, on associe une valeur (numéro de téléphone, adresse...) à une clé (le nom).
- A partir de la clé, on accède à la valeur associée.
- Pour insérer un nouvel élément dans ce conteneur, il ne sera théoriquement plus utile de préciser un emplacement.

plan

- Notions de conteneur, d'itérateur et d'algorithme
- Les différentes sortes de conteneurs
- **Fonctions, prédicats et classes fonctions**

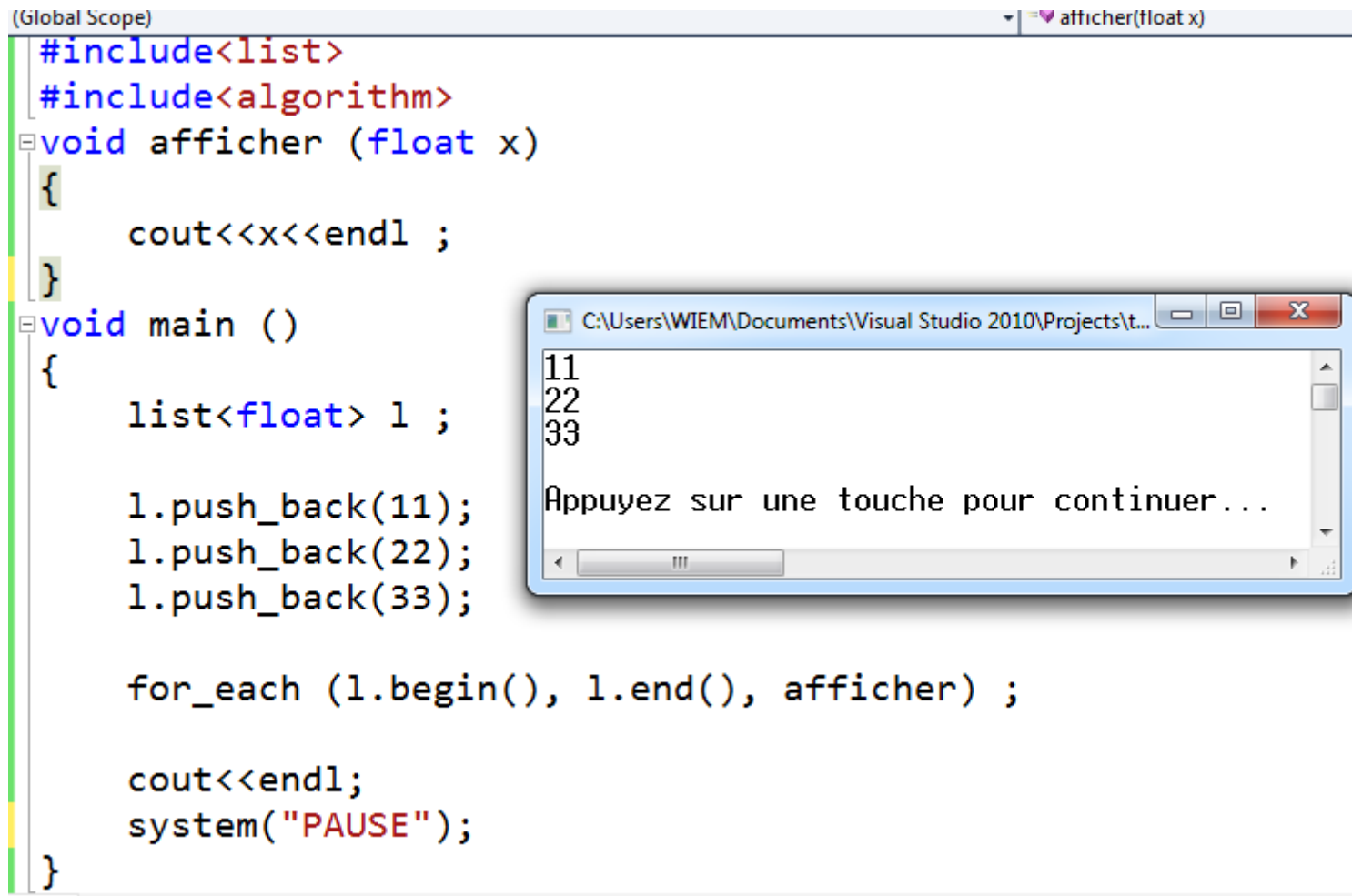
Fonction unaire

- Beaucoup d'algorithmes et quelques fonctions membres permettent d'appliquer une fonction donnée aux différents éléments d'une séquence (définie par un intervalle d'itérateur).
- Cette fonction est alors passée simplement en argument de l'algorithme, comme dans:

`for_each(it1, it2, f);` // applique la fonction **f** à chacun des éléments de la séquence **[it1, it2)**

- Bien entendu, la fonction **f** doit posséder un argument du type des éléments correspondants (dans le cas contraire, on obtiendrait une erreur de compilation).
- Il n'est pas interdit qu'une telle fonction possède une valeur de retour mais, elle ne sera pas utilisée.

afficher tous les éléments d'une liste:



```
(Global Scope) afficher(float x)
#include<list>
#include<algorithm>
void afficher (float x)
{
    cout<<x<<endl ;
}
void main ()
{
    list<float> l ;

    l.push_back(11);
    l.push_back(22);
    l.push_back(33);

    for_each (l.begin(), l.end(), afficher) ;

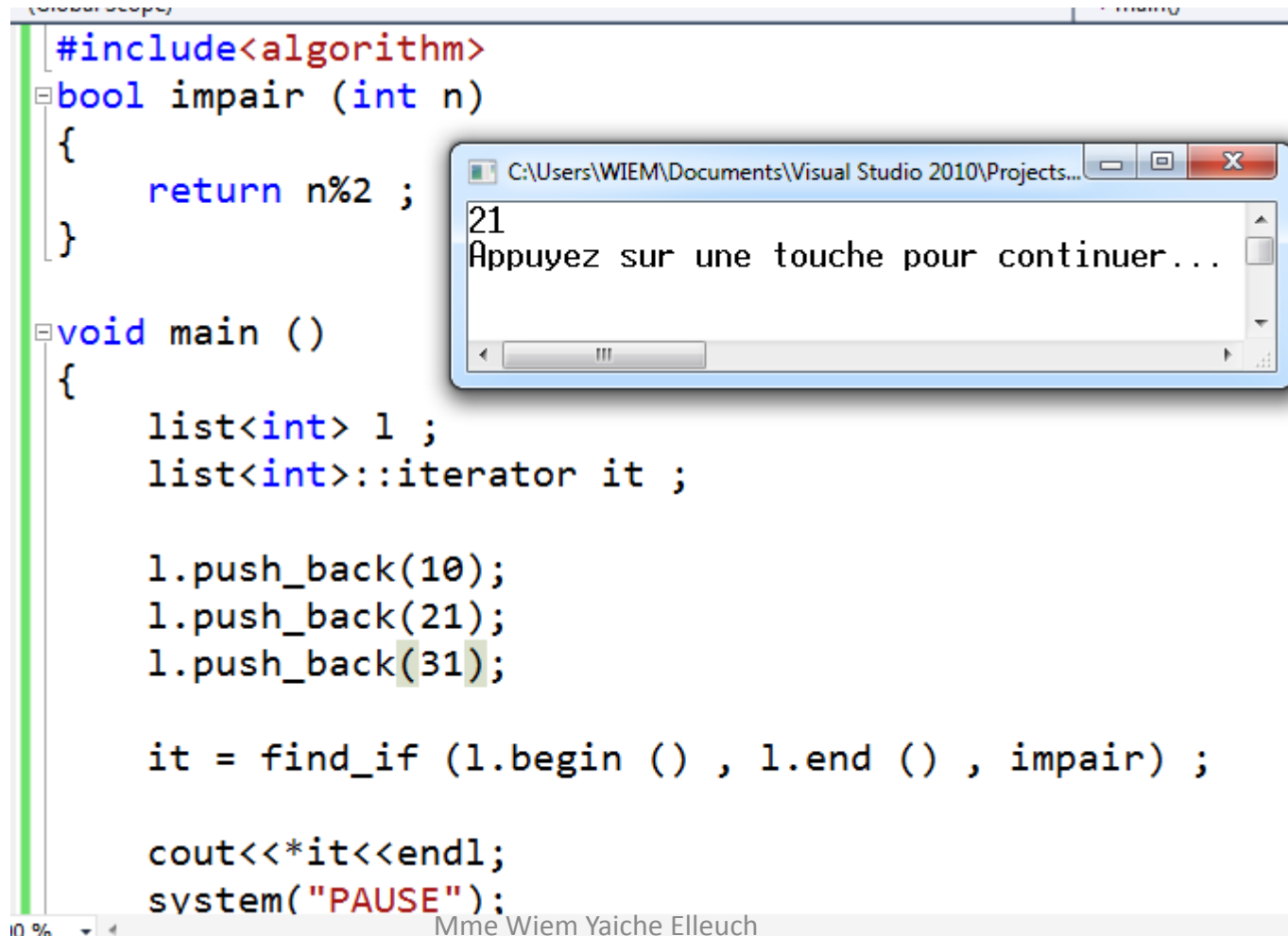
    cout<<endl;
    system("PAUSE");
}
```

11
22
33
Appuyez sur une touche pour continuer...

Prédicats

- On parle de **prédicat** pour caractériser une fonction qui renvoie une valeur de type *bool*.
- On rencontrera des prédicats unaires, c'est-à-dire disposant d'un seul argument et des prédicats binaires, c'est-à-dire disposant de deux arguments de même type.

- l'algorithme `find_if` permet de trouver le premier élément d'une séquence vérifiant un prédicat passé en argument:



The screenshot shows a Visual Studio 2010 IDE window with a C++ source file. The code defines a function `impair` that checks if a number is odd, and a `main` function that uses `find_if` to find the first odd number in a list. A console window is open, showing the output `21` and a pause message.

```
#include<algorithm>
bool impair (int n)
{
    return n%2 ;
}

void main ()
{
    list<int> l ;
    list<int>::iterator it ;

    l.push_back(10);
    l.push_back(21);
    l.push_back(31);

    it = find_if (l.begin () , l.end () , impair) ;

    cout<<*it<<endl;
    system("PAUSE");
}
```

Console Output:

```
21
Appuyez sur une touche pour continuer...
```

Mme Wiem Yaiche Elleuch

plan

1. Fonctionnalités communes aux conteneurs
vector, list et deque
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack
et priority_queue

introduction

- les conteneurs se classent en deux catégories très différentes:
 - les conteneurs séquentiels
 - les conteneurs associatifs;
- les conteneurs séquentiels sont ordonnés suivant un ordre imposé explicitement par le programme lui-même,
- les conteneurs associatifs sont ordonnés de manière **intrinsèque**.
- Les trois conteneurs séquentiels principaux sont les classes *vector*, *list* et *deque*.
- La classe *vector* généralise la notion de tableau
- la classe *list* correspond à la notion de liste doublement chaînée
- *vector* dispose d'un itérateur à accès direct,
- *List* dispose d'un itérateur bidirectionnel.
- Classe deque: c'est une classe intermédiaire entre les deux précédentes dont la présence ne se justifie que pour des questions d'efficacité.

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Fonctionnalités communes aux conteneurs *vector*, *list* et *deque*

- Comme tous les conteneurs, *vector*, *list* et *deque* sont de taille dynamique, c'est-à-dire susceptibles de varier au fil de l'exécution.
- Malgré leur différence de nature, ces trois conteneurs possèdent des fonctionnalités communes. Elles concernent:
 - leur construction,
 - l'affectation globale,
 - leur comparaison,
 - l'insertion de nouveaux éléments ou la suppression d'éléments existants.

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque

1. Construction

- Construction d'un conteneur vide
- Construction avec un nombre donné d'éléments
- Construction avec un nombre donné d'éléments initialisés à une valeur
- Construction à partir d'une séquence
- Construction à partir d'un autre conteneur de même type

2. Modifications globales

3. Comparaison de conteneurs

4. Insertion/suppression d'éléments

2. Le conteneur vector

3. Le conteneur deque

4. Le conteneur list

5. Les adaptateurs de conteneur: queue, stack et priority_queue

exemple

```
void main ()
{
    list<float> l;
    // l est une liste vide
    // l.size() est égal à 0
    // l.begin()==l.end()
    system("PAUSE");
}
```

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque

1. Construction

- Construction d'un conteneur vide
- Construction avec un nombre donné d'éléments
- Construction avec un nombre donné d'éléments initialisés à une valeur
- Construction à partir d'une séquence
- Construction à partir d'un autre conteneur de même type

2. Modifications globales

3. Comparaison de conteneurs

4. Insertion/suppression d'éléments

2. Le conteneur vector

3. Le conteneur deque

4. Le conteneur list

5. Les adaptateurs de conteneur: queue, stack et priority_queue

construction

- construit un conteneur comprenant n éléments.
- l'initialisation de ces éléments:
 - dans le cas d'éléments de type standard (0 pour la classe statique, indéterminé sinon).
 - éléments de type classe: ils sont initialisés par appel d'un constructeur sans argument.

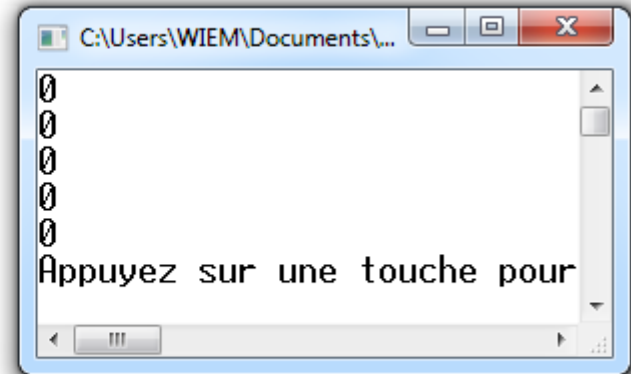
```

void main ()
{
    list<float> l(5);
    list<float>::iterator it;

    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}

```



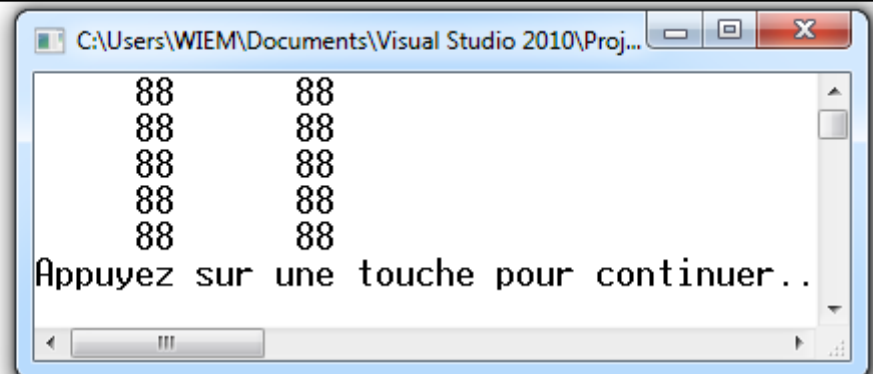
```

#include"myException.h"
#include<list>
#include<algorithm>
void main ()
{
    list<point> l(5);
    list<point>::iterator it;

    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}

```



plan

1. Fonctionnalités communes aux conteneurs vector, list et deque

1. Construction

- Construction d'un conteneur vide
- Construction avec un nombre donné d'éléments
- Construction avec un nombre donné d'éléments initialisés à une valeur
- Construction à partir d'une séquence
- Construction à partir d'un autre conteneur de même type

2. Modifications globales

3. Comparaison de conteneurs

4. Insertion/suppression d'éléments

2. Le conteneur vector

3. Le conteneur deque

4. Le conteneur list

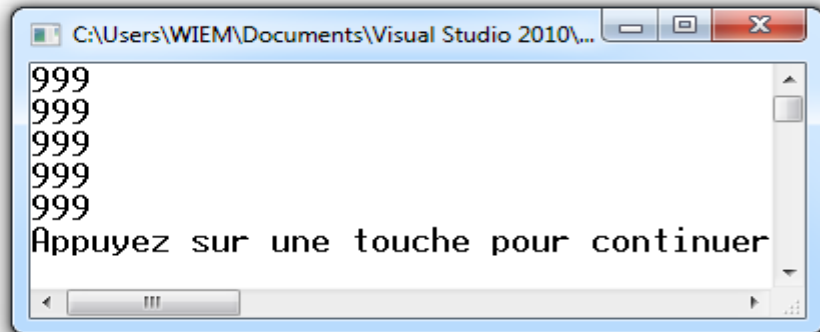
5. Les adaptateurs de conteneur: queue, stack et priority_queue

- Le premier argument du constructeur fournit le nombre d'éléments, le second argument en fournit la valeur.

```
#include "myException.h"
#include <list>
#include <algorithm>
void main ()
{
    list<int> l(5,999);
    list<int>::iterator it;

    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



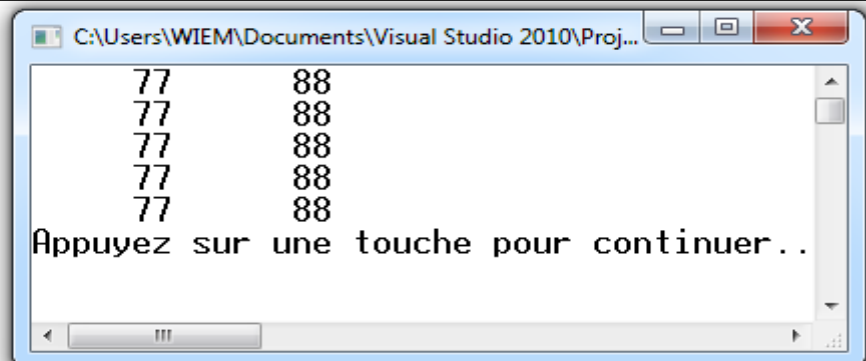
C:\Users\WIEM\Documents\Visual Studio 2010\...

999
999
999
999
999
Appuyez sur une touche pour continuer

```
#include "myException.h"
#include <list>
#include <algorithm>
void main ()
{
    point a(77,88);
    list<point> l(5,a);
    list<point>::iterator it;

    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



C:\Users\WIEM\Documents\Visual Studio 2010\Proj...

77 88
77 88
77 88
77 88
77 88
Appuyez sur une touche pour continuer..

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque

1. Construction

- Construction d'un conteneur vide
- Construction avec un nombre donné d'éléments
- Construction avec un nombre donné d'éléments initialisés à une valeur
- Construction à partir d'une séquence
- Construction à partir d'un autre conteneur de même type

2. Modifications globales

3. Comparaison de conteneurs

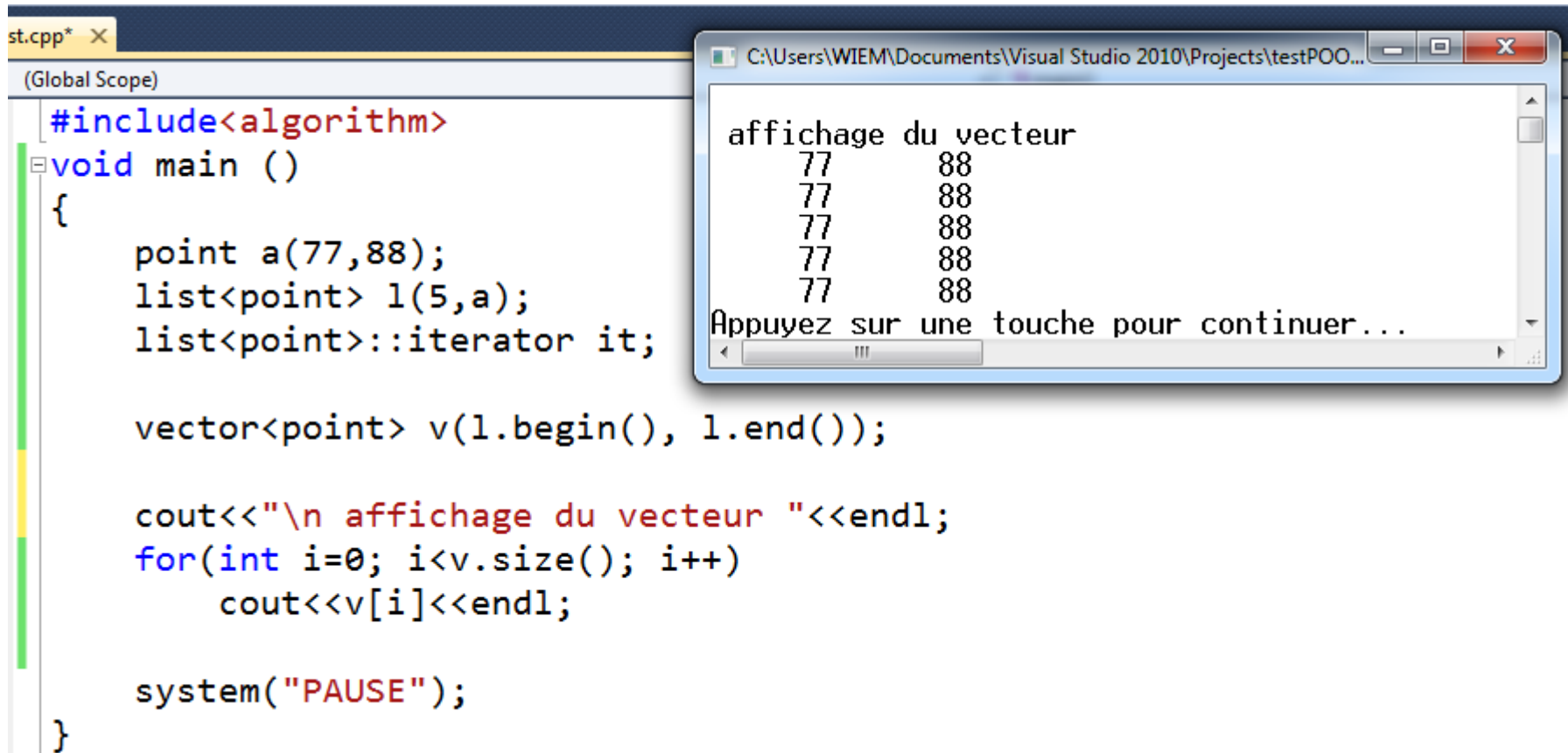
4. Insertion/suppression d'éléments

2. Le conteneur vector

3. Le conteneur deque

4. Le conteneur list

5. Les adaptateurs de conteneur: queue, stack et priority_queue



The image shows a Visual Studio 2010 IDE window with a C++ file named 'st.cpp'. The code defines a 'point' struct, a 'list' container, and a 'vector' of points. The main function creates a list with 5 points (all with x=77, y=88), constructs a vector from the list, and prints the vector's contents. A console window is open, showing the output: 'affichage du vecteur' followed by five lines of '77 88', and a prompt 'Appuyez sur une touche pour continuer...'. The code in the editor is as follows:

```
#include<algorithm>
void main ()
{
    point a(77,88);
    list<point> l(5,a);
    list<point>::iterator it;

    vector<point> v(l.begin(), l.end());

    cout<<"\n affichage du vecteur "<<endl;
    for(int i=0; i<v.size(); i++)
        cout<<v[i]<<endl;

    system("PAUSE");
}
```

- construit un vecteur de points en recopiant les points de la liste l;
- le constructeur par copie de point sera appelé pour chacun des points
- On peut construire un conteneur à partir d'une séquence d'éléments de même type.
- Dans ce cas, on fournit simplement au constructeur deux arguments représentant les bornes de l'intervalle correspondant.

Exemple 2

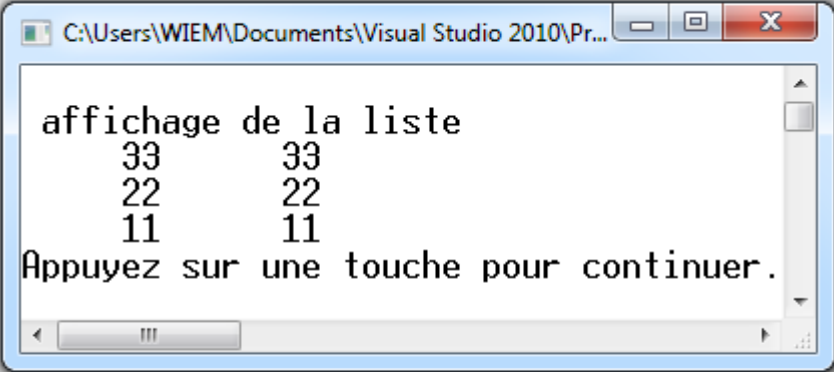
```
#include<algorithm>
void main ()
{
    point a(11,11);
    point b(22,22);
    point c(33,33);
    list<point> l;
    list<point>::iterator it;

    l.push_back(a); l.push_back(b); l.push_back(c);

    list<point> l2(l.rbegin(), l.rend());

    cout<<"\n affichage de la liste "<<endl;
    for(it=l2.begin(); it!=l2.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



- On peut utiliser des intervalles d'itérateurs

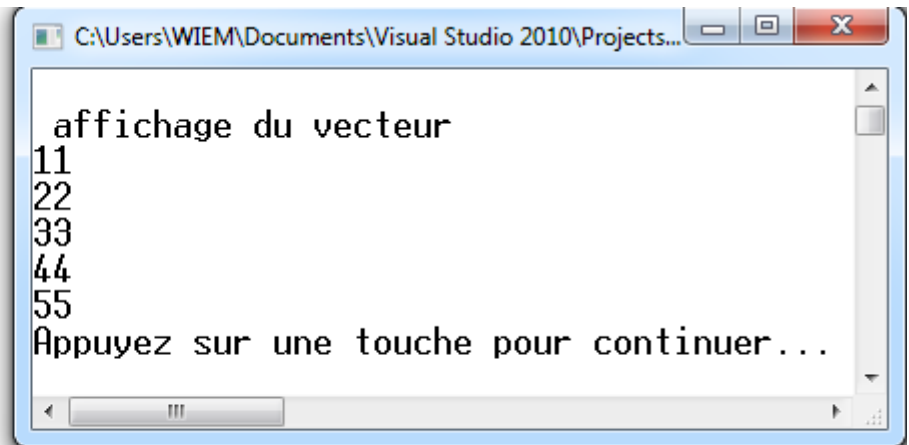
Exemple 3:

construction de conteneurs, à partir de séquences de valeurs issues d'un tableau classique, utilisant des intervalles définis par des pointeurs:

```
#include "parcVoiture.h"
#include "myException.h"
#include <list>
#include <algorithm>
void main ()
{
    int tab[5]={11,22,33,44,55};
    vector<int> v(tab, tab+5);

    cout<<"\n affichage du vecteur "<<endl;
    for(int i=0; i<v.size(); i++)
        cout<<v[i]<<endl;

    system("PAUSE");
}
```



plan

1. Fonctionnalités communes aux conteneurs vector, list et deque

1. Construction

- Construction d'un conteneur vide
- Construction avec un nombre donné d'éléments
- Construction avec un nombre donné d'éléments initialisés à une valeur
- Construction à partir d'une séquence
- Construction à partir d'un autre conteneur de même type

2. Modifications globales

3. Comparaison de conteneurs

4. Insertion/suppression d'éléments

2. Le conteneur vector

3. Le conteneur deque

4. Le conteneur list

5. Les adaptateurs de conteneur: queue, stack et priority_queue

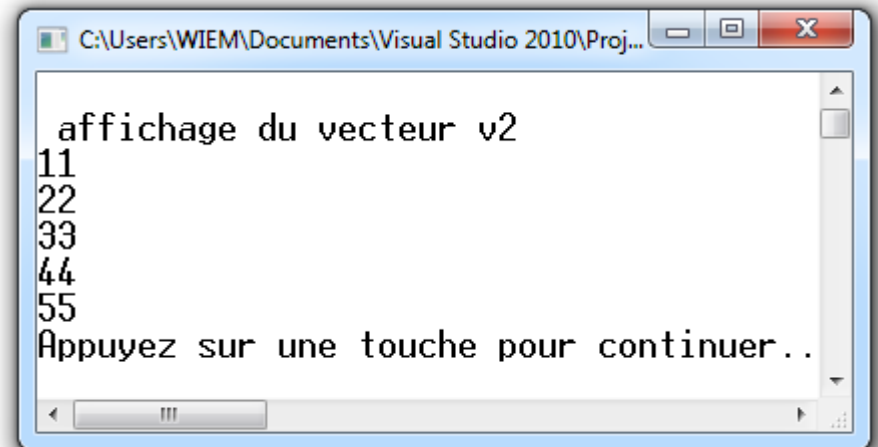
exemple

```
#include "myException.h"
#include <list>
#include <algorithm>
void main ()
{
    int tab[5]={11,22,33,44,55};
    vector<int> v(tab, tab+5);

    vector<int> v2(v);

    cout<<"\n affichage du vecteur v2"<<endl;
    for(int i=0; i<v2.size(); i++)
        cout<<v2[i]<<endl;

    system("PAUSE");
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\Proj...
affichage du vecteur v2
11
22
33
44
55
Appuyez sur une touche pour continuer..
```

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 1. Opérateur d'affectation
 2. La fonction membre assign
 3. La fonction clear
 4. La fonction swap
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Opérateur d'affectation

- On peut affecter un conteneur d'un type donné à un autre conteneur de même type, càd ayant le même nom de patron et le même type d'éléments.
- il n'est pas nécessaire que le nombre d'éléments de chacun des conteneurs soit le même.

(Global Scope)

```
#include<algorithm>
void main ()
{
    vector<int> v1;
    v1.push_back(11); v1.push_back(22);

    vector<int> v2;
    v2.push_back(33); v2.push_back(44), v2.push_back(55);

    v2=v1;

    cout<<"\n affichage du vecteur v2"<<endl;
    for(int i=0; i<v2.size(); i++)
        cout<<v2[i]<<endl;

    system("PAUSE");
}
```

C:\Users\WIEM\Documents\Visual Studio 2010\Pr...

affichage du vecteur v2
11
22
Appuyez sur une touche pour continuer.

- Remarque:
vector<float> v3;
v3=v1; // ERREUR
➔ Les éléments doivent être de même type

- Remarque 2:
list<int> v3;
v3=v1; // ERREUR
➔ Les conteneurs doivent être de même type

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 1. Opérateur d'affectation
 2. La fonction membre assign
 3. La fonction clear
 4. La fonction swap
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

la fonction *assign*

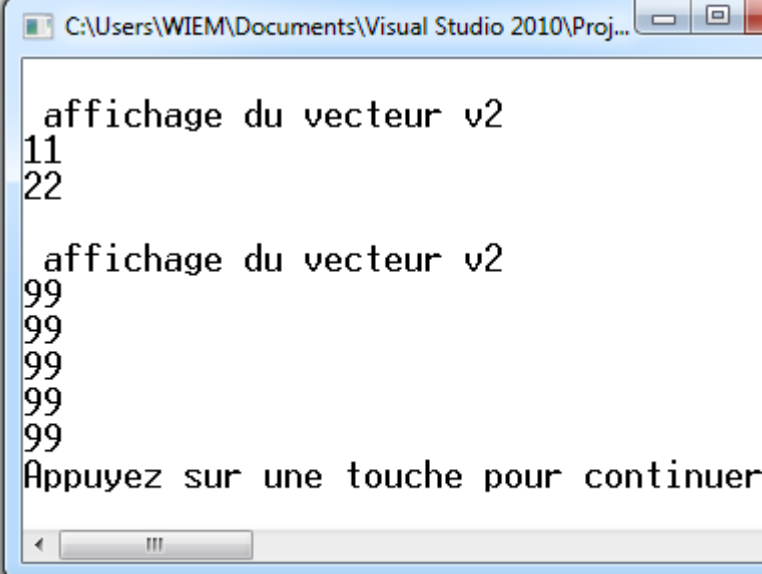
- la fonction *assign* permet d'affecter, à un conteneur, les éléments d'une séquence définie par un intervalle [*debut*, *fin*)
- *assign (debut, fin)*

Ou bien

- *assign (nb_fois, valeur)*

la fonction *assign*

```
void main ()  
{  
    list<int> v1;  
    v1.push_back(11); v1.push_back(22);  
  
    vector<int> v2;  
    v2.push_back(33); v2.push_back(44); v2.push_back(55);  
  
    v2.assign(v1.begin(), v1.end());  
  
    cout<<"\n affichage du vecteur v2"<<endl;  
    for(int i=0; i<v2.size(); i++)  
        cout<<v2[i]<<endl;  
  
    v2.assign(5,99);  
  
    cout<<"\n affichage du vecteur v2"<<endl;  
    for(int i=0; i<v2.size(); i++)  
        cout<<v2[i]<<endl;  
}
```



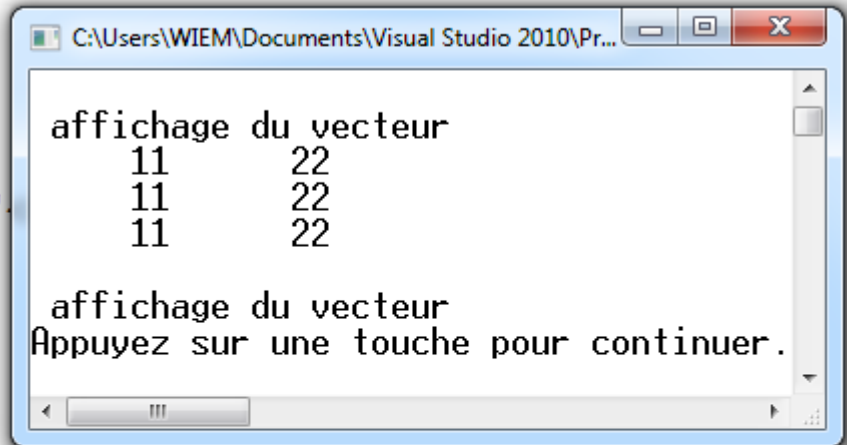
```
C:\Users\WIEM\Documents\Visual Studio 2010\Proj...  
affichage du vecteur v2  
11  
22  
  
affichage du vecteur v2  
99  
99  
99  
99  
99  
Appuyez sur une touche pour continuer
```

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 1. Opérateur d'affectation
 2. La fonction membre assign
 3. La fonction clear
 4. La fonction swap
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

- La fonction *clear()* vide le conteneur de son contenu

```
void main ()  
{  
    point a(11,22);  
    vector<point> v(3,a);  
  
    cout<<"\n affichage du vecteur "  
    for(int i=0; i<v.size(); i++)  
        cout<<v[i]<<endl;  
  
    v.clear();  
  
    cout<<"\n affichage du vecteur "<<endl;  
    for(int i=0; i<v.size(); i++)  
        cout<<v[i]<<endl;  
  
    system("PAUSE");  
}
```



- Dans cet exemple, il y a appel du destructeur pour chaque point

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 1. Opérateur d'affectation
 2. La fonction membre assign
 3. La fonction clear
 4. La fonction swap
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

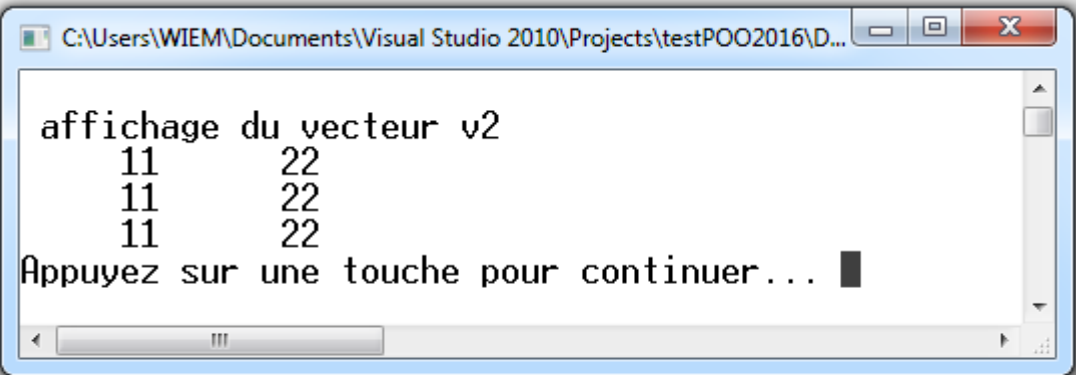
- La fonction membre *swap (conteneur)* permet d'échanger le contenu de deux conteneurs de même type

```
void main ()
{
    point a(11,22);
    point b(88,99);
    vector<point> v1(3,a);
    vector<point> v2(2,b);

    v1.swap(v2);

    cout<<"\n affichage du vecteur v2"<<endl;
    for(int i=0; i<v2.size(); i++)
        cout<<v2[i]<<endl;

    system("PAUSE");
}
```



La fonction swap est plus efficace que:

```
vector<point> v3=v1;  
v1=v2;  
v2=v3;
```

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 1. L'opérateur==
 2. L'opérateur <
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

opérateur ==

- Les trois conteneurs *vector*, *deque* et *list* disposent des opérateurs == et < ; ils disposent également de !=, <=, > et >=.
- Si *c1* et *c2* sont deux conteneurs de même type, leur comparaison par == sera vraie s'ils ont la même taille et si les éléments de même rang sont égaux.
- si les éléments concernés sont de type classe, il faut que l'opérateur == soit surchargé dans la classe.

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 1. L'opérateur==
 2. L'opérateur <
 4. Insertion/suppression d'éléments
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Opérateur <

- Il effectue une comparaison des éléments des deux conteneurs.
- Pour ce faire, il compare les éléments de même rang, par l'opérateur <, en commençant par les premiers, s'ils existent.
- Il s'interrompt dès que l'une des conditions suivantes est réalisée:
 - fin de l'un des conteneurs atteinte; le résultat de la comparaison est vrai,
 - comparaison de deux éléments fausse; le résultat de la comparaison des conteneurs est alors faux.
- Si un seul des deux conteneurs est vide, il apparaît comme < à l'autre.
- si les éléments concernés sont de type classe, il sera nécessaire que cette dernière dispose elle-même d'un opérateur < approprié.

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
 1. Insertion
 2. Suppression
 3. Insertion/suppression en fin
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

insertion

La fonction *insert* permet d'insérer:

1. une valeur avant une position donnée:

insert (position, valeur)

- insère *valeur* avant l'élément pointé par *position*
- fournit un itérateur sur l'élément après celui inséré

2. *n* fois une valeur donnée, avant une position donnée:

insert (position, nb_fois, valeur)

- Insère *nb_fois valeur*, avant l'élément pointé par *position*
- fournit un itérateur sur l'élément dernièrement inséré

3. les éléments d'un intervalle, à partir d'une position donnée:

insert (position, debut, fin)

- insère les valeurs de l'intervalle [*debut*, *fin*) avant l'élément pointé par *position*

Exemple (cas 1)

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);

    list<int>::iterator it;

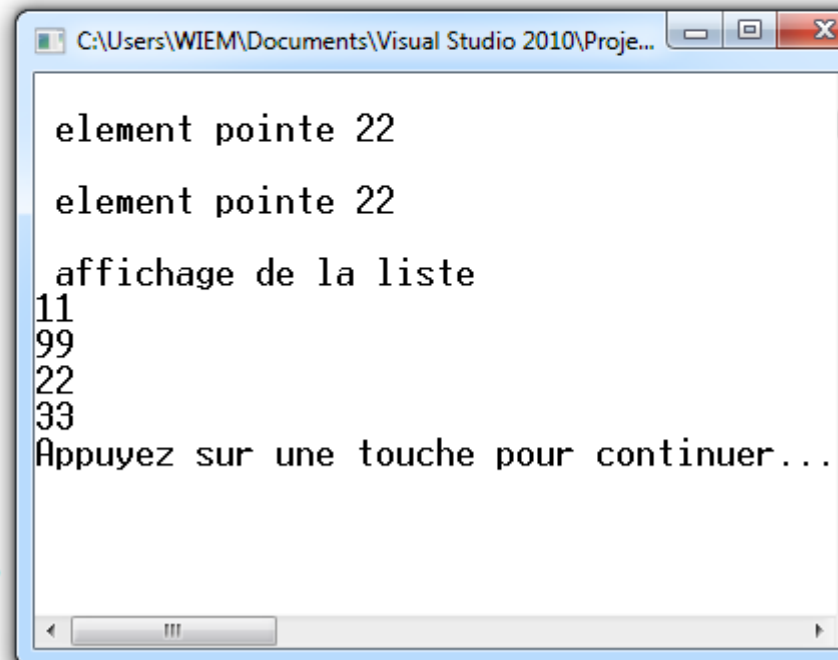
    it=l.begin();
    it++; // pointe sur le 22

    cout<<"\n element pointe "<<*it<<endl;

    l.insert(it,99);

    cout<<"\n element pointe "<<*it<<endl;

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\Proje...
element pointe 22
element pointe 22
affichage de la liste
11
99
22
33
Appuyez sur une touche pour continuer...
```

Exemple (cas 2)

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);

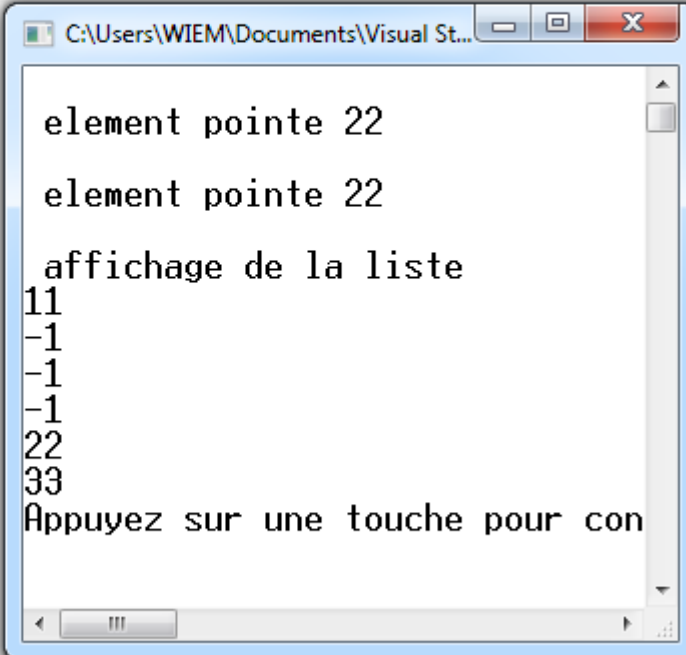
    list<int>::iterator it;

    it=l.begin();
    it++; // pointe sur le 22

    cout<<"\n element pointe "<<*it<<endl;

    1.insert(it,3,-1);
    // insère 3 fois la valeur -1
    cout<<"\n element pointe "<<*it<<endl;

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
}
```



```
C:\Users\WIEM\Documents\Visual St...
element pointe 22
element pointe 22
affichage de la liste
11
-1
-1
-1
-1
22
33
Appuyez sur une touche pour con
```

Exemple (cas 3)

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it;
    // l: 11 22 33

    vector<int> v;
    v.push_back(99); v.push_back(88); v.push_back(77);
    // v: 99 88 77

    it=l.end(); // itérateurs de fin consiste à pointer,
    // non pas sur le dernier élément d'un conteneur, mais juste après.

    l.insert(it,v.begin(),v.end() );

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
```

C:\Users\WIEM\Documents\Visual Studio 2010\

affichage de la liste

11
22
33
99
88
77

Appuyez sur une touche pour

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
 1. Insertion
 2. Suppression
 3. Insertion/suppression en fin
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

La fonction *erase*

La fonction *erase* permet de supprimer:

- un élément de position donnée:

erase (position)

- supprime l'élément désigné par *position*
- fournit un itérateur sur l'élément suivant ou sur la fin de la séquence

- les éléments d'un intervalle:

erase (debut, fin)

- supprime les valeurs de l'intervalle *[debut, fin)*
- fournit un itérateur sur l'élément suivant ou sur la fin de la séquence

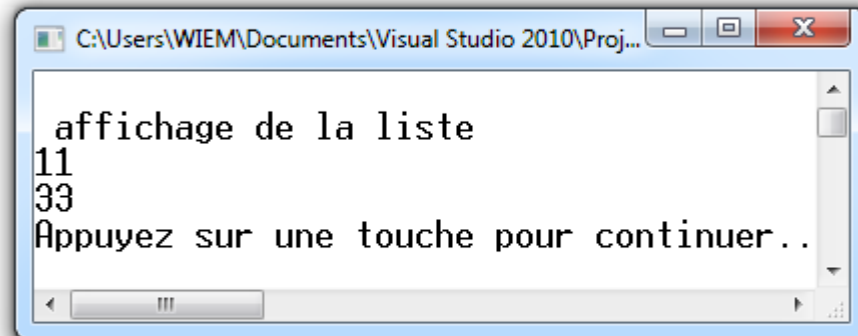
exemple

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it;
    // l: 11 22 33

    it=l.begin();
    it++; // pointe sur 22

    l.erase(it);

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



C:\Users\WIEM\Documents\Visual Studio 2010\Proj...

affichage de la liste
11
33
Appuyez sur une touche pour continuer..

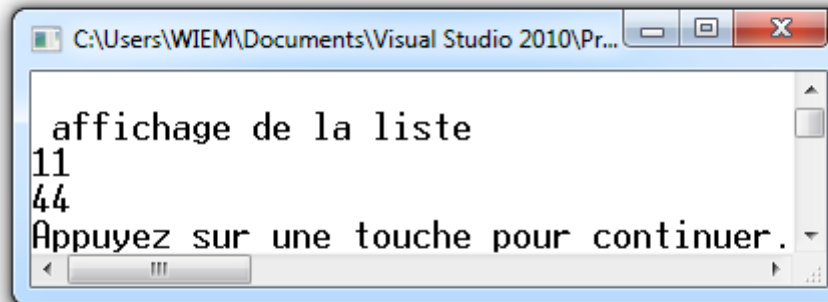
exemple

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33); l.push_back(44);
    list<int>::iterator it, it1, it2;
    // l: 11 22 33 44

    it1=l.begin(); it1++;
    it2=l.end(); it2--;

    l.erase(it1, it2);

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



plan

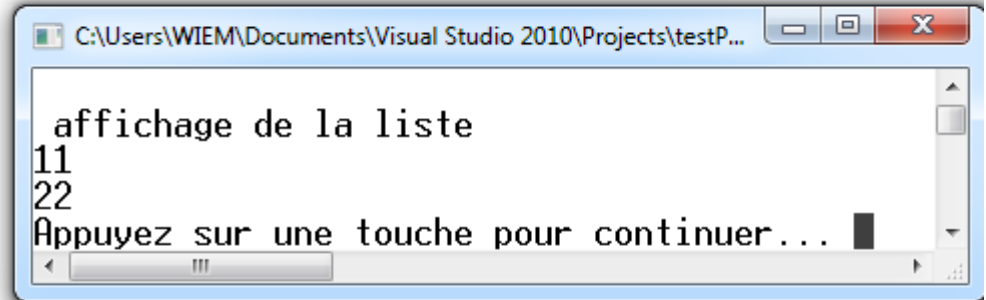
1. Fonctionnalités communes aux conteneurs vector, list et deque
 1. Construction
 2. Modifications globales
 3. Comparaison de conteneurs
 4. Insertion/suppression d'éléments
 1. Insertion
 2. Suppression
 3. Insertion/suppression en fin
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

pop_back

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it;
    // l: 11 22 33

    l.pop_back();

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



```
it=l.end(); // se positionner sur le suivant du dernier élément
it--; // se positionner sur le dernier élément
l.erase(it);
```



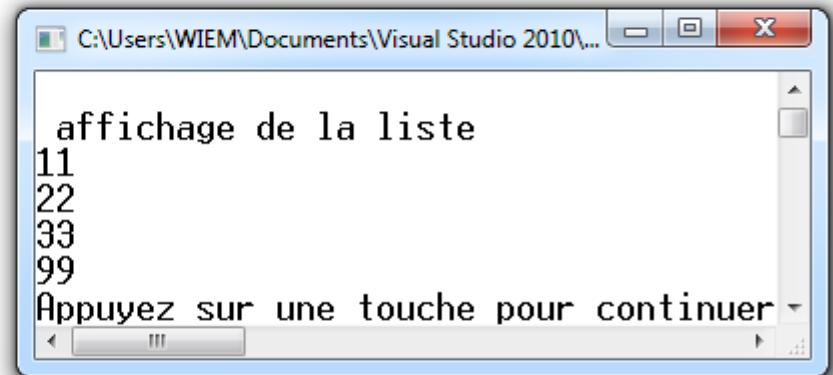
1.pop_back();

push_back

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it;
    // l: 11 22 33

    l.push_back(99);
    // ou bien
    // l.insert(l.end(),99);

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Le conteneur vector

- Il reprend la notion usuelle de tableau en autorisant un accès direct à un élément quelconque
- Cet accès peut se faire soit par le biais
 - d'un itérateur à accès direct,
 - par l'opérateur `[]`
 - ou par la fonction membre `at`.
 - l'accès au dernier élément peut se faire par une fonction membre *back*
- Mais il offre un cadre plus général que le tableau puisque:
 - la taille, c'est-à-dire le nombre d'éléments, peut varier au fil de l'exécution (comme celle de tous les conteneurs) ;
 - on peut effectuer toutes les opérations de construction, d'affectation et de comparaisons
 - on dispose des possibilités générales d'insertion ou de suppressions

Accès par itérateur

- Les itérateurs *iterator* et *reverse_iterator* d'un conteneur de type *vector* sont à accès direct.
- ***vector<int>:: iterator it;***
une expression telle que *it+i* désigne l'élément du vecteur *v*, situé *i* éléments plus loin que celui qui est désigné par *it* (à condition que la valeur de *i* soit compatible avec le nombre d'éléments de *v*).
- L'itérateur *it* peut, comme tout itérateur bidirectionnel, être incrémenté ou décrémenté par ++ ou --. Mais, comme il est à accès direct, il peut également être incrémenté ou décrémenté d'une quantité quelconque (*it+=3; it-=2;*

exemple

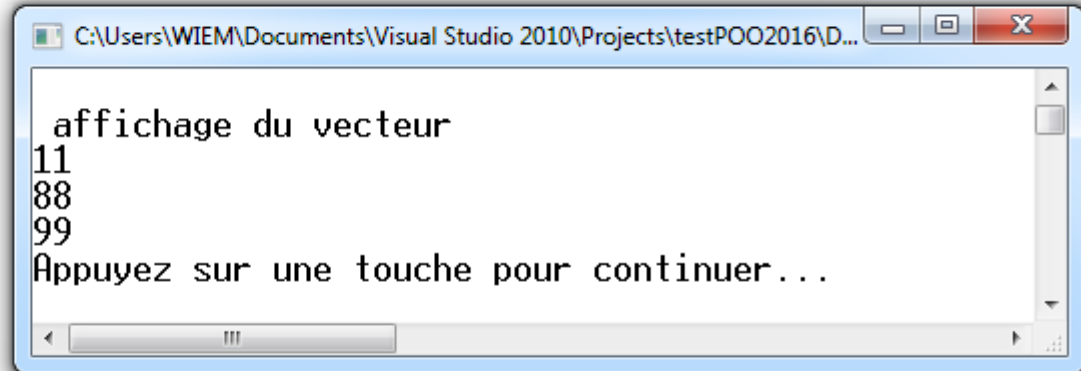
```
void main ()
{
    vector<int> v;
    v.push_back(11); v.push_back(22); v.push_back(33);
    vector<int>::iterator it=v.begin();
    // v: 11  22  33

    it+=2;
    *it=99;

    it=v.end()-2;
    *it=88;

    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



Accès par indice

- Si v est de type *vector*, l'expression $v[i]$ est une référence à l'élément de rang i .

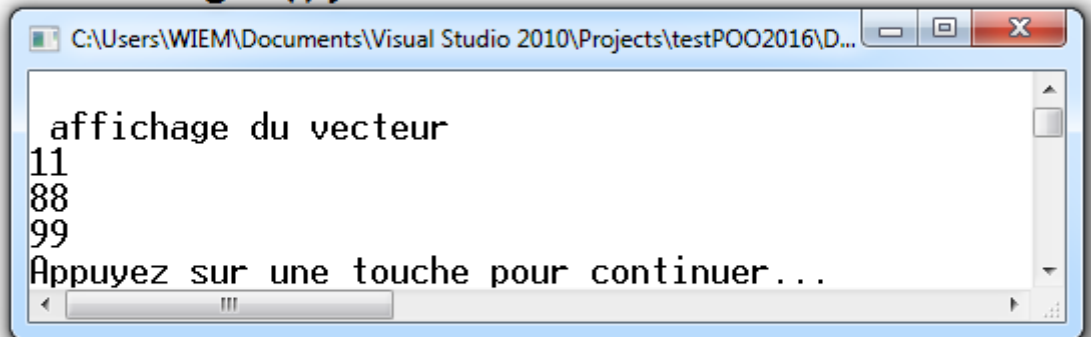
```
void main ()
{
    vector<int> v;
    v.push_back(11); v.push_back(22); v.push_back(33);
    vector<int>::iterator it=v.begin();
    // v: 11  22  33

    v[1]=88;

    *(v.begin()+2)= 99;

    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



La fonction membre at

- il existe également une fonction membre *at* telle que *v.at(i)* soit équivalente à *v[i]*.
- Sa seule raison d'être est de générer une exception *out_of_range* en cas d'indice incorrect, ce que l'opérateur *[]* ne fait pas.
- en contrepartie, *at* est légèrement moins rapide que l'opérateur *[]*.

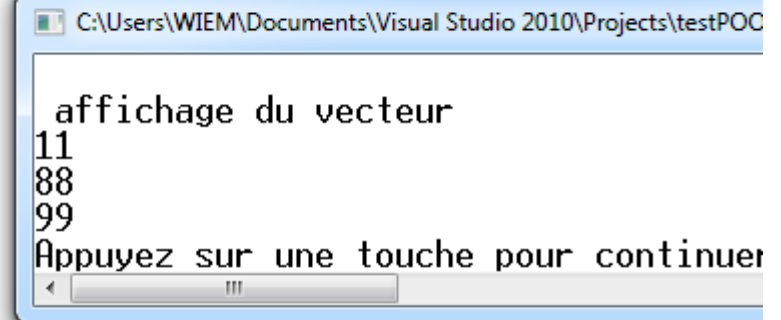
```
void main ()
{
    vector<int> v;
    v.push_back(11); v.push_back(22); v.push_back(33);
    vector<int>::iterator it=v.begin();
    // v: 11  22  33

    v.at(1)=88;    // v[1]=88;

    v.at(2)=99;    // *(v.begin()+2)= 99;

    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```

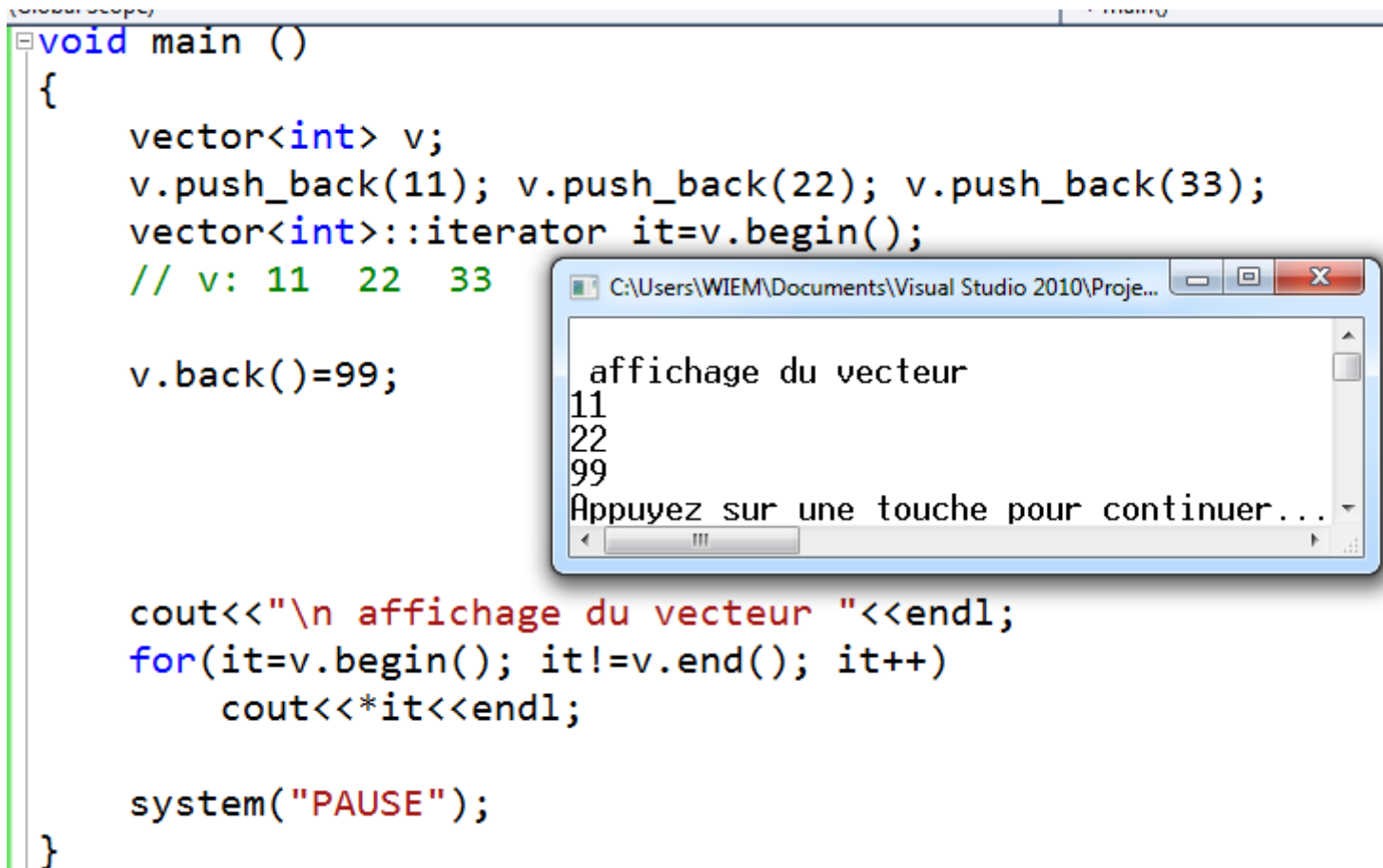


```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\testPOC

affichage du vecteur
11
88
99
Appuyez sur une touche pour continuer
```

Accès au dernier élément

- il existe une fonction membre *back* qui permet d'accéder directement au dernier élément.



The screenshot shows a Visual Studio 2010 IDE window with a C++ program. The code defines a `vector<int>` `v`, pushes back the values 11, 22, and 33, and then uses `v.back()` to set the last element to 99. The program prints the vector contents and pauses. An output window in the foreground displays the results: 'affichage du vecteur', '11', '22', '99', and a prompt to press a key to continue.

```
void main ()
{
    vector<int> v;
    v.push_back(11); v.push_back(22); v.push_back(33);
    vector<int>::iterator it=v.begin();
    // v: 11  22  33

    v.back()=99;

    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```

affichage du vecteur
11
22
99
Appuyez sur une touche pour continuer...

Gestion de l'emplacement mémoire

- la fonction *size()* permet de connaître le nombre d'éléments d'un vecteur.
- *capacity()*, fournit la taille potentielle du vecteur, càd le nombre d'éléments qu'il pourra accepter, sans avoir à effectuer de nouvelle allocation.
 - La différence *capacity()-size()* permet de connaître le nombre d'éléments qu'on pourra insérer dans un vecteur sans qu'une réallocation de mémoire soit nécessaire.
- *reserve(taille)* permet d'imposer la taille minimale de l'emplacement alloué à un vecteur à un moment donné
- *max_size()* permet de connaître la taille maximale qu'on peut allouer au vecteur, à un instant donné.
- *resize(taille)*, permet de modifier la taille effective du vecteur, aussi bien dans le sens de l'accroissement que dans celui de la réduction

La fonction membre flip

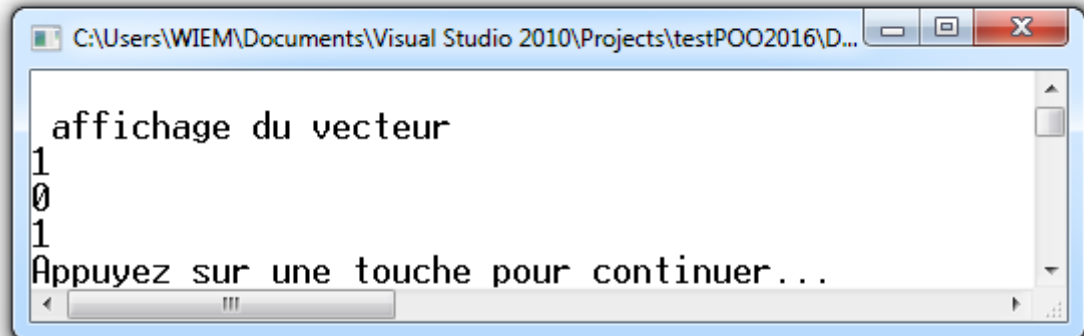
- lorsque l'argument du vector est de type *bool*, il existe une fonction membre *flip()*, destinée à inverser tous les bits d'un tel vecteur

```
void main ()
{
    vector<bool> v;
    v.push_back(0); v.push_back(1); v.push_back(0);
    vector<bool>::iterator it=v.begin();
    // v: 0 1 0

    v.flip();

    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<endl;

    system("PAUSE");
}
```



plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Le conteneur deque

- les fonctionnalités de *deque* sont celles de *vector*, auxquelles sont rajoutées les fonctions spécialisées concernant le premier élément:
- *front()* pour accéder au premier élément; elle complète la fonction *back* permettant l'accès au dernier élément;
- *push_front(valeur)*, pour insérer un nouvel élément en début; elle complète la fonction *push_back()*;
- *pop_front()*, pour supprimer le premier élément; elle complète la fonction *pop_back()*.

Différence entre deque et vector

- L'espace mémoire alloué pour un conteneur de type deque (allocation de plusieurs bloc) est différent de celui d'un vector (allocation d'un seul bloc).
- *Ce qui implique que certaines opérations sont plus rapides sur deque que sur vector et inversement.*

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

Le conteneur *list*

- Le conteneur *list* correspond au concept de liste doublement chaînée,
- Il dispose d'un itérateur bidirectionnel permettant de parcourir la liste à l'endroit ou à l'envers.
- En contrepartie, le conteneur *list* ne dispose plus d'un itérateur à accès direct.

Accès aux éléments existants

- Les conteneurs *vector* et *deque* permettaient d'accéder aux éléments existants de deux manières : par itérateur ou par indice; (les itérateurs de ces classes étaient à accès direct)
- Le conteneur *list* offre toujours les itérateurs *iterator* et *reverse_iterator* mais ils sont seulement bidirectionnels.
- Si *it* désigne un tel itérateur, on pourra toujours consulter l'élément pointé par la valeur de l'expression **it*, ou le modifier par une affectation.

Accès aux éléments existants

- L'itérateur *it* pourra être incrémenté par ++ ou --, mais il ne sera plus possible de l'incrémenter d'une quantité quelconque. (*it+=2; it-=3; //* ERREUR)
- Ainsi, pour accéder une première fois à un élément d'une liste, il faut obligatoirement la parcourir depuis son début ou depuis sa fin, élément par élément, jusqu'à l'élément concerné.
- La classe *list* dispose des fonctions *front()* et *back()*, : la première est une référence au premier élément, la seconde est une référence au dernier élément:

exemple

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it=l.begin();
    // l: 11  22  33

    it++; // correct
    *it=99; // l: 11  99  33

    it--; // correct
    *it=88; // 88  99  33

    // it+=2; // ERREUR

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```

C:\Users\WIEM\Documents\Visual Studio 2010\Projects\t...

affichage de la liste

88

99

33

Appuyez sur une touche pour continuer...

Insertions et suppressions

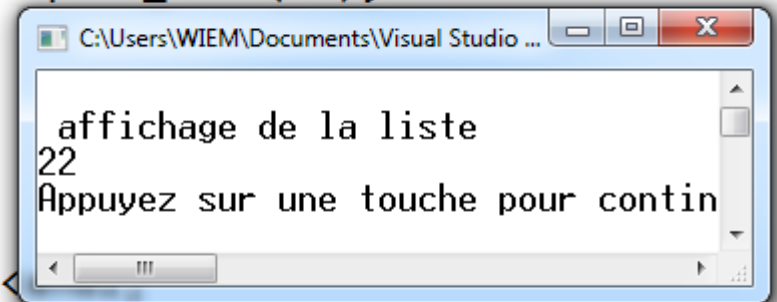
- Le conteneur *list* dispose des possibilités générales d'insertion et de suppression procurées par les fonctions *insert* et *erase*
- On dispose également des fonctions spécialisées d'insertion en début *push_front(valeur)* ou en fin *push_back(valeur)* ou de suppression en début *pop_front()* ou en fin *pop_back()*,
- En outre, la classe *list* dispose de fonctions de suppressions conditionnelles (que ne possède pas *vector* et *deque*)
 - suppression de tous les éléments ayant une valeur donnée,
remove(valeur); // supprimer tous les éléments égaux à valeur
// ➔ surcharger operator== pour les classes
 - suppression des éléments satisfaisant à une condition donnée.
remove_if(prédicat); // supprimer tous les éléments répondant au prédicat

remove

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(11);
    list<int>::iterator it=l.begin();
    // l: 11 22 11

    l.remove(11);

    cout<<"\n affichage de la liste "<<
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



remove_if - prédicat

```
bool estPair(int n) // estPair: prédicat
{
    if (n%2==0) return 1;
    return 0;
}

void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(44);
    list<int>::iterator it=l.begin();
    // l: 11 22 44

    l.remove_if(estPair);

    cout<<"\n affichage de la liste "
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```

C:\Users\WIEM\Documents\Visual Studi...
affichage de la liste
11
Appuyez sur une touche pour conti

Opérations globales

- En plus des possibilités générales offertes par l'affectation et la fonction membre *assign*, la classe *list* offre d'autres:
 - tri de ses éléments avec suppression des occurrences multiples,
 - fusion de deux listes préalablement ordonnées,
 - transfert de tout ou partie d'une liste dans une autre liste de même type.

Opérations globales: tri d'une liste

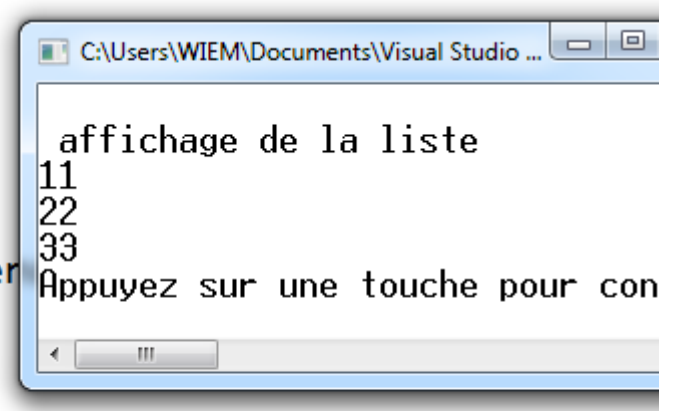
- Il existe des algorithmes de tri des éléments d'un conteneur, mais la plupart nécessitent des itérateurs à accès direct.
- la classe *list* dispose de sa propre fonction *sort*,
 - *sort()*: trie la liste en s'appuyant sur *operator<*
 - *sort(predicat)*: trie la liste en s'appuyant sur le prédicat binaire *predicat*

Sort: tri d'une liste d'entiers

```
void main ()
{
    list<int> l;
    l.push_back(33); l.push_back(11); l.push_back(22);
    list<int>::iterator it=l.begin();
    // l: 33 11 22

    l.sort();

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```

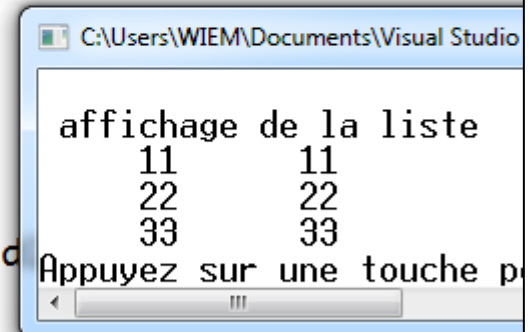


Tri d'une liste de points

```
void main ()
{
    list<point> l;
    point a(33,33);
    point b(11,11);
    point c(22,22);
    l.push_back(a); l.push_back(b); l.push_back(c);
    list<point>::iterator it=l.begin();
    // l: 33 33 11 11 22 22

    l.sort();

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



C:\Users\WIEM\Documents\Visual Studio

affichage de la liste

11	11
22	22
33	33

Appuyez sur une touche p

```
bool point::operator<(point& pt)
{
    if (x<pt.x && y<pt.y) return 1;
    return 0;
}
```

Opérations globales: Suppression des éléments en double

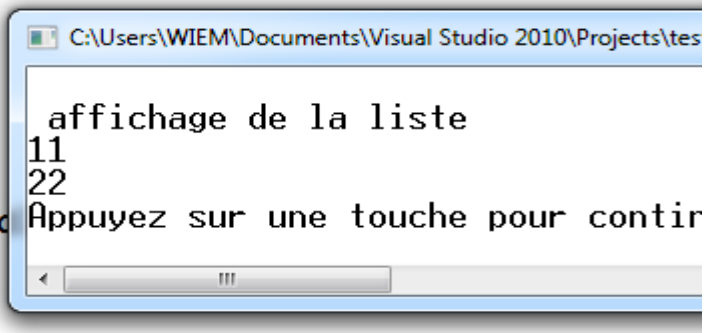
- La fonction *unique* permet d'éliminer les éléments en double, à condition de la faire porter sur une liste préalablement triée.
- Dans le cas contraire, elle peut fonctionner mais, alors, elle se contente de remplacer par un seul élément, les séquences de valeurs consécutives identiques, ce qui signifie que, la liste pourra encore contenir des valeurs identiques, mais non consécutives
- `operator==` doit être surchargé pour les classes.
- *unique()* : ne conserve que le premier élément d'une suite de valeurs consécutives égales (==)
- *unique(predicat)* : ne conserve que le premier élément d'une suite de valeurs consécutives satisfaisant au prédicat binaire *predicat*

exemple

```
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(11); l.push_back(22);
    list<int>::iterator it=l.begin();
    // l: 11 11 22

    l.unique();

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    system("PAUSE");
}
```



Opérations globales: fusion de 2 listes

- Bien qu'il existe un algorithme général de fusion pouvant s'appliquer à deux conteneurs triés, la classe *list* dispose d'une fonction membre spécialisée généralement plus performante
- La fonction membre *merge* permet de venir fusionner une autre liste de même type avec la liste concernée. La liste fusionnée est vidée de son contenu.
- la fonction *merge* s'appuie, comme *sort*, sur une relation d'ordre l'opérateur $<$.
- *merge (liste)*: fusionne *liste* avec la liste concernée, en s'appuyant sur l'opérateur $>$; à la fin: *liste* est vide
- *merge (liste, predicat)*: fusionne *liste* avec la liste concernée, en s'appuyant sur le prédicat binaire *predicat*

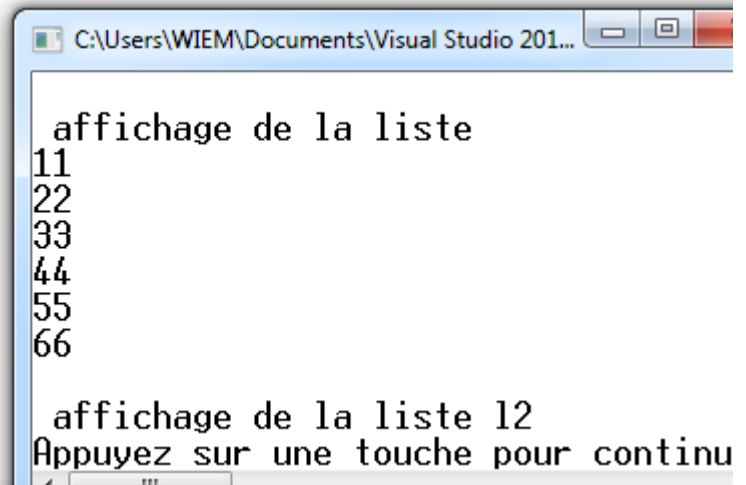
Opérations globales: fusion de 2 listes

```
void main ()
{
    list<int> l1;
    l1.push_back(11); l1.push_back(33); l1.push_back(55);
    list<int>::iterator it=l1.begin();
    // l1: 11 33 55

    list<int> l2;
    l2.push_back(22); l2.push_back(44); l2.push_back(66);
    //l2: 22 44 66

    l1.merge(l2);

    cout<<"\n affichage de la liste 1"<<endl;
    for(it=l1.begin(); it!=l1.end(); it++)
        cout<<*it<<endl;
    cout<<"\n affichage de la liste 2"<<endl;
    for(it=l2.begin(); it!=l2.end(); it++)
        cout<<*it<<endl;
}
```



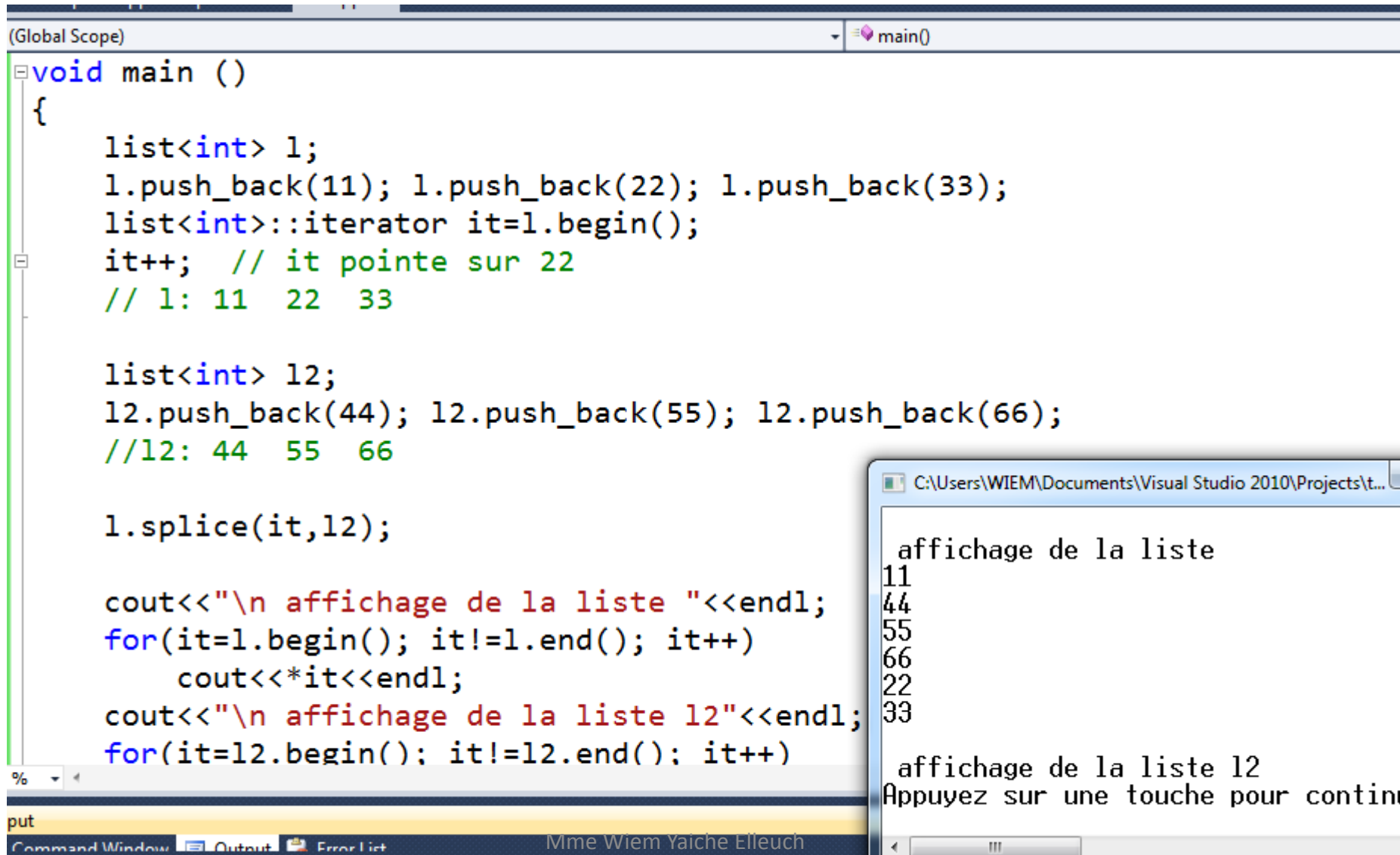
```
C:\Users\WIEM\Documents\Visual Studio 2010\...
affichage de la liste
11
22
33
44
55
66

affichage de la liste l2
Appuyez sur une touche pour continuer
```

Opérations globales: Transfert d'une partie de liste dans une autre

- La fonction *splice* permet de déplacer des éléments d'une autre liste dans la liste concernée.
- comme avec *merge*, les éléments déplacés sont supprimés de la liste d'origine et pas seulement copiés
 - *splice(position, liste_or)*: déplace les éléments de *liste_or* à l'emplacement *position*
 - *splice (position, liste_or, position_or)*: déplace l'élément de *liste_or* pointé par *position_or* à l'emplacement *position*
 - *splice (position, liste_or, debut_or, fin_or)*: déplace l'intervalle *[debut_or, fin_or)* de *liste_or* à l'emplacement *position*

exemple



The screenshot displays the Visual Studio 2010 IDE interface. The main window shows a C++ source file with the following code:

```
(Global Scope) main()
void main ()
{
    list<int> l;
    l.push_back(11); l.push_back(22); l.push_back(33);
    list<int>::iterator it=l.begin();
    it++; // it pointe sur 22
    // l: 11 22 33

    list<int> l2;
    l2.push_back(44); l2.push_back(55); l2.push_back(66);
    //l2: 44 55 66

    l.splice(it,l2);

    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<endl;
    cout<<"\n affichage de la liste l2"<<endl;
    for(it=l2.begin(); it!=l2.end(); it++)
        cout<<*it<<endl;
}
```

The bottom of the IDE shows the Command Window and Output window. The Command Window contains the text "put". The Output window shows the execution results:

```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\t...
affichage de la liste
11
44
55
66
22
33

affichage de la liste l2
Appuyez sur une touche pour continuer
```

The status bar at the bottom indicates the file is named "Mme Wiem Yaiche Elleuch".

plan

1. Fonctionnalités communes aux conteneurs vector, list et deque
2. Le conteneur vector
3. Le conteneur deque
4. Le conteneur list
5. Les adaptateurs de conteneur: queue, stack et priority_queue

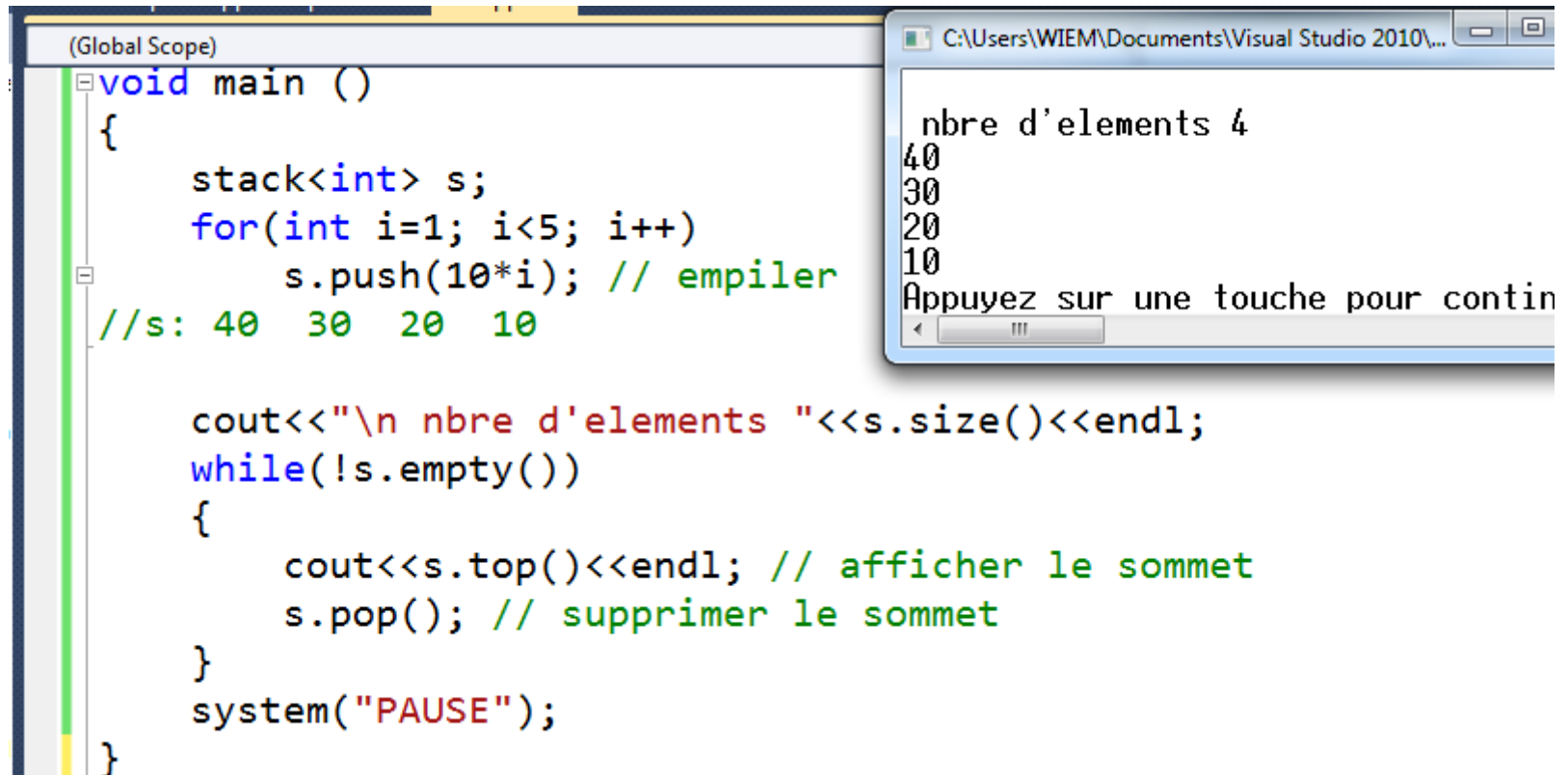
Les adaptateurs de conteneur

- La bibliothèque standard dispose de trois patrons particuliers *stack*, *queue* et *priority_queue*, dits adaptateurs de conteneurs.
- Ils disposent tous d'un constructeur sans argument.

L'adaptateur stack

- Le patron *stack* est destiné à la gestion de piles de type *LIFO* (Last In, First Out) ;
- Dans un tel conteneur, on ne peut qu'introduire (*push*) des informations qu'on empile les unes sur les autres et qu'on recueille, à raison d'une seule à la fois, en extrayant la dernière introduite. On y trouve uniquement les fonctions membres suivantes:
 - *empty()* : fournit *true* si la pile est vide,
 - *size()* : fournit le nombre d'éléments de la pile,
 - *top()* : accès à l'information située au sommet de la pile qu'on peut connaître ou modifier (sans la supprimer),
 - *push (valeur)* : place *valeur* sur la pile,
 - *pop()* : fournit la valeur de l'élément situé au sommet, en le supprimant de la pile.

exemple



The image shows a Visual Studio 2010 IDE window with a C++ program. The code defines a stack, pushes elements 10, 20, 30, and 40, and then prints the size and pops the elements. An output window on the right shows the execution results.

```
(Global Scope)
void main ()
{
    stack<int> s;
    for(int i=1; i<5; i++)
        s.push(10*i); // empiler
    //s: 40 30 20 10

    cout<<"\n nbre d'elements "<<s.size()<<endl;
    while(!s.empty())
    {
        cout<<s.top()<<endl; // afficher le sommet
        s.pop(); // supprimer le sommet
    }
    system("PAUSE");
}
```

Output Window (C:\Users\WIEM\Documents\Visual Studio 2010\...):

```
nbre d'elements 4
40
30
20
10
Appuyez sur une touche pour continuer
```

L'adaptateur queue

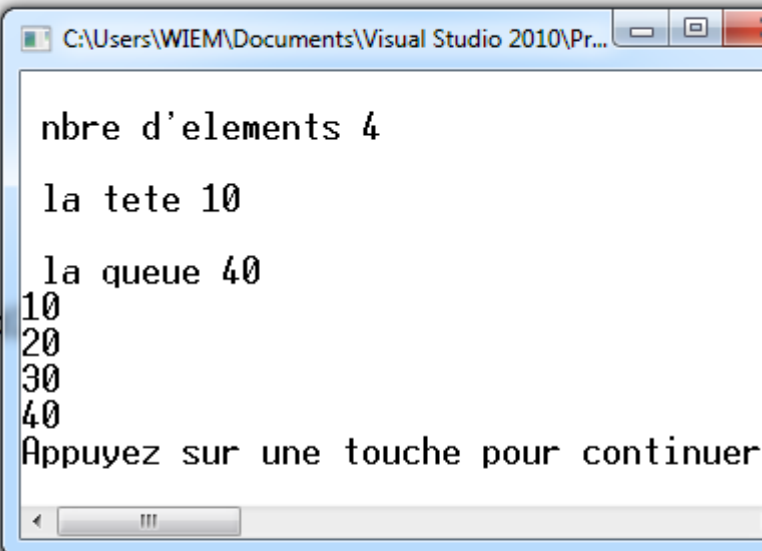
- Le patron *queue* est destiné à la gestion de files d'attentes, dites aussi queues, ou encore piles de type *FIFO* (First In, First Out).
- On y place des informations qu'on introduit en fin et qu'on recueille en tête, dans l'ordre inverse de leur introduction.
- On y trouve uniquement les fonctions membres suivantes:
 - *empty()*: fournit *true* si la queue est vide,
 - *size()*: fournit le nombre d'éléments de la queue,
 - *front()* : accès à l'information située en tête de la queue, qu'on peut ainsi connaître ou modifier, sans la supprimer,
 - *back()* : accès à l'information située en fin de la queue, qu'on peut ainsi connaître ou modifier, sans la supprimer,
 - *push (valeur)* : place *valeur* dans la queue,
 - *pop()* : fournit l'élément situé en tête de la queue en le supprimant.

exemple

```
#include<queue>
void main ()
{
    queue<int> q;
    for(int i=1; i<5; i++)
        q.push(10*i); // empiler
    //q: 10 20 30 40

    cout<<"\n nbre d'elements "<<q.size()<<endl;
    cout<<"\n la tete "<<q.front()<<endl;
    cout<<"\n la queue "<<q.back()<<endl;

    while(!q.empty())
    {
        cout<<q.front()<<endl; // afficher le sommet
        q.pop(); // supprimer le sommet
    }
    system("PAUSE");
}
```



C:\Users\WIEM\Documents\Visual Studio 2010\Pr...

nbre d'elements 4

la tete 10

la queue 40

10

20

30

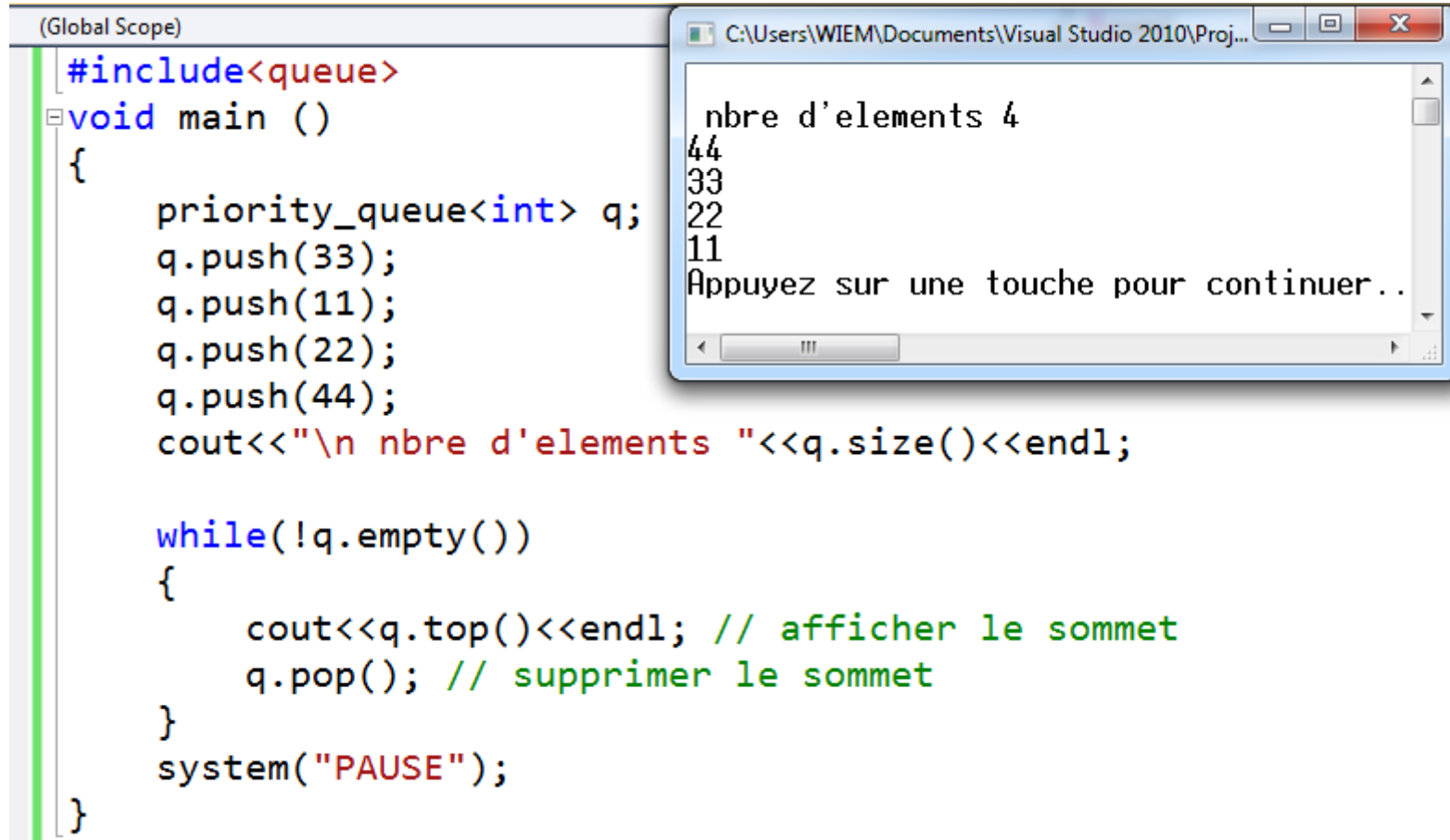
40

Appuyez sur une touche pour continuer

L'adaptateur `priority_queue`

- Un tel conteneur ressemble à une file d'attente, dans laquelle on introduit toujours des éléments en fin
- l'emplacement des éléments dans la queue est modifié à chaque introduction, de manière à respecter une certaine priorité définie par une relation d'ordre fournie sous forme d'un prédicat binaire.
- On parle parfois de file d'attente avec priorités.
- On y trouve uniquement les fonctions membres suivantes:
 - *empty()*: fournit *true* si la queue est vide;
 - *size()* : fournit le nombre d'éléments de la queue;
 - *push (valeur)* : place *valeur* dans la queue;
 - *top()*: accès à l'information située en tête de la queue qu'on peut connaître ou, théoriquement modifier (sans la supprimer)
 - *pop()* : fournit l'élément situé en tête de la queue en le supprimant.

exemple



The image shows a C++ program in a code editor and its execution output in a console window. The code defines a priority queue and pushes elements 33, 11, 22, and 44. It then prints the size of the queue (4) and enters a loop that prints and removes the top element. The console output shows the size 4 and the elements 44, 33, 22, and 11 in descending order.

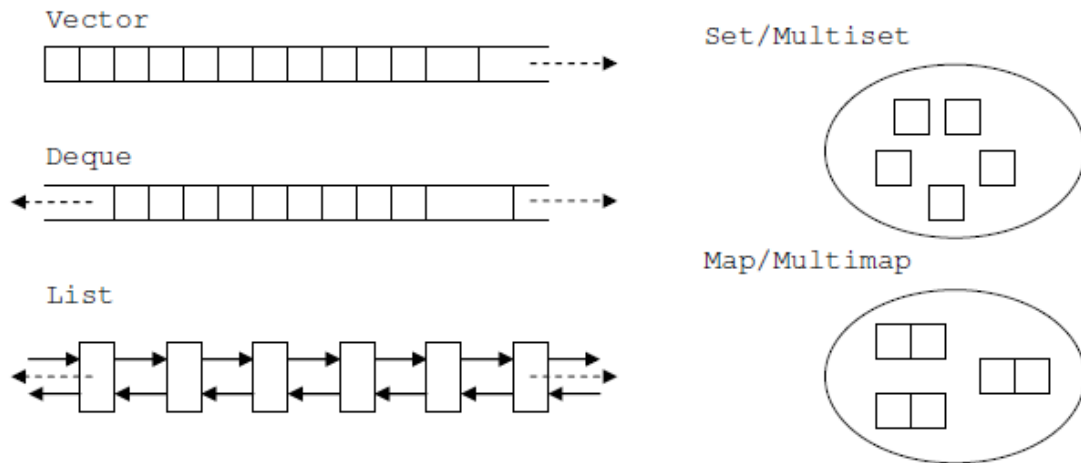
```
(Global Scope)
#include<queue>
void main ()
{
    priority_queue<int> q;
    q.push(33);
    q.push(11);
    q.push(22);
    q.push(44);
    cout<<"\n nbre d'elements "<<q.size()<<endl;

    while(!q.empty())
    {
        cout<<q.top()<<endl; // afficher le sommet
        q.pop(); // supprimer le sommet
    }
    system("PAUSE");
}
```

C:\Users\WIEM\Documents\Visual Studio 2010\Proj...
nbre d'elements 4
44
33
22
11
Appuyez sur une touche pour continuer..

Les conteneurs associatifs

- Il y a 2 catégories de conteneurs :
 - Séquentiels** : l'accès est linéaire, c'est l'ordre qui compte.



- Associatifs** : l'accès par clés.

Les conteneurs associatifs

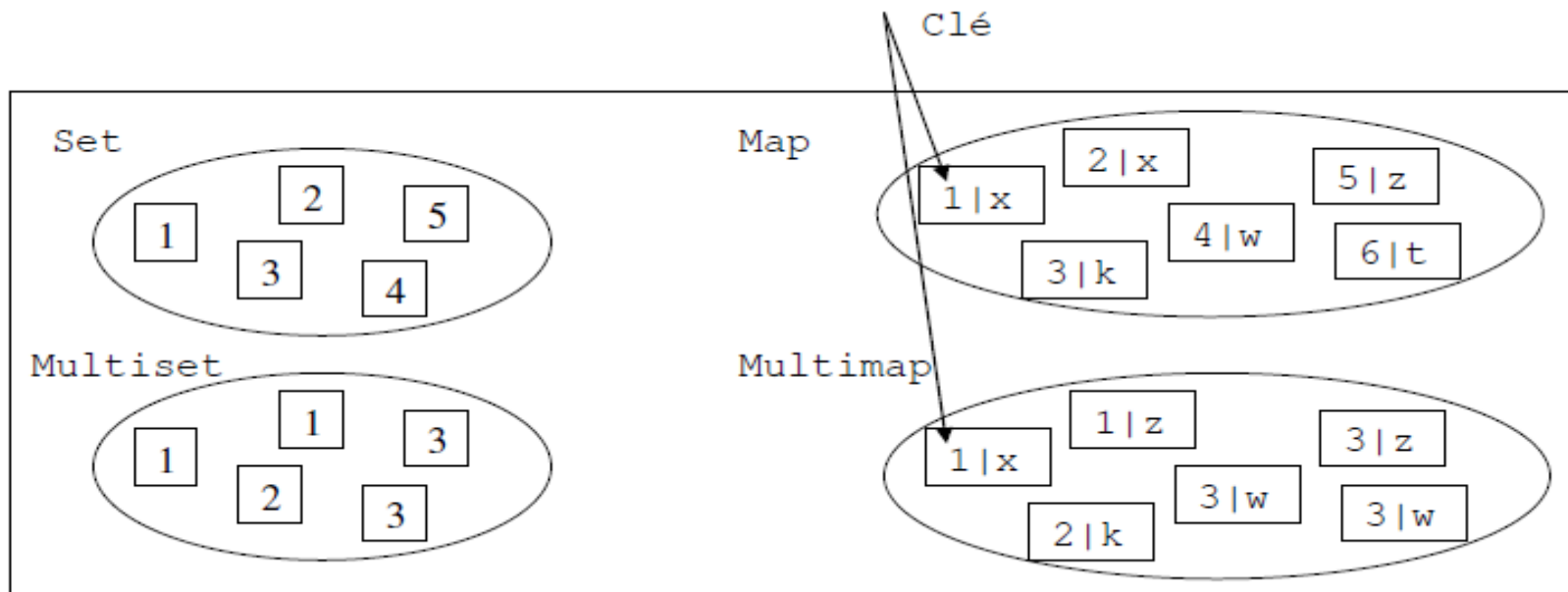
- les conteneurs se classent en deux catégories:
 - les conteneurs séquentiels
 - les conteneurs associatifs.
- Les conteneurs séquentiels sont ordonnés suivant un ordre imposé explicitement par le programme lui-même
- Les conteneurs associatifs ont pour principale vocation de retrouver une valeur (information), en fonction d'une clé. (et non plus en fonction de sa place dans le conteneur)
- exemple du répertoire téléphonique: la clé: nom de la personne, la valeur: numéro de téléphone
- pour de questions d'efficacité, un conteneur associatif se trouve ordonné intrinsèquement en permanence, en se fondant sur une relation (par défaut <) choisie à la construction.

Les conteneurs associatifs

- Les deux conteneurs associatifs les plus importants sont *map* et *multimap*.
- *map* impose l'unicité des clés, (absence de deux éléments ayant la même clé), *multimap* ne l'impose pas (plusieurs éléments de même clé apparaissent **consécutivement**).
- Cette distinction permet précisément de redéfinir l'opérateur [] sur un conteneur de type *map*.

Les conteneurs associatifs

- Il existe deux autres conteneurs qui correspondent à des cas particuliers de *map* et *multimap*, dans le cas où la valeur associée à la clé n'existe plus, (les éléments se limitent à la seule clé).
- Ces conteneurs se nomment *set* et *multiset* (*ensemble et multiEnsemble*)



- Le conteneur map

Conteneur map

- Jusqu'à maintenant, l'accéder aux éléments d'un conteneur se fait en utilisant les crochets[].
- Dans un vector ou une deque, les éléments sont accessibles *via* leur index, un nombre entier positif.
- Les tables associatives sont des structures de données qui autorisent l'emploi de n'importe quel type comme index.
- Map: quand on a besoin d'utiliser autre chose que des entiers pour indexer les éléments.

Conteneur map

- La bibliothèque standard distingue deux types de conteneurs associatifs : les conteneurs qui différencient la valeur de la clef de la valeur de l'objet lui-même et les conteneurs qui considèrent que les objets sont leur propre clé.
- Les conteneurs de la première catégorie constituent ce que l'on appelle des *associations* car ils permettent d'associer des clés aux valeurs des objets.
- Les conteneurs associatifs de la deuxième catégorie sont appelés quant à eux des *ensembles*, en raison du fait qu'ils servent **généralement à indiquer si un objet fait partie ou non d'un ensemble d'objets.**

Le conteneur map

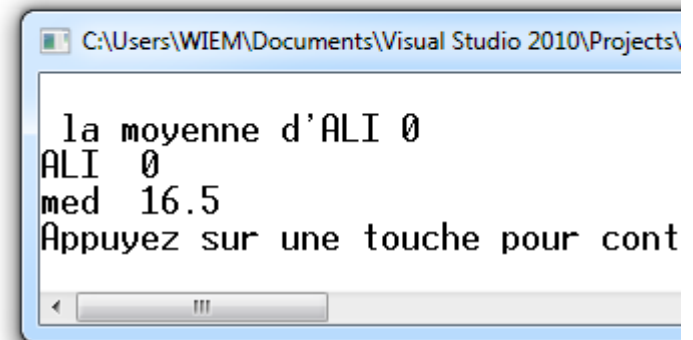
- Le conteneur *map* est formé d'éléments composés de deux parties: une clé et une valeur.
- Pour représenter de tels éléments, il existe un **patron de classe** approprié, nommé *pair*, paramétré par le type de la clé et par celui de la valeur.
- Le patron de classe *pair* possède deux attributs publics:
 - *first* correspondant à la clé,
 - *second* correspondant à la valeur associée.
- Un conteneur *map* permet d'accéder rapidement à la valeur associée à une clé en utilisant l'opérateur [] ;

exemple

```
#include<algorithm>
#include<stack>
#include<queue>
#include<map>
void main ()
{
    map<string, float> m;    // nom et moyenne
    map<string, float>::iterator it;
    m["med"]=16.5;

    cout<<"\n la moyenne d'ALI "<<m["ALI"]<<endl;

    for(it=m.begin(); it!=m.end(); it++)
        cout<<(*it).first<<"  "<<(*it).second<<endl;
    system("PAUSE");
}
```



remarques

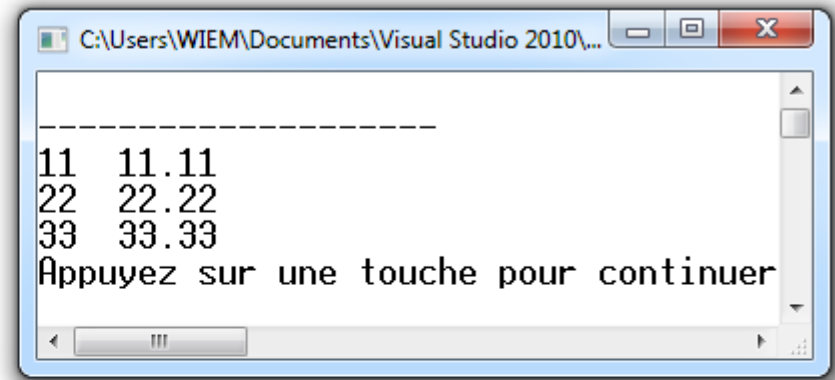
- **`m["med"]=16.5;`**
 - ➔ insertion de cet élément dans le conteneur
 - ➔ dans un vecteur, l'opérateur `[]` permet d'accéder uniquement aux éléments existants
- **`cout<<"\n la moyenne d'ALI "<<m["ALI"]<<endl;`**
 - ➔ le simple fait de chercher à consulter `m["ALI"]` créera l'élément correspondant, en initialisant la valeur associée à 0.

Remplissage et affichage d'un map v1

```
#include<map>
void main()
{
    map<int,float> m;
    map<int,float>::iterator it;

    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(33,33.33));
    cout<<"\n-----"<<endl;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;

    system("PAUSE");
}
```



C:\Users\WIEM\Documents\Visual Studio 2010\...

11 11.11
22 22.22
33 33.33
Appuyez sur une touche pour continuer

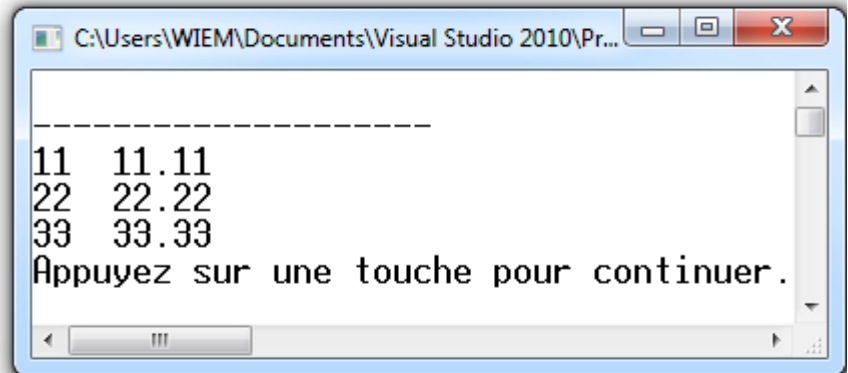
Remplissage et affichage d'un map v2

```
void afficher(map<int,float> m)
{
    map<int,float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    map<int,float> m;

    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(33,33.33));
    cout<<"\n-----"<<endl;

    afficher(m);
    system("PAUSE");
}
```



Le patron de classes pair

- il existe un patron de **classe *pair***, comportant deux paramètres de type et permettant de regrouper dans un objet deux valeurs

pair<string,float> a("AAA",11.11);

- Pour affecter des valeurs données à la paire (a)

```
a=pair<string,float> ("ZZZ", 99.99);  
  
a=make_pair("YYY",88.88);  
  
a.first="XXX";  
a.second=77.77;
```

- make_pair**: est une fonction standard

```

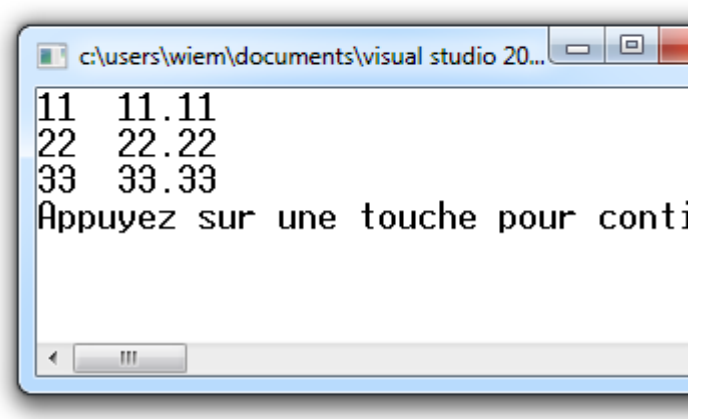
void main()
{
    pair<int, float> p(11,11.11);
    cout<<p.first<<" "<<p.second<<endl;

    // modification
    p.first=22;
    p.second=22.22;
    cout<<p.first<<" "<<p.second<<endl;

    // modification

    p=make_pair(33,33.33);
    cout<<p.first<<" "<<p.second<<endl;
    system("PAUSE");
}

```



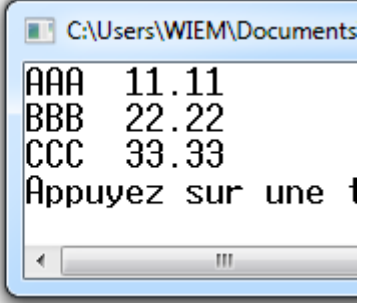
Map ordonné intrinsèquement

```
(Global Scope) main()
#include<map>
void main ()
{
    map<string, float> m;    // nom et moyenne
    map<string, float>::iterator it;

    pair<string,float> a("CCC",33.33);
    pair<string,float> b("AAA",11.11);
    pair<string,float> c("BBB",22.22);

    m.insert(a); m.insert(b); m.insert(c);

    for(it=m.begin(); it!=m.end(); it++)
        cout<<(*it).first<<"  "<<(*it).second<<endl;
    system("PAUSE");
}
```

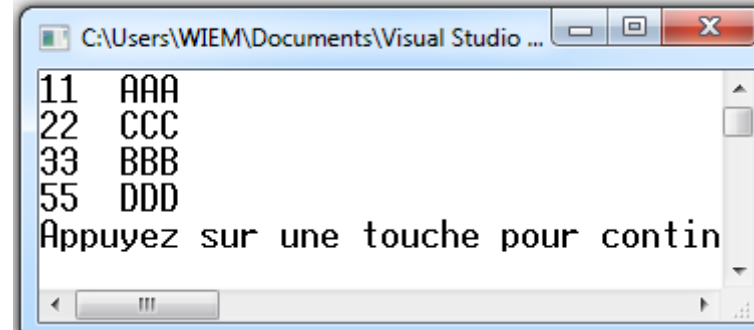


Création d'une paire

```
void main()
{
    map<int,string> m;
    // création d'une paire
    pair<int,string> a;
    a.first=11;
    a.second="AAA";
    // création d'une paire
    pair<int,string> b(33,"BBB");
    // création d'une paire avec insertion
    m[22]="CCC";
    // création d'une paire avec insertion
    m.insert(make_pair<int,string> (55,"DDD"));

    m.insert(a);
    m.insert(b);
    afficher(m);
    system("PAUSE");
}
```

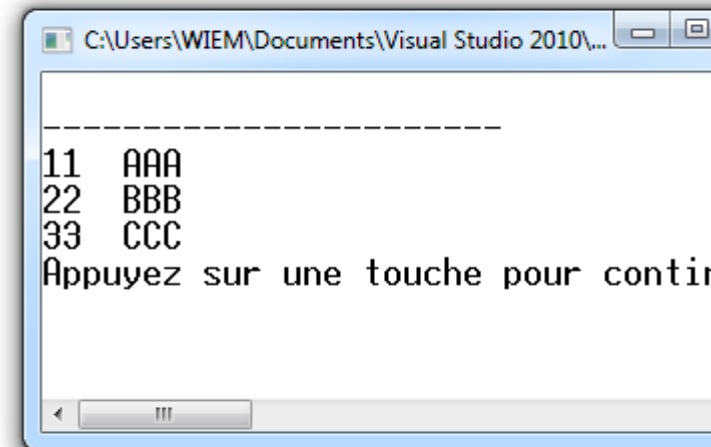
```
void afficher(map<int,string> m)
{
    map<int,string>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}
```



```
C:\Users\WIEM\Documents\Visual Studio ...
11 AAA
22 CCC
33 BBB
55 DDD
Appuyez sur une touche pour continuer
```

Modification d'une paire

```
void main()
{
    map<int,string> m;
    // création des paires
    pair<int,string> a;
    pair<int,string> b;
    pair<int,string> c;
    // modification 1 d'une paire qui existe déjà
    a=pair<int,string>(33,"CCC");
    // modification 2 d'une paire qui existe déjà
    b=make_pair (11,"AAA");
    // modification 3 d'une paire qui existe déjà
    c.first=22;
    c.second="BBB";
    cout<<"\n-----"<<endl;
    m.insert(a); m.insert(b); m.insert(c);
    afficher(m);
    system("PAUSE");
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\...
-----
11 AAA
22 BBB
33 CCC
Appuyez sur une touche pour continuer...
```

Construction d'un conteneur de type *map*

- 3 manières possibles pour construire un map
 - construction d'un conteneur vide
 - construction à partir d'un autre conteneur de même type;
 - construction à partir d'une séquence.

Constructions utilisant la relation d'ordre par défaut

```
void main ()  
{  
    // Construction d'un conteneur vide  
    map<string, float> m;  
    map<int,point> m2;  
  
    // remplissage de m  
  
    // Construction a partir d'un autre conteneur de même type  
    map<string, float> m3(m);  
}
```


Pour connaître la relation d'ordre utilisée par un conteneur

- Les classes *map* disposent d'une fonction membre *key_comp()* fournissant la fonction utilisée pour ordonner les clés.

```
map<char, int> m;  
  
~~~~~  
if('a' < 'c')....  
ou bien  
if(m.key_comp('a', 'c')....
```

- Remarque: Tant que l'on utilise des clés de type scalaire ou *string* et qu'on se limite à la relation par défaut (<), aucun problème particulier ne se pose.

Pour connaître la relation d'ordre utilisée par un conteneur

- D'une manière similaire, la classe *map* dispose d'une fonction membre *value_comp()* fournissant la fonction utilisable pour comparer deux éléments, toujours selon la valeur des clés.
- L'intérêt de cette fonction est de permettre de comparer deux éléments (donc, deux paires), suivant l'ordre des clés, sans avoir à en extraire les membres *first*.

```
map<char, int> m;  
map<char, int>::iterator it1, it2;  
  
// comparer les clés relatives aux éléments pointés par it1 et it2  
if(m.value_comp>(*it1, *it2))...  
  
// avec key_comp  
if(m.key_comp ((*it1).first, (*it2).first)) ...|
```

Accès aux éléments

- Comme tout conteneur, *map* permet théoriquement d'accéder aux éléments existants, soit pour en connaître la valeur, soit pour la modifier.
- une tentative d'accès à une clé inexistante amène à la création d'un nouvel élément,
- une tentative de modification globale (clé + valeur) d'un élément existant sera fortement déconseillée.

Accès par l'opérateur []

- Cet opérateur conduit à la création d'un nouvel élément, dès qu'on l'applique à une clé inexistante et cela, aussi bien en consultation qu'en modification.

```
map<char, int> m;
```

```
m['a']=3;
```

```
// si la clé 'a' n'existe pas, l'élément est créé make_pair('a',3)
```

```
// si la clé existe, on modifie la valeur de l'élément qui ne change pas de place
```

```
...=m['b'];
```

```
// si la clé b n'existe pas, on crée l'élément make_pair('b',0)
```

Accès par itérateur

- si *it* est un itérateur valide sur un conteneur de type *map*, l'expression **it* désigne l'élément correspondant.
- L'élément est une paire formée de la clé (**it*).*first* (ou bien *it->first*) et de la valeur associée (**it*).*second* (ou bien *it->second*)
- En théorie, il n'est pas interdit de modifier la valeur de l'élément désigné par *it*, mais c'est déconseillé

```
*it=make_pair('A',5);
```

- 2 problèmes:
 - comme une telle opération modifie la valeur de la clé, le nouvel élément risque de ne plus être à sa place; il devrait donc être déplacé.
 - il se peut que la clé 'A' existe déjà

Il est fortement déconseillé de modifier la valeur d'un élément d'un *map*, par le biais d'un itérateur.

Recherche par la fonction membre *find*

- La fonction membre *find* (clé) permet de fournir un itérateur sur un élément ayant une clé donnée. Si aucun élément n'est trouvé, cette fonction fournit la valeur *end()*

exemple

(Global Scope)

main()

```
void afficher(map<int,float> m)
{
    map<int,float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    map<int,float> m;
    map<int,float>::iterator it;
    // remplissage
    m[11]=11.11; m[55]=55.55; m[88]=88.88;
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(77,77.77));
    cout<<"\n-----"<<endl;
    it=m.find(55);
    if(it!= m.end()) m.erase(55);
    afficher(m);
    system("PAUSE");
}
```

C:\Users\WIEM\Documents\Visual Studio 2010\Projects\testPOO2016\

```
-----
11  11.11
22  22.22
77  77.77
88  88.88
```

Appuyez sur une touche pour continuer...

Insertions et suppressions

- le conteneur *map* offre des possibilités de modifications dynamiques fondées sur des insertions et des suppressions, analogues à celles qui sont offertes par les conteneurs séquentiels.

- insert (element);* // insère la paire *element*

```
map<int,float> m;  
  
m.insert(make_pair(99,99.99));
```

- insert (debut, fin);* // insère les paires de la séquence *[debut, fin)*

```
map<int,float> m1,m2;  
  
...  
m2.insert(m1.begin(), m1.end());
```

- Insert (position, paire);

```
m.insert(it, make_pair(33,33.33));
```

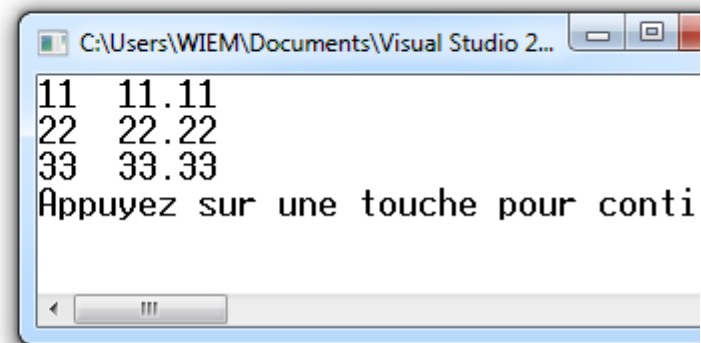

Exemple 1

```
void main()
{
    map<int,float> m;
    map<int,float>::iterator it;

    m.insert( make_pair(11,11.11) );
    m.insert( make_pair(33,33.33) );
    m.insert( make_pair(22,22.22) );

    for(it=m.begin() ; it!= m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;

    system("PAUSE");
}
```



Exemple 2

```
void main()
{
    map<int,float> m;
    map<int,float>::iterator it;

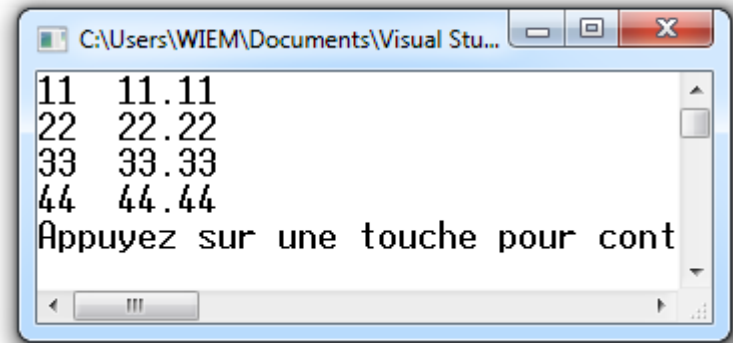
    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(44,44.44));

    it=m.begin(); it++;

    m.insert(it, make_pair(33,33.33));

    for(it=m.begin(); it!=m.end(); it++)
        cout<<(*it).first<<"  "<<(*it).second<<endl;

    svstem("PAUSE");
}
```



remarque

- Les deux fonctions d'insertion d'un élément fournissent une valeur de retour qui est une paire de la forme *pair(position, indice)*, dans laquelle le booléen *indice* précise si l'insertion a eu lieu et *position* est l'itérateur correspondant;

exemple

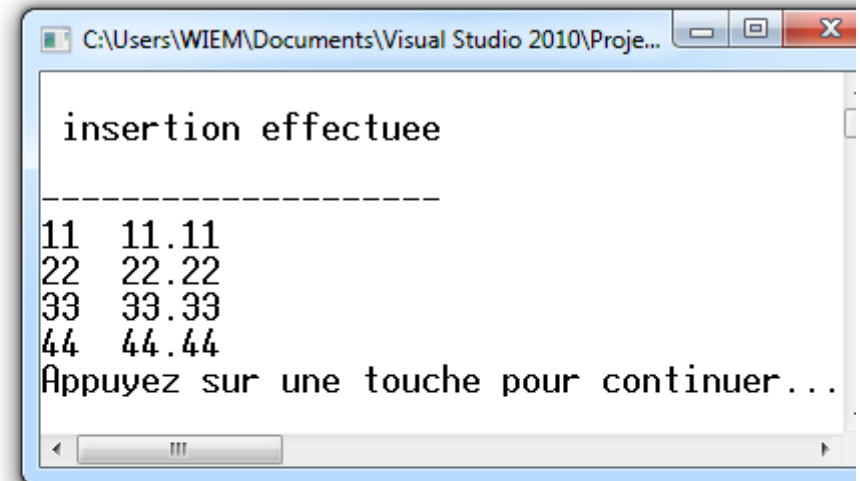
```
void main()
{
    map<int,float> m;
    map<int,float>::iterator it;

    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(44,44.44));

    it=m.begin(); it++;

    if((m.insert(it, make_pair(33,33.33))->second))
        cout<<"\n insertion effectuee "<<endl;
    else cout<<"\n element existant "<<endl;

    cout<<"\n-----"<<endl;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<(*it).first<<" "<<(*it).second<<endl;
}
```



suppression

- La fonction *erase* permet de supprimer:

- un élément de position donnée:

erase (position) // supprime l'élément désigné par *position*

- les éléments d'un intervalle:

erase (debut, fin) // supprime les paires de l'intervalle [*debut*, *fin*)

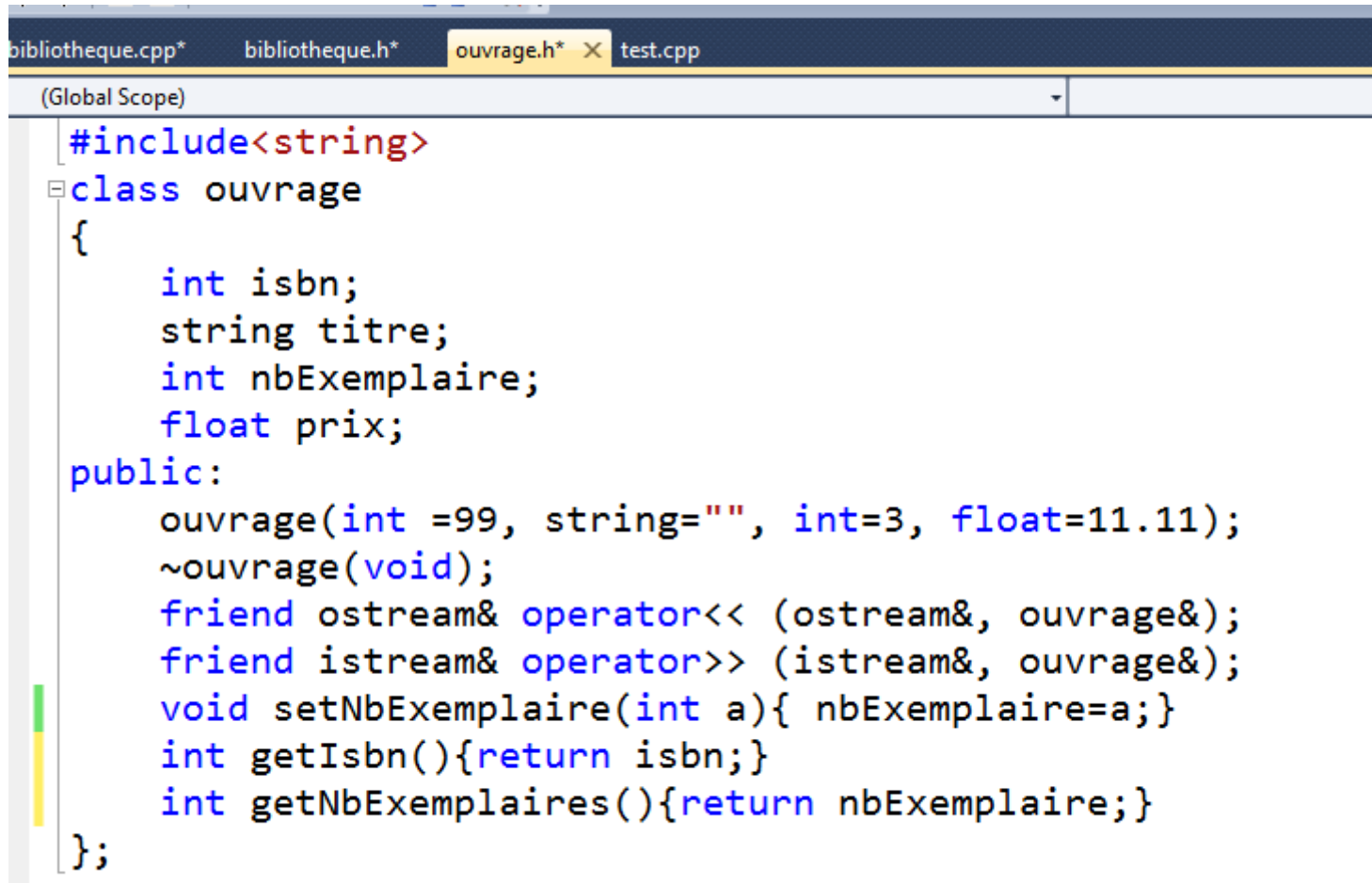
- l'élément de clé donnée:

erase (cle) // supprime les elements de clé équivalente a *cle*

```
void main()
{
    map<int,float> m;
    map<int,float>::iterator it1, it2;
    ....
    m.erase(5); // supprime l'élément de clé 5 s'il existe
    m.erase(it1); // supprime l'élément désigné par it1
    m.erase(it1, it2); // supprime les éléments de l'intervalle [it1, it2)

    m.clear(); // vider le conteneur
}
```

Exercice d'application (voir corrigé)



```
bibliotheque.cpp*  bibliotheque.h*  ouvrage.h* X test.cpp
(Global Scope)
#include<string>
class ouvrage
{
    int isbn;
    string titre;
    int nbExemplaire;
    float prix;
public:
    ouvrage(int =99, string="", int=3, float=11.11);
    ~ouvrage(void);
    friend ostream& operator<< (ostream&, ouvrage&);
    friend istream& operator>> (istream&, ouvrage&);
    void setNbExemplaire(int a){ nbExemplaire=a;}
    int getIsbn(){return isbn;}
    int getNbExemplaires(){return nbExemplaire;}
};
```

Exercice d'application (voir corrigé)

```
#pragma once
#include<iostream>
using namespace std;
#include<string>
#include<map>
#include"ouvrage.h"
class bibliotheque
{
    map<int, ouvrage> tab;
public:
    bibliotheque(void);
    ~bibliotheque(void);
    void inserer(pair<int,ouvrage>);
    void modifierNbExempl(int, int); //recherche selon code
    void modifierNbExemplIsbn(int, int); // recherche selon isbn
    friend ostream& operator<< (ostream&, bibliotheque&);
    friend istream& operator>> (istream&, bibliotheque&);
    bool rechercherIsbn(int);
};
```

- Le conteneur multimap
(voir corrigé)

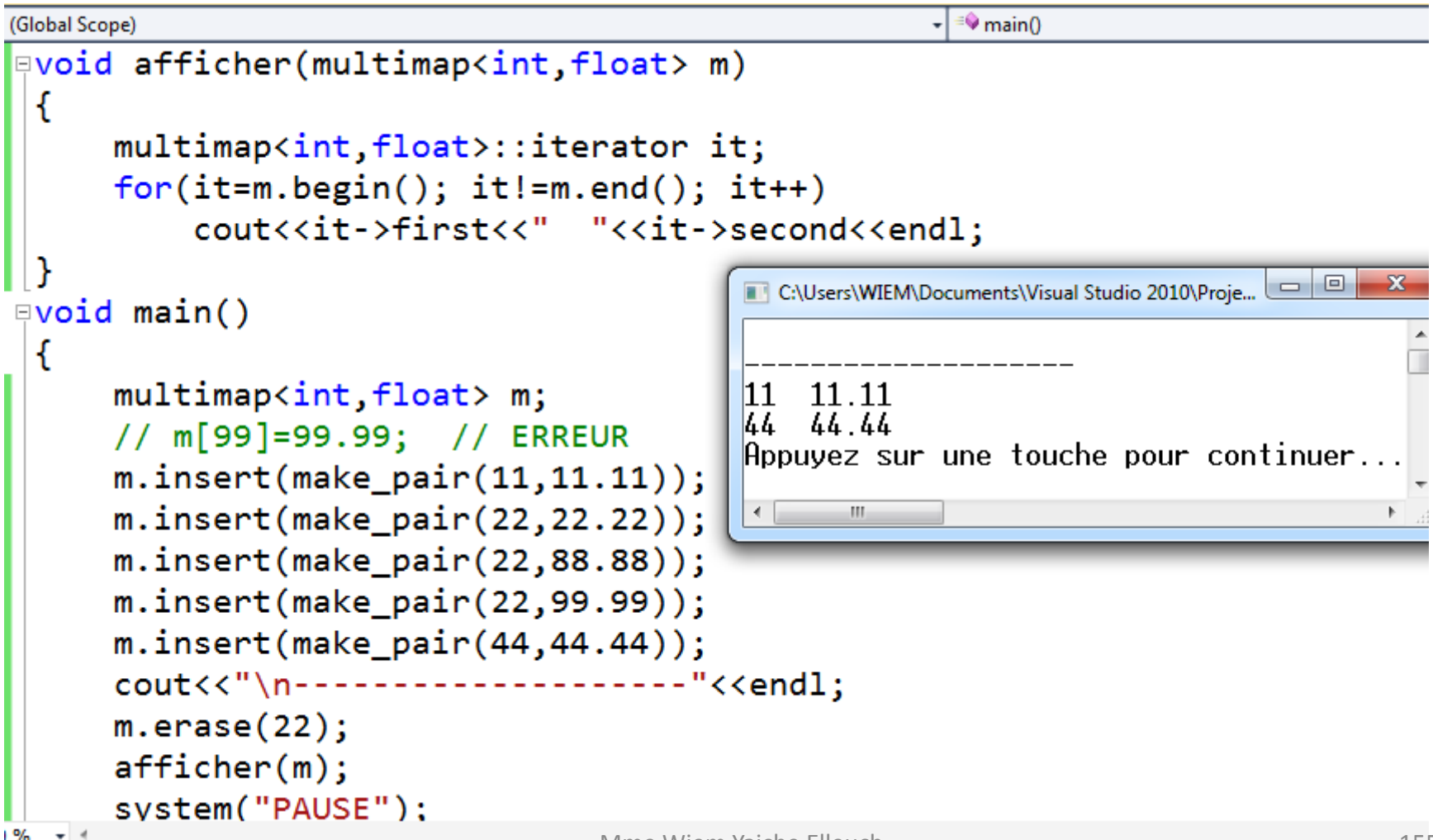
Le conteneur multimap

- dans un conteneur de type *multimap*, une même clé peut apparaître plusieurs fois (les éléments correspondants apparaissent alors consécutifs).
- **l'opérateur [] n'est plus applicable** à un tel conteneur, compte tenu de l'ambiguïté qu'induirait la non-unicité des clés.
- les possibilités des conteneurs *map* se généralisent sans difficultés aux conteneurs *multimap* qui possèdent les mêmes fonctions membres, mais:
 - s'il existe plusieurs clés équivalentes, la fonction membre **find** fournit un itérateur sur un des éléments ayant la clé voulue; on ne précise pas qu'il s'agit du premier; celui-ci peut être connu en recourant à la fonction *lower_bound*;
 - la fonction membre *erase (cle)* peut supprimer plusieurs éléments tandis qu'avec un conteneur *map*, elle n'en supprimait qu'un seul au maximum.

Le conteneur multimap

- un certain nombre de fonctions membres de la classe *map*, prennent tout leur intérêt lorsqu'on les applique à un conteneur *multimap*. On peut, en effet:
 - connaître le nombre d'éléments ayant une clé équivalente à une clé donnée, à l'aide de *count(cle)* ;
 - obtenir des informations concernant l'intervalle d'éléments ayant une clé équivalente à une clé donnée, à savoir:
 - *lower_bound(cle)* // fournit un itérateur sur le premier élément ayant // une clé équivalente à *cle*
 - *upper_bound(cle)* // fournit un itérateur sur le dernier élément ayant // une clé équivalente à *cle*
 - *equal_range(cle)* // fournit une paire formée des valeurs des deux // itérateurs précédents, *lower_bound(cle)* et *upper_bound(cle)*
- *m.equal_range(cle) = make_pair(m.lower_bound(cle), m.upper_bound(cle))*

Ne pas faire `#include<multimap>` // ERREUR
`#include<map>` // OK pour le multimap aussi



The screenshot shows a Visual Studio 2010 IDE window with a C++ project. The code is in a file named `main()` and is located in the `(Global Scope)`. The code defines a function `afficher` that iterates over a `multimap<int, float>` and prints its elements. The `main` function creates a `multimap<int, float>` `m`, inserts several pairs, erases the pair with key 22, and calls `afficher`. The console output shows the pairs (11, 11.11) and (44, 44.44) printed, followed by a prompt to press a key to continue.

```
(Global Scope) main()
void afficher(multimap<int, float> m)
{
    multimap<int, float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    multimap<int, float> m;
    // m[99]=99.99; // ERREUR
    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(22,88.88));
    m.insert(make_pair(22,99.99));
    m.insert(make_pair(44,44.44));
    cout<<"\n-----"<<endl;
    m.erase(22);
    afficher(m);
    system("PAUSE");
}
```

Console Output:

```
-----
11 11.11
44 44.44
Appuyez sur une touche pour continuer...
```

```

void afficher(multimap<int,float> m)
{
    multimap<int,float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    multimap<int,float> m;
    // m[99]=99.99; // ERREUR
    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(22,88.88));
    m.insert(make_pair(22,99.99));
    m.insert(make_pair(44,44.44));
    cout<<"\n-----"<<endl;
    cout<<m.count(22)<<endl;
    cout<<"\n-----"<<endl;
    afficher(m);
}

```

```

C:\Users\WIEM\Documents\Visual Studio 2010\Proj...
-----
3
-----
11 11.11
22 22.22
22 88.88
22 99.99
44 44.44
Appuyez sur une touche pour continuer..
3

```

```

void afficher(multimap<int,float> m)
{
    multimap<int,float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    multimap<int,float> m;
    // m[99]=99.99; // ERREUR
    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(22,88.88));
    m.insert(make_pair(22,99.99));
    m.insert(make_pair(44,44.44));
    cout<<"\n-----"<<endl;
    m.erase(m.lower_bound(22));
    cout<<"\n-----"<<endl;
    afficher(m);
}

```

```

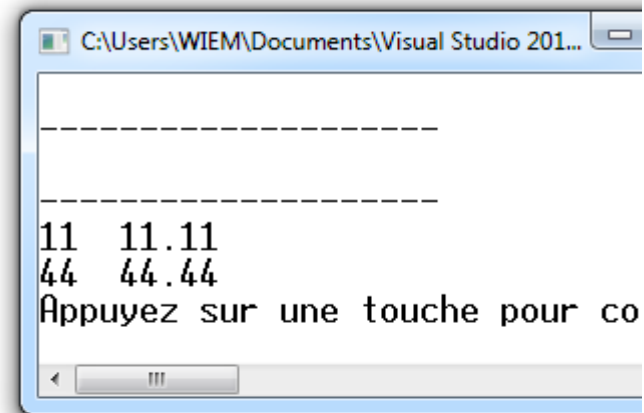
C:\Users\WIEM\Documents\Visual Studio 2010\...
-----
11 11.11
22 88.88
22 99.99
44 44.44
Appuyez sur une touche pour contin

```

```
test.cpp x
(Global Scope) main()

void afficher(multimap<int,float> m)
{
    multimap<int,float>::iterator it;
    for(it=m.begin(); it!=m.end(); it++)
        cout<<it->first<<" "<<it->second<<endl;
}

void main()
{
    multimap<int,float> m;
    // m[99]=99.99; // ERREUR
    m.insert(make_pair(11,11.11));
    m.insert(make_pair(22,22.22));
    m.insert(make_pair(22,88.88));
    m.insert(make_pair(33,33.33));
    m.insert(make_pair(33,99.99));
    m.insert(make_pair(44,44.44));
    cout<<"\n-----"<<endl;
    m.erase(m.lower_bound(22), m.upper_bound(33));
    cout<<"\n-----"<<endl;
}
```



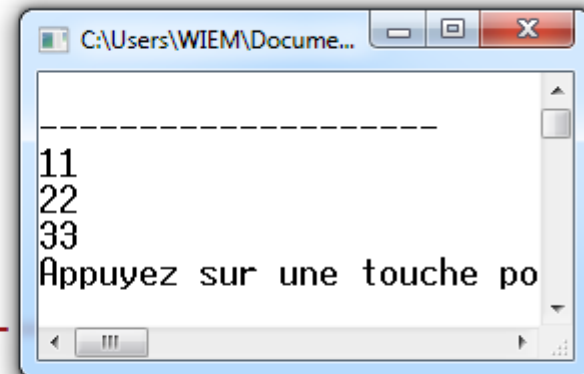
- Le conteneur **set**
- (**ensemble**)

Le conteneur set

- le conteneur *set* est un cas particulier du conteneur *map*, dans lequel **aucune valeur** n'est associée à la clé, Les éléments d'un conteneur *set* ne sont donc plus des paires, ce qui en facilite la manipulation,
- Une autre différence entre les conteneurs *set* et les conteneurs *map* est **qu'un élément d'un conteneur *set* est une constante**; on ne peut pas en modifier la valeur.
- Un conteneur de type *set* est obligatoirement ordonné

Le conteneur set

```
#include<set>
void afficher(set<int> s)
{
    set<int>::iterator it;
    for(it=s.begin(); it!=s.end(); it++)
        cout<<*it<<endl;
}
void main()
{
    set<int> s;
    s.insert(33);
    s.insert(11);
    s.insert(22);
    cout<<"\n-----"
    afficher(s);
    system("PAUSE");
}
```



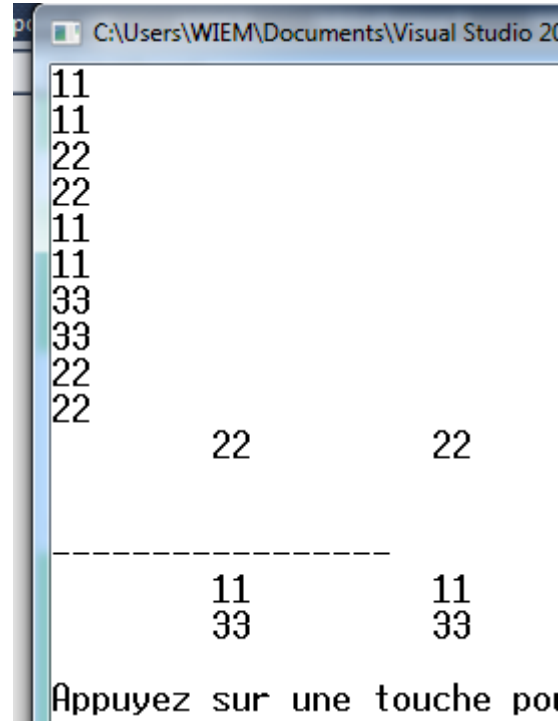
```
class point
{
protected:
    int x;
    int y;
public:
    friend bool operator< (const point&, const point&) ;
```

```
bool operator< (const point& pt, const point& p)
{
    return (pt.x< p.x );
}
```

Création, et remplissage de 2 ensembles (sets) de points (pp et pi), selon un critère (attribut de point (x))

```
void main()
{
    set<point> pp; // points ayant x pair
    set<point> pi; // points ayant x impair
    point a;
    for(int i=1; i<6;i++)
    {
        cin>>a;
        if(a.getX()%2==0) pp.insert(a);
        else pi.insert(a);
    }
    afficher(pp);
    cout<<"\n-----"<<endl;
    afficher(pi);
}
```

```
void afficher(set<point> s)
{
    set<point>::iterator it;
    for(it= s.begin() ; it != s.end() ; it++)
        cout<<*it<<endl;
    cout<<endl;
}
```



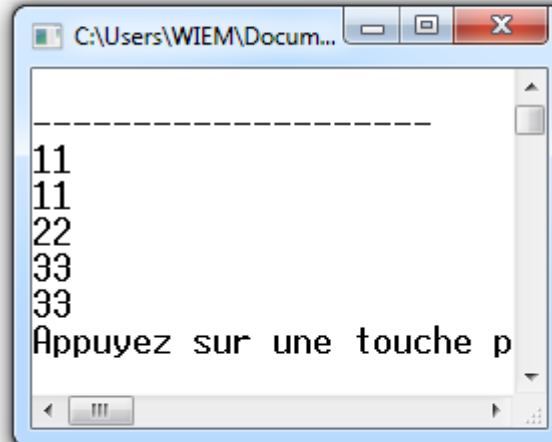
- Le conteneur multiset
- multi

le conteneur *multimap*

- De même que le conteneur *multimap* est un conteneur *map*, dans lequel on autorise plusieurs clés équivalentes, le conteneur *multiset* est un conteneur *set*, dans lequel on autorise plusieurs éléments équivalents à apparaître.

Le conteneur multiset

```
#include<set>
void afficher(multiset<int> s)
{
    multiset<int>::iterator it;
    for(it=s.begin(); it!=s.end(); it++)
        cout<<*it<<endl;
}
void main()
{
    multiset<int> s;
    s.insert(33);
    s.insert(11);
    s.insert(22);
    s.insert(11);
    s.insert(33);
    cout<<"\n-----"<<endl;
    afficher(s);
}
```



```

#include<set>
void afficher(multiset<point> s)
{
    multiset<point>::iterator it;
    for(it= s.begin() ; it != s.end() ; it++)
        cout<<*it<<endl;
    cout<<endl;
}

void main()
{
    multiset<point> pp; // points ayant x pair
    multiset<point> pi; // points ayant x impair
    point a;
    for(int i=1; i<6;i++)
    {
        cin>>a;
        if(a.getX()%2==0) pp.insert(a);
        else pi.insert(a);
    }
    afficher(pp);
    cout<<"\n-----"<<endl;
    afficher(pi);
}

```

```

C:\Users\WIEM\Documents\Visual Studio
11
11
22
22
11
11
33
33
22
22
22
22
-----
11
11
33
11
11
33
Appuyez sur une touche pour continuer...

```

Les algorithmes standard

- Les algorithmes standard se présentent sous forme de patrons de fonctions
 1. **Algorithmes d'initialisation de séquences existantes**
 2. **Algorithmes de recherche**
 3. **Algorithmes de transformation d'une séquence**
 4. **Algorithmes de suppression**
 5. **Algorithmes de tri**
 6. **Algorithmes de recherche et de fusion sur des séquences ordonnées**
 7. **Algorithmes à caractère numérique**
 8. **Algorithmes à caractère ensembliste**
 9. **Algorithmes de manipulation de tas**
 10. **Algorithmes divers**

rappel

La nature des itérateurs reçus en argument est précisée en utilisant les abréviations suivantes

- le : Itérateur d'entrée ;
- ls : Itérateur de sortie ;
- lu : Itérateur unidirectionnel ;
- lb : Itérateur bidirectionnel ;
- la : Itérateur à accès direct.


```

void afficher(vector<int> v)
{
    vector<int>::iterator it;
    cout<<"\n affichage du vecteur "<<endl;
    for(it=v.begin(); it!=v.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
}

void afficher(list<int> l)
{
    list<int>::iterator it;
    cout<<"\n affichage de la liste "<<endl;
    for(it=l.begin(); it!=l.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
}

```

(Global Scope)

main()

```

void main()
{
    vector<int> v(5,999);
    list<int> l(3,111);
    cout<<"\n-----
    affichage(v);
    afficher(l);
    system("PAUSE");
}

```

C:\Users\WIEM\Documents\Visual Studio 2010\Projects\testPOO2016\D...

```

-----
affichage du vecteur
999 999 999 999 999

affichage de la liste
111 111 111
Appuyez sur une touche pour continuer...

```

Mme Wient Yaiche Elleuch

- 1. Algorithmes d'initialisation de séquences existantes**
- 2. Algorithmes de recherche**
- 3. Algorithmes de transformation d'une séquence**
- 4. Algorithmes de suppression**
- 5. Algorithmes de tri**
- 6. Algorithmes de recherche et de fusion sur des séquences ordonnées**
- 7. Algorithmes à caractère numérique**
- 8. Algorithmes à caractère ensembliste**
- 9. Algorithmes de manipulation de tas**
- 10. Algorithmes divers**

Algorithmes d'initialisation de séquences existantes

void fill (lu début, lu fin, valeur)

→ Place valeur dans l'intervalle [début, fin).

void fill_n (ls position, NbFois, valeur)

→ Place valeur NbFois consécutives à partir de position ; les emplacements correspondants doivent exister.

ls copy (le début, le fin, ls position)

→ Copie l'intervalle [début, fin), à partir de position ; les emplacements correspondants doivent exister ;

lb copy_backward (lb début, lb fin, lb position)

→ Comme copy, copie l'intervalle [début, fin), en progressant du dernier élément vers le premier, à partir de position qui désigne donc l'emplacement de la première copie, mais aussi la fin de l'intervalle ;

Algorithmes d'initialisation de séquences existantes

void generate (lu début, lu fin, fct_gen)

→ Appelle, pour chacune des valeurs de l'intervalle [début, fin), la fonction fct_gen et affecte la valeur fournie à l'emplacement correspondant.

void generate_n (lu début, NbFois, fct_gen)

→ Même chose que generate, mais l'intervalle est défini par sa position début et son nombre de valeurs NbFois (la fonction fct_gen est bien appelée NfFois).

lu swap_ranges (lu début_1, lu fin_1, lu début_2)

→ Échange les éléments de l'intervalle [début, fin) avec l'intervalle de même taille commençant en début_2. Les deux intervalles ne doivent pas se chevaucher.

1. Algorithmes d'initialisation de séquences existantes
2. **Algorithmes de recherche**
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes de recherche

le **find** (le début, le fin, valeur)

→ Fournit un itérateur sur le premier élément de l'intervalle [début, fin) égal à valeur (au sens de $==$) s'il existe, la valeur fin sinon.

le **find_if** (le début, le fin, prédicat_u)

→ Fournit un itérateur sur le premier élément de l'intervalle [début, fin) satisfaisant au prédicat unaire prédicat_u spécifié, s'il existe, la valeur fin sinon

lu **find_end** (lu début_1, lu fin_1, lu début_2, lu fin_2)

→ Fournit un itérateur sur le dernier élément de l'intervalle [début_1, fin_1) tel que les éléments de la séquence débutant en début_1 soit égaux (au sens de $==$) aux éléments de l'intervalle [début_2, fin_2).

Algorithmes de recherche

lu **find_end** (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)

➔ Fonctionne comme la version précédente, avec cette différence que la comparaison d'égalité est remplacée par l'application du prédicat binaire prédicat_b.

lu **find_first_of** (lu début_1, lu fin_1, lu début_2, lu fin_2)

➔ Recherche, dans l'intervalle [début_1, fin_1), le premier élément égal (au sens de ==) à l'un des éléments de l'intervalle [début_2, fin_2). Fournit un itérateur sur cet élément s'il existe, la valeur de fin_1, dans le cas contraire.

lu **find_first_of** (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)

➔ Recherche, dans l'intervalle [début_1, fin_1), le premier élément satisfaisant, avec l'un des éléments de l'intervalle [début_2, fin_2) au prédicat binaire prédicat_b. Fournit un itérateur sur cet élément s'il existe, la valeur de fin_1, dans le cas contraire

Algorithmes de recherche

lu **adjacent_find** (lu début, lu fin)

➔ Recherche, dans l'intervalle [début, fin), la première occurrence de deux éléments successifs égaux (==) ; fournit un itérateur sur le premier des deux éléments égaux, s'ils existent, la valeur fin sinon.

lu **adjacent_find** (lu début, lu fin, prédicat_b)

➔ Recherche, dans l'intervalle [début, fin), la première occurrence de deux éléments successifs satisfaisant au prédicat binaire prédicat_b ; fournit un itérateur sur le premier des deux éléments, s'ils existent, la valeur fin sinon.

Algorithmes de recherche

lu **search** (lu début_1, lu fin_1, lu début_2, lu fin_2)

➔ Recherche, dans l'intervalle [début_1, fin_1), la première occurrence d'une séquence d'éléments identique (==) à celle de l'intervalle [début_2, fin_2). Fournit un itérateur sur le premier élément de cette occurrence, si elle existe, la fin fin_1 sinon.

lu **search** (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)

➔ Fonctionne comme la version précédente de search, avec cette différence que la comparaison de deux éléments de chacune des deux séquences se fait par le prédicat binaire prédicat_b, au lieu de se faire par égalité.

Algorithmes de recherche

lu **search_n** (ludébut, lu fin, NbFois, valeur)

➔ Recherche dans l'intervalle [début, fin), une séquence de NbFois éléments égaux (au sens de ==) à valeur. Fournit un itérateur sur le premier élément si une telle séquence existe, la valeur fin sinon

lu **search_n** (ludébut, lu fin, NbFois, valeur, prédicat_b)

➔ Fonctionne comme la version précédente avec cette différence que la comparaison entre un élément et valeur se fait par le prédicat binaire prédicat_b, au lieu de se faire par égalité. Complexité : au maximum N applications du prédicat.

Algorithmes de recherche

lu **max_element** (lu début, lu fin)

➔ Fournit un itérateur sur le premier élément de l'intervalle [début, fin) qui ne soit inférieur ($<$) à aucun des autres éléments de l'intervalle

lu **max_element** (lu début, lu fin, prédicat_b)

➔ Fonctionne comme la version précédente de max_element, mais en utilisant le prédicat binaire prédicat_b en lieu et place de l'opérateur $<$.

lu **min_element** (lu début, lu fin)

➔ Fournit un itérateur sur le premier élément de l'intervalle [début, fin), tel qu'aucun des autres éléments de l'intervalle ne lui soit inférieur ($<$).

lu **min_element** (lu début, lu fin, prédicat_b)

➔ Fonctionne comme la version précédente de min_element, mais en utilisant le prédicat binaire prédicat_b en lieu et place de l'opérateur $<$.

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. **Algorithmes de transformation d'une séquence**
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes de transformation d'une séquence

void **reverse** (lb début, lb fin)

➔ Inverse le contenu de l'intervalle [début, fin).

ls **reverse_copy** (lb début, lb fin, ls position)

➔ Copie l'intervalle [début, fin), dans l'ordre inverse, à partir de position ; les emplacements correspondants doivent exister ; attention, ici position désigne donc l'emplacement de la première copie et aussi le début de l'intervalle ; renvoie un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher

Algorithmes de transformation d'une séquence

void **replace** (lu début, lu fin, anc_valeur, nouv_valeur)

➔ Remplace, dans l'intervalle [début, fin), tous les éléments égaux (==) à anc_valeur par nouv_valeur.

void **replace_if** (lu début, lu fin, prédicat_u, nouv_valeur)

➔ Remplace, dans l'intervalle [début, fin), tous les éléments satisfaisant au prédicat unaire prédicat_u par nouv_valeur.

ls **replace_copy** (le début, le fin, ls position, anc_valeur, nouv_valeur)

➔ Recopie l'intervalle [début, fin) à partir de position, en remplaçant tous les éléments égaux (==) à anc_valeur par nouv_valeur ; les emplacements correspondants doivent exister. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher.

Algorithmes de transformation d'une séquence

Is **replace_copy_if** (le début, le fin, Is position, prédicat_u, nouv_valeur)

→ Recopie l'intervalle [début, fin) à partir de position, en remplaçant tous les éléments satisfaisant au prédicat unaire prédicat_u par nouv_valeur ; les emplacements correspondants doivent exister Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher.

void **rotate** (lu début, lu milieu, lu fin)

→ Effectue une permutation circulaire (vers la gauche) des éléments de l'intervalle [début, fin) dont l'ampleur est telle que, après permutation, l'élément désigné par milieu soit venu en début.

Is **rotate_copy** (lu début, lu milieu, lu fin, Is position)

→ Recopie, à partir de position, les éléments de l'intervalle [début, fin), affectés d'une permutation circulaire définie de la même façon que pour rotate ; les emplacements correspondants doivent exister. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie.

Algorithmes de transformation d'une séquence

lb **partition** (lb début, lb fin, Prédicat_u)

→ Effectue une partition de l'intervalle [début, fin) en se fondant sur le prédicat unaire `prédicat_u` ; il s'agit d'une réorganisation telle que tous les éléments satisfaisant au prédicat arrivent avant tous les autres. Fournit un itérateur `it` tel que les éléments de l'intervalle [début, `it`) satisfont au prédicat, tandis que les éléments de l'intervalle [`it`, fin) n'y satisfont pas.

lb **stable_partition** (lb début, lb fin, Prédicat_u)

→ Fonctionne comme `partition`, avec cette différence que les positions relatives des différents éléments à l'intérieur de chacune des deux parties sont préservées

Algorithmes de transformation d'une séquence

bool next_permutation (lb début, lb fin)

➔ Cet algorithme réalise ce que l'on nomme la « permutation suivante » des éléments de l'intervalle [début, fin). Il suppose que l'ensemble des permutations possibles est ordonné à partir de l'opérateur $<$, d'une manière lexicographique. On considère que la permutation suivant la dernière possible n'est rien d'autre que la première. Fournit la valeur true s'il existait bien une permutation suivante et la valeur false dans le cas où l'on est revenu à la première permutation possible.

bool next_permutation (lb début, lb fin, prédicat_b)

➔ Fonctionne comme la version précédente, avec cette seule différence que l'ensemble des permutations possibles est ordonné à partir du prédicat binaire prédicat_b.

Algorithmes de transformation d'une séquence

bool prev_permutation (lb début, lb fin)

bool prev_permutation (lb début, lb fin, prédicat_b)

→ Ces deux algorithmes fonctionnent comme next_permutation, en inversant simplement l'ordre des permutations possibles.

void random_shuffle (la début, la fin)

→ Répartit au hasard les éléments de l'intervalle [début, fin).

void random_shuffle (la début, la fin, générateur)

→ Même chose que random_shuffle, mais en utilisant la fonction générateur pour générer des nombres au hasard. Cette fonction doit fournir une valeur appartenant à l'intervalle $[0, n)$, n étant une valeur fournie en argument.

Algorithmes de transformation d'une séquence

Is **transform** (le début, le fin, ls position, opération_u)

➔ Place à partir de position (les éléments correspondants doivent exister) les valeurs obtenues en appliquant la fonction unaire (à un argument) opération_u à chacune des valeurs de l'intervalle [début, fin). Fournit un itérateur sur la fin de l'intervalle ainsi rempli.

Is **transform** (le début_1, le fin_1, le début_2, ls position, opération_b)

➔ Place à partir de position (les éléments correspondants doivent exister) les valeurs obtenues en appliquant la fonction binaire (à deux arguments) opération_b à chacune des valeurs de même rang de l'intervalle [début_1, fin_1) et de l'intervalle de même taille commençant en début_2. Fournit un itérateur sur la fin de l'intervalle ainsi rempli.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. **Algorithmes de suppression**
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes de suppression

lu `remove` (lu début, lu fin, valeur)

→ Fournit un itérateur `it` tel que l'intervalle `[début, it)` contienne toutes les valeurs initialement présentes dans l'intervalle `[début, fin)`, débarrassées de celles qui sont égales (`==`) à `valeur`. Attention, aucun élément n'est détruit ; tout au plus, peut-il avoir changé de valeur. L'algorithme est stable, c'est-à-dire que les valeurs non éliminées conservent leur ordre relatif

lu `remove_if` (lu début, lu fin, prédicat_u)

→ Fonctionne comme `remove`, avec cette différence que la condition d'élimination est fournie sous forme d'un prédicat unaire `prédicat_u`.

ls `remove_copy` (le début, le fin, ls position, valeur)

→ Recopie l'intervalle `[début, fin)` à partir de `position` (les éléments correspondants doivent exister), en supprimant les éléments égaux (`==`) à `valeur`. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Comme `remove`, l'algorithme est stable

Algorithmes de suppression

Is **remove_if** (le début, le fin, Is position, prédicat_u)

→ Fonctionne comme remove_copy, avec cette différence que la condition d'élimination est fournie sous forme d'un prédicat unaire prédicat_u

lu **unique** (lu début, lu fin)

→ Fournit un itérateur it tel que l'intervalle [début, it) corresponde à l'intervalle [début, fin), dans lequel les séquences de plusieurs valeurs consécutives égales (==) sont remplacées par la première. Attention, aucun élément n'est détruit ; tout au plus, peut-il avoir changé de place et de valeur.

lu **unique** (lu début, lu fin, prédicat_b)

→ Fonctionne comme la version précédente, avec cette différence que la condition de répétition est fournie sous forme d'un prédicat binaire prédicat_b.

Algorithmes de suppression

Is **unique_copy** (le début, le fin, Is position)

➔ Recopie l'intervalle [début, fin) à partir de position (les éléments correspondants doivent exister), en ne conservant que la première valeur des séquences de plusieurs valeurs consécutives égales (==). Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher.

Is **unique_copy** (le début, le fin, Is position, prédicat_b)

➔ Fonctionne comme unique_copy, avec cette différence que la condition de répétition de deux valeurs est fournie sous forme d'un prédicat binaire prédicat_u. On notera que la décision d'élimination d'une valeur se fait toujours par comparaison avec la précédente et non avec la première d'une séquence ; cette remarque n'a en fait d'importance qu'au cas où le prédicat fourni ne serait pas transitif.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. **Algorithmes de tri**
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Les algorithmes de tri

void `sort` (la début, la fin)

- ➔ Trie les éléments de l'intervalle [début, fin), en se fondant sur l'opérateur $<$. L'algorithme n'est pas stable, c'est-à-dire que l'ordre relatif des éléments équivalents (au sens de $<$) n'est pas nécessairement respecté.

void `sort` (la début, la fin, fct_comp)

- ➔ Trie les éléments de l'intervalle [début, fin), en se fondant sur le prédicat binaire fct_comp.

void `stable_sort` (la début, la fin)

- ➔ Trie les éléments de l'intervalle [début, fin), en se basant sur l'opérateur $<$. Contrairement à sort, cet algorithme est stable..

void `stable_sort` (la début, la fin, fct_comp)

- ➔ Même chose que stable_sort en se basant sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict.

Les algorithmes de tri

void `partial_sort` (la début, la milieu, la fin)

➔ Réalise un tri partiel des éléments de l'intervalle [début, fin), en se basant sur l'opérateur $<$ et en plaçant les premiers éléments convenablement triés dans l'intervalle [début, milieu) (c'est la taille de cet intervalle qui définit l'ampleur du tri). Les éléments de l'intervalle [milieu, fin) sont placés dans un ordre quelconque.

void `partial_sort` (la début, la milieu, la fin, fct_comp)

➔ Fonctionne comme `partial_sort`, avec cette différence qu'au lieu de se fonder sur l'opérateur $<$, cet algorithme se fonde sur le prédicat binaire `fct_comp` qui doit correspondre à une relation d'ordre faible strict.

la **partial_sort_copy** (le début, le fin, la pos_début, la pos_fin)

➔ Place dans l'intervalle [pos_début, pos_fin) le résultat du tri partiel ou total des éléments de l'intervalle [début, fin). Si l'intervalle de destination comporte plus d'éléments que l'intervalle de départ, ses derniers éléments ne seront pas utilisés. Fournit un itérateur sur la fin de l'intervalle de destination (pos_fin) lorsque ce dernier est de taille inférieure ou égale à l'intervalle d'origine). Les deux intervalles ne doivent pas se chevaucher.

la **partial_sort_copy** (le début, le fin, la pos_début, la pos_fin, fct_comp)

➔ Fonctionne comme partial_sort_copy avec cette différence qu'au lieu de se fonder sur l'opérateur <, cet algorithme se fonde sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict.

void nth_element (la début, la position, la fin)

- ➔ Place dans l'emplacement désigné par position – qui doit donc appartenir à l'intervalle [début, fin) – l'élément de l'intervalle [début, fin) qui se trouverait là, à la suite d'un tri. Les autres éléments de l'intervalle peuvent changer de place.

void nth_element (la début, la position, la fin, fct_comp)

- ➔ Fonctionne comme la version précédente, avec cette différence qu'au lieu de se fonder sur l'opérateur <, cet algorithme se fonde sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes de recherche et de fusion sur des séquences ordonnées

lu **lower_bound** (lu début, lu fin, valeur)

→ Fournit un itérateur sur la première position où valeur peut être insérée, compte tenu de l'ordre induit par l'opérateur <.

lu **lower_bound** (lu début, lu fin, valeur, fct_comp)

→ Fournit un itérateur sur la première position où valeur peut être insérée, Compte tenu de l'ordre induit par le prédicat binaire fct_comp.

lu **upper_bound** (lu début, lu fin, valeur)

→ Fournit un itérateur sur la dernière position où valeur peut être insérée, compte tenu de l'ordre induit par l'opérateur <.

lu **upper_bound** (lu début, lu fin, valeur, fct_comp)

→ Fournit un itérateur sur la dernière position où valeur peut être insérée, compte tenu de l'ordre induit par le prédicat binaire fct_comp.

Algorithmes de recherche et de fusion sur des séquences ordonnées

pair <lu, lu> **equal_range** (lu début, lu fin, valeur)

→ Fournit le plus grand intervalle [it1, it2) tel que valeur puisse être insérée en n'importe quel point de cet intervalle, compte tenu de l'ordre induit par l'opérateur <.

pair <lu, lu> **equal_range** (lu début, lu fin, valeur, fct_comp)

→ Fonctionne comme la version précédente, en se basant sur l'ordre induit par le prédicat binaire fct_comp au lieu de l'opérateur <.

bool **binary_search** (lu début, lu fin, valeur)

→ Fournit la valeur true s'il existe, dans l'intervalle [début, fin), un élément équivalent à valeur, et la valeur false, dans le cas contraire.

Algorithmes de recherche et de fusion sur des séquences ordonnées

bool **binary_search** (lu début, lu fin, valeur, fct_comp)

➔ Fournit la valeur true s'il existe, dans l'intervalle [début, fin), un élément équivalent à valeur (au sens de la relation induite par le prédicat fct_comp) et la valeur false dans le cas contraire.

ls **merge** (le début_1, le fin_1, le début_2, le fin_2, ls position)

➔ Fusionne les deux intervalles [début_1, fin_1) et [début_2, fin_2), à partir de position (les éléments correspondants doivent exister), en se fondant sur l'ordre induit par l'opérateur <. L'algorithme est stable : l'ordre relatif d'éléments équivalents dans l'un des intervalles d'origine est respecté dans l'intervalle d'arrivée ; si des éléments équivalents apparaissent dans les intervalles à fusionner, ceux du premier intervalle apparaissent toujours avant ceux du second. L'intervalle d'arrivée ne doit pas chevaucher les intervalles d'origine (en revanche, rien n'interdit que les deux intervalles d'origine se chevauchent).

Algorithmes de recherche et de fusion sur des séquences ordonnées

Is **merge** (le début_1, le fin_1, le début_2, le fin_2, Is position, fct_comp)

➔ Fonctionne comme la version précédente, avec cette différence que l'on se base sur l'ordre induit par le prédicat binaire fct_comp

void **inplace_merge** (lb début, lb milieu, lb fin)

➔ Fusionne les deux intervalles [début, milieu) et [milieu, fin) dans l'intervalle [début, fin) en se basant sur l'ordre induit par l'opérateur <.

void **inplace_merge** (lb début, lb milieu, lb fin, fct_comp)

➔ Fonctionne comme la version précédente, avec cette différence que l'on se base sur l'ordre induit par le prédicat binaire fct_comp.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. **Algorithmes à caractère numérique**
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes à caractère numérique

valeur **accumulate** (le debut, le fin, val_init)

→ Fournit la valeur obtenue en ajoutant (opérateur +) à la valeur initiale val_init, la valeur de chacun des éléments de l'intervalle [debut, fin).

valeur **accumulate** (le debut, le fin, val_initiale, fct_cumul)

→ Fonctionne comme la version précédente, en la généralisant : l'opération appliquée n'étant plus définie par l'opérateur +, mais par la fonction fct_cumul, recevant deux arguments du type des éléments concernés et fournissant un résultat de ce même type (la valeur accumulée courante est fournie en premier argument, celle de l'élément courant, en second).

valeur **inner_product** (le début_1, le fin_1, le début_2, val_init)

→ Fournit le produit scalaire de la séquence des valeurs de l'intervalle [début_1, fin_2) et de la séquence de valeurs de même longueur débutant en début_2, augmenté de la valeur initiale val_init.

Algorithmes à caractère numérique

valeur **inner_product** (le début_1, le fin_1, le début_2, val_init, fct_cumul, fct_prod)

→ Fonctionne comme la version précédente, en remplaçant l'opération de cumul (+) par l'appel de la fonction fct_cumul (la valeur cumulée est fournie en premier argument) et l'opération de produit par l'appel de la fonction fct_prod (la valeur courante du premier intervalle étant fournie en premier argument).

Is **partial_sum** (le début, le fin, Is position)

→ Crée, à partir de position (les éléments correspondants doivent exister), un intervalle de même taille que l'intervalle [début, fin), contenant les sommes partielles du premier intervalle : le premier élément correspond à la première valeur de [début, fin), le second élément à la somme des deux premières valeurs et ainsi de suite. Fournit un itérateur sur la fin de l'intervalle créé.

Is **partial_sum** (le début, le fin, Is position, fct_cumul)

→ Fonctionne comme la version précédente, en remplaçant l'opération de sommation (+) par l'appel de la fonction fct_cumul (la valeur cumulée est fournie en premier argument).

Algorithmes à caractère numérique

Is **adjacent_difference** (le début, le fin, Is position)

➔ Crée, à partir de position (les éléments correspondants doivent exister), un intervalle de même taille que l'intervalle [début, fin), contenant les différences entre deux éléments consécutifs de ce premier intervalle : l'élément de rang i , hormis le premier, s'obtient en faisant la différence (opérateur $-$) entre l'élément de rang i et celui de rang $i-1$. Le premier élément reste inchangé. Fournit un itérateur sur la fin de l'intervalle créé.

Is **adjacent_difference** (le début, le fin, Is position, fct_diff)

➔ Fonctionne comme la version précédente, en remplaçant l'opération de différence ($-$) par l'appel de la fonction fct_diff.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. **Algorithmes à caractère ensembliste**
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes à caractère ensembliste

bool includes (le début_1, le fin_1, le début_2, le fin_2)

→ Fournit la valeur true si, à toute valeur appartenant à l'intervalle [début_1, fin_1), correspond une valeur égale (==) dans l'intervalle [début_2, fin_2), avec la même pluralité : autrement dit, (si une valeur figure n fois dans le premier intervalle, elle devra figurer au moins n fois dans le second intervalle).

bool includes (le début_1, le fin_1, le début_2, le fin_2, fct_comp)

→ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs.

Algorithmes à caractère ensembliste

Is **set_union** (le début_1, le fin_1, le début_2, le fin_2, Is position)

→ Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant au moins à l'un des deux intervalles [début_1, fin_1) [début_2, fin_2), avec la pluralité maximale : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra $\max(n, n')$ fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments ($==$) devra correspondre aux classes d'équivalence de R . Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé.

Is **set_union** (le début_1, le fin_1, le début_2, le fin_2, Is position, fct_comp)

→ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire `fct_comp` pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles.

Algorithmes à caractère ensembliste

Is **set_intersection** (le début_1, le fin_1, le début_2, le fin_2, Is position)

→ Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant simultanément aux deux intervalles [début_1, fin_1) [début_2, fin_2), avec la pluralité minimale : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra $\min(n, n')$ fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments ($==$) devra correspondre aux classes d'équivalence de R . Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé.

Is **set_intersection** (le début_1, le fin_1, le début_2, le fin_2, Is position, fct_comp)

→ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles.

Algorithmes à caractère ensembliste

Is **set_difference** (le début_1, le fin_1, le début_2, le fin_2, Is position)

→ Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant à l'intervalle [début_1, fin_1) sans appartenir à l'intervalle [début_2, fin_2) ; on tient compte de la pluralité : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra $\max(0, n - n')$ fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments ($==$) devra correspondre aux classes d'équivalence de R . Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé

Is **set_difference** (le début_1, le fin_1, le début_2, le fin_2, Is position, fct_comp)

→ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles.

Algorithmes à caractère ensembliste

Is **set_symetric_difference** (le début_1, le fin_1, le début_2, le fin_2, Is position)

➔ Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant à l'intervalle [début_1, fin_1) sans appartenir à l'intervalle [début_2, fin_2) ou appartenant au second, sans appartenir au premier ; on tient compte de la pluralité : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra $|n-n'|$ fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments ($==$) devra correspondre aux classes d'équivalence de R . Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé.

Is **set_symetric_difference** (le début_1, le fin_1, le début_2, le fin_2, Is position, fct_comp)

➔ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. **Algorithmes de manipulation de tas**
10. Algorithmes divers

Algorithmes de manipulation de tas

void `make_heap` (la début, la fin)

➔ Transforme l'intervalle [début, fin) en un tas, en se fondant sur l'opérateur <.

void `make_heap` (la début, la fin, fct_comp)

➔ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour ordonner le tas.

void `push_heap` (la début, la fin)

➔ La séquence [debut, fin-1) doit être initialement un tas valide. En se fondant sur l'opérateur <, l'algorithme ajoute l'élément désigné par fin-1, de façon que [debut, fin) soit un tas.

void `push_heap` (la début, la fin, fct_comp)

➔ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour ordonner le tas.

Algorithmes de manipulation de tas

void `sort_heap` (la début, la fin)

- ➔ Transforme le tas défini par l'intervalle [debut, fin) en une séquence ordonnée par valeurs croissantes. L'algorithme n'est pas stable, c'est-à-dire que l'ordre relatif des éléments équivalents (au sens de $<$) n'est pas nécessairement Respecté.

void `sort_heap` (la début, la fin, fct_comp)

- ➔ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour ordonner les valeurs

void `pop_heap` (la début, la fin)

- ➔ La séquence [debut, fin) doit être initialement un tas valide. L'algorithme échange les éléments désignés par debut et fin-1 et, en se fondant sur l'opérateur $<$, fait en sorte que [debut, fin-1) soit un tas.

void `pop_heap` (la début, la fin, fct_comp)

- ➔ Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour ordonner le tas.

Les algorithmes

1. Algorithmes d'initialisation de séquences existantes
2. Algorithmes de recherche
3. Algorithmes de transformation d'une séquence
4. Algorithmes de suppression
5. Algorithmes de tri
6. Algorithmes de recherche et de fusion sur des séquences ordonnées
7. Algorithmes à caractère numérique
8. Algorithmes à caractère ensembliste
9. Algorithmes de manipulation de tas
10. Algorithmes divers

Algorithmes divers

nombre `count` (le début, le fin, valeur)

→ Fournit le nombre de valeurs de l'intervalle [début, fin) égales à valeur (au sens de ==).

nombre `count_if` (le début, le fin, prédicat_u)

→ Fournit le nombre de valeurs de l'intervalle [début, fin) satisfaisant au prédicat unaire prédicat_u.

fct `for_each` (le début, le fin, fct)

→ Applique la fonction fct à chacun des éléments de l'intervalle [début, fin) ; fournit fct en résultat.

bool `equal` (le début_1, le fin_1, le début_2)

→ Fournit la valeur true si tous les éléments de l'intervalle [début_1, fin_2) sont égaux (au sens de ==) aux éléments correspondants de l'intervalle de même taille commençant en début_2.

bool `equal` (le début_1, le fin_1, le début_2, prédicat_b)

→ Fonctionne comme la version précédente, en utilisant le prédicat binaire prédicat_b, à la place de l'opérateur ==.

void `iter_swap` (lu pos1, lu pos2)

→ Échange les valeurs des éléments désignés par les deux itérateurs pos1 et pos2.

Algorithmes divers

bool lexicographical_compare (le début_1, le fin_1, le début_2, le fin_2)

→ Effectue une comparaison lexicographique (analogue à la comparaison de deux mots dans un dictionnaire) entre les deux séquences [début_1, fin_1) et [début_2, fin_2), en se basant sur l'opérateur <. Fournit la valeur true si la première séquence apparaît avant la seconde.

bool lexicographical_compare (le début_1, le fin_1, le début_2, le fin_2, prédicat_b)

→ Fonctionne comme la version précédente, en utilisant le prédicat binaire prédicat_b à la place de l'opérateur <.

valeur max (valeur_1, valeur_2)

→ Fournit la plus grande des deux valeurs valeur_1 et valeur_2 (qui doivent être d'un même type), en se fondant sur l'opérateur <.

valeur min (valeur_1, valeur_2)

→ Fournit la plus petite des deux valeurs valeur_1 et valeur_2 (qui doivent être d'un même type), en se fondant sur l'opérateur <.