# Programmation Orientée Objets 2
# Cours 6

Rémi Watrigant[1]

[1] basé sur le cours de Marianne Huchard

# Sommaire du cours 6

1. Présentation
2. Conteneurs
3. Itérateurs
4. Algorithmes
5. Function-object

# Sa petite histoire

STL (pour *Standard Template Library*) est une librairie de classes et fonctions utilitaires :

- à l'origine définie par A. Stepanov et M. Lee chez Hewlett Packard
- adoptée en 1994 par le comité de standardisation de C++

Intérêt :

- portabilité sur les compilateurs C++ respectant la norme *International Standard ISO/IEC 14882, sept. 98*
- intégrée au langage, elle est amenée à en suivre les évolutions
- efficacité des algorithmes
- facilité d'application

# Ses constituants

- des *conteneurs* (collections d'éléments),
- des *itérateurs* (pointeurs généralisés sur les conteneurs),
- des *algorithmes génériques*,
- des *classes-fonction* représentant les fonctions les plus utilisées (destinées à servir de paramètres aux fonctions génériques),
- des *adaptateurs* qui proposent essentiellement des spécialisations des conteneurs, itérateurs et classes-fonctions,
- des *allocateurs* qui permettent de gérer l'espace de stockage.

# Conteneurs

Collections de données paramétrées par :

- le type des éléments stockés
- optionnellement une fonction de comparaison (pour les collections qui trient leurs éléments)
- optionnellement un mode d'allocation

Deux grandes catégories :

- les *séquences*, qui stockent séquentiellement les éléments et y accèdent séquentiellement ou directement suivant le cas,
- les *conteneurs associatifs*, qui associent une clé à chaque objet stocké, ces clés servant à récupérer efficacement les objets.

# Conteneurs

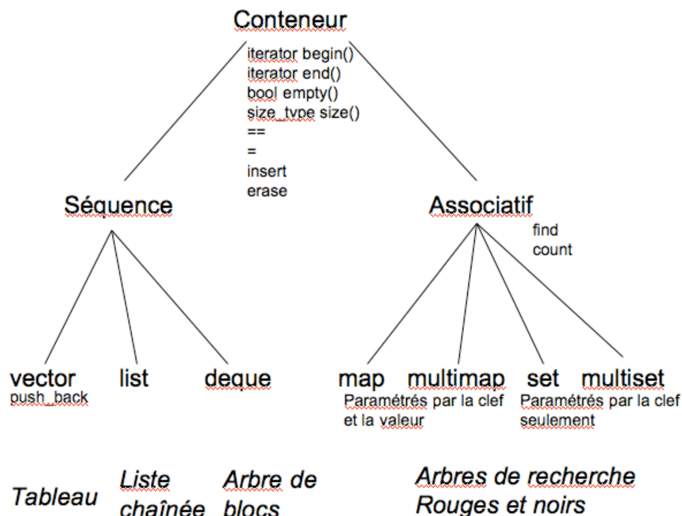


Figure: Les conteneurs avec quelques méthodes

# Conteneurs

Opérations partagées :

- divers constructeurs (dont un sans paramètres),
- `begin()` retourne un itérateur sur la position qui repère le début de la séquence,
- `end()` retourne un itérateur sur la position juste après le dernier élément,
- `size()` qui retourne le nombre d'éléments stockés,
- `bool empty()` qui retourne vrai ssi le conteneur est vide,
- divers opérateurs de comparaison

Spécifications accessibles depuis de nombreux sites

- http://www.cplusplus.com
- http://www.cppreference.com (français!)

# Conteneurs

Autres opérations assez communément partagées, avec des signatures et des effets parfois différents suivant le conteneur :

- insertion d'éléments avec `insert(...)`, `push_back(...)`, `push_front(...)`,
- effacement d'éléments avec `erase(...)`, `clear(...)`, `pop_back()`, `pop_front()`,
- accès avec `operator[](int)`, `at(int)`, `front()`, `back()`.

# Séquences : `vector`

Principe :

- Implémentation à l'aide d'un bloc de mémoire contigu, semblable à un tableau
- s'agrandit automatiquement lorsque sa taille n'est pas suffisante (recopies internes)

Complexité des opérations :

- ajout et effacement à la fin en temps constant si on considère la complexité amortie
- ajout et effacement internes linéaires dans la taille du tableau (recopies internes)

# Séquences : vector

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

int main()
{
  vector<int> v;
  v.push_back(4);
  v.push_back(7);
  v.push_back(3);
  v.push_back(8);

  cout << "taille=" << v.size() << endl;

  cout <<"------ affichage  du vecteur --------------"<< endl;
  for (int ii=0; ii<v.size(); ii++)
    cout << v[ii] << " ";
}
```

# Séquences : `list`

Principe :

- implémentée à l'aide d'un ensemble de cellules doublement chaînées
- un peu plus de place occupée que le vecteur pour un même ensemble d'éléments

Complexité des opérations :

- ajout et effacement en temps constant à n'importe quelle position

# Séquences : list

```
#include<iostream>
#include<list>

int main()
{
  list<float> liste;
  liste.push_back(4);
  liste.push_back(7);
  liste.push_back(3);
  liste.push_back(8);
  cout << "taille liste=" << liste.size() << endl;
}
```

# Séquences : deque

Principe :

- queues à double entrée
- implémentées à l'aide d'un tableau qui référence $n$ tableaux secondaires stockant les éléments eux-mêmes. Dans les tableaux secondaires les éléments sont insérés à partir du centre. Lorsqu'un tableau secondaire est rempli (au moins du centre vers une extrémité), un autre tableau secondaire est créé. Les tableaux secondaires sont insérés dans le tableau principal en partant aussi du centre.

Complexité :

- ajout en début et en fin en temps constant
- moins d'espace occupé que la liste
- insertion au milieu linéaire

# Séquences : deque

```
#include<iostream.h>
#include<deque>

int main()
{
  deque<int> li;
  li.push_back(4);
  li.push_back(7);
  li.push_back(3);
  li.push_back(8);
  li.pop_front();

  cout << "-------- deque -------------"<< endl;
  for (int d=0; d<li.size(); d++)
    cout << li[d] << " ";
  cout << endl;
}
```

# Conteneurs associatifs

Principes généraux

- Eléments identifiés (et récupérés) par une clé
- implémentés à l'aide d'arbres rouges et noirs (arbres de recherche équilibrés)
- insertion et récupération sont en $O(log(n))$

Conteneurs existants

- `map`, `multimap`
- `set`, `multiset`

# Associatifs : `map`

- clés et valeurs distinctes
- clés uniques
- paramétrée par le type des clés, le type des valeurs et optionnellement par la fonction de comparaison des clés et un allocateur
- contient des instances de la classe paramétrée `pair`, instanciée avec le type des clés et le type des valeurs

# Associatifs : map

```
int main()
{
  map<string,Date > Anniversaires;
  Anniversaires["Julie"]=Date(8,5,1988);
  Anniversaires["Cecilia"]=Date(18,12,1991);
  pair<string, Date> nouvelleEntree;
  nouvelleEntree.first = "Eloise";
  nouvelleEntree.second = Date(4, 12, 1998);
  Anniversaires.insert( nouvelleEntree );
}
```

# Associatifs : multimap

Variante dans laquelle les clefs sont multiples

```
#include<iostream>
#include<string>
#include<map>
#include<algorithm>
#include <pair.h>
// .......... avec la meme classe Date
void main()
{
  multimap<string,Date> AnniversairesBis;
  AnniversairesBis.insert(pair<const string,Date>
   ("Julie",Date(8,5,1988)));
  AnniversairesBis.insert(pair<const string,Date>
   ("Cecilia",Date(18,12,1991)));
  AnniversairesBis.insert(pair<const string,Date>
   ("Eloise", Date(11,4,1998)));
  AnniversairesBis.insert(pair<const string,Date>
   ("Julie",Date(12,3,1994)));
}
```

# Associatifs : `set` et `multiset`

- clés et valeurs sont confondues
- `set` ensembles sans répétition des éléments
- `multiset` ensembles avec répétition des éléments (famille).

# Itérateur

- pointeur généralisé sur un conteneur
- objet qui retourne successivement une référence vers chaque élément du conteneur
- utilisé par de nombreux algorithmes appliqués aux conteneurs

# Insérer et afficher les éléments d'un vecteur/d'un ensemble

```
#include<iostream>
#include<vector>
#include<set>
using namespace std;
void main(){
  vector<int> v;
  vector<int>::iterator i;
  v.insert(v.end(), 4); v.insert(v.end(),7);
  v.insert(v.end(),3);  v.insert(v.end(),8);

  cout <<"------ affichage du vecteur -------------"<< endl;
  for (i=v.begin(); i<v.end(); i++)
    cout << *i << " ";
  cout << "taille=" << v.size() << endl;

  cout <<"------ affichage du vecteur -------------"<< endl;
  vector<int>::iterator iter=v.begin();
  for (int ii=0; ii<v.size(); ii++)
    cout << iter[ii] << " ";
......}
```

# Insérer et afficher les éléments d'un vecteur/d'un ensemble

```
void main()
{
.....

  set<int> s;
  set<int>::iterator is;
  s.insert(s.end(),4);
  s.insert(s.end(),7);
  s.insert(s.end(),3);
  s.insert(s.end(),8);

  cout <<"------ affichage de l'ensemble --------------"<< endl;
  for (is=s.begin(); is!=s.end(); is++)
    cout << *is << " ";
  cout << endl;
}
```

# Accéder aux éléments d'une map

```
void main()
{
.... // suite du programme définissant la map anniversaires

    cout<<"--- acces a la cle et la valeur d'un element ---"<<endl;

    map<string, Date>::iterator annifJulie = anniversaires.find("Julie");

    if (annifJulie != anniversaires.end()) {
         cout << "annif de Julie : " << annifJulie->second << endl;
    }
}
```

# Algorithmes : présentation

*# include <algorithm>*

La librairie offre une grande variété d'algorithmes :

- numériques (min, max, sommes partielles, accumulations, produits, etc.)
- tris
- modifiant les conteneurs (remplissage, copie, échanges, union, intersection, etc.)
- sans modification (recherche, application de fonction, inclusion, etc.)

Les algorithmes ne sont pas écrits spécifiquement pour tel ou tel conteneur, mais prennent comme paramètres des itérateurs et des fonctions.

# Exemple d'algorithme numérique (accumule)

Appliquer cumulativement d'une fonction binaire $f$ aux éléments du conteneur.
Par défaut cette fonction est l'addition.
Exemple : calcule $v[3] + (v[2] + (v[1] + (v[0] + 0)))$.

```
#include<iostream.h>
#include<vector>
#include<algorithm>
#include<set>
void main(){
  vector<int> v;
  v.insert(v.end(), 4); v.insert(v.end(),7);
  v.insert(v.end(),3);  v.insert(v.end(),8);
  cout << accumulate(v.begin(),v.end(),0) << endl;

  set<int> s;
  s.insert(s.end(),4);  s.insert(s.end(),7);
  s.insert(s.end(),3);  s.insert(s.end(),8);
  cout << accumulate(s.begin(),s.end(),0) << endl;
}
```

# Exemple d'algorithme numérique (accumule)

Appliquer cumulativement d'une fonction binaire $f$ aux éléments du conteneur.
Par défaut cette fonction est l'addition.
Exemple : calcule $v[3] * (v[2] * (v[1] * (v[0] * 1)))$.

```
#include<iostream.h>
#include<vector>
#include<algorithm>
#include<set>
void main(){
  vector<int> v;
  v.insert(v.end(), 4); v.insert(v.end(),7);
  v.insert(v.end(),3);  v.insert(v.end(),8);
  cout << accumulate(v.begin(),v.end(),1,multiplies<int>()) << endl;

  set<int> s;
  s.insert(s.end(),4);  s.insert(s.end(),7);
  s.insert(s.end(),3);  s.insert(s.end(),8);
  cout << accumulate(s.begin(),s.end(),1,multiplies<int>()) << endl;
}
```

# Exemple de Tri (quicksort en nlog(n))

`less<>` est utilisée par défaut

```
... // ----- tri du vecteur
  sort(v.begin(),v.end());
... // ----- tri de la queue
  sort(li.begin(),li.end());
```

# Exemple d'algorithme modificateur (intersection d'ensembles)

```
vector<int> v1;
v1.push_back(2);
v1.push_back(3);
v1.push_back(4);

vector<int> v2;
v2.push_back(3);
v2.push_back(6);
v2.push_back(7);

vector<int> v3( 6 );      // 0 0 0 0 0 0
vector<int>::iterator finIntersection =
   set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());

cout << "v3 = ";
for (vector<int>::iterator i = v3.begin() ; i != finIntersection ; i++) {
    cout << *i << " ";
}
```

# Exemple d'algorithme non modificateur (application d'une fonction à tous les éléments)

La séquence n'est pas modifiée ... mais l'état de ses éléments peut l'être.

```
void ajoute2(int& i){
    i+=2;
}

int main() {

    vector<int> v;
    v.push_back(2) ; v.push_back(32) ; v.push_back(4);

    for_each(v.begin(),v.end(),ajoute2);

}
```

# Function-object

Ce sont des classes munies d'un opérateur `operator()`. Elles généralisent la notion de fonction.

```
class comparePersonnes : public binary_function<Personne, Personne, bool>  {
    bool operator()(Personne p1, Personne p2) const {
        return p1.getAge() < p2.getAge();
    }
};

template<class comparaisonFunction>
void trier( vector<Personne> liste, comparaisonFunction c) {
    //ici on trie avec la fonction de comparaison, à utiliser comme suit :
    // c( p1, p2 );
}

int main(int argc, char** argv) {

    vector<Personne> liste;
    //remplissage du vecteur

    trier<comparePersonnes>(liste, comparePersonnes());

}
```

# Classes existantes

Function-objects unaires ou binaires
Exemples de Function-object: `plus`, `minus`, `multiplies`, `divides`, `...`, `less`, `greater`, `...`
Extrait du code de plus:

```
template <typename T>
class plus : public binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const { return x + y; }
};
```

Elles sont très utilisées pour passer des paramètres aux algorithmes ou aux classes paramétrées (on ne peut pas passer d'opérateur comme paramètre).

# Utilisation avec bind2nd

`bind2nd` est une fonction qui :

- prend en entrée une Function-object binaire et une valeur
- retourne une fonction unaire qui applique la fonction binaire au paramètre et à la valeur

Exemple : `bind2nd(less<int>(),5)` produit une fonction unaire qui répond vrai si son argument est inférieur à 5.
Exemple : on veut par exemple remplacer dans le vecteur toutes les valeurs inférieures à 5 par 5

```
replace_if(v.begin(),v.end(),bind2nd(less<int>(),5),5);
```

# Classes définies par l'utilisateur

On peut aussi en définir de nouvelles, par exemple un équivalent de `ajoute2`:

```
class Ajoute2 : public unary_function<int,int>
{public:
  Ajoute2(){}
  virtual int operator()(int& i){i+=2; return i;}
};
........
// exemple d'utilisation directe
void main(){
Ajoute2 x; int i=28;
 cout<<" i avant: "<<i<<endl;
 cout<<"retour de Ajoute2: "<<x(i)<<endl;
 cout<<"i apres: "<<i<<endl;
}
//exemple d'utilisation en paramètre d'un algorithme
  for_each(li.begin(),li.end(),Ajoute2());
```