

HLIN603 : Programmation Objet Avancée

STL : la librairie standard C++

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

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.

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

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.

On distingue parfois les conteneurs associatifs des ensembles ; on range alors les deux dans les conteneurs triés

Conteneurs

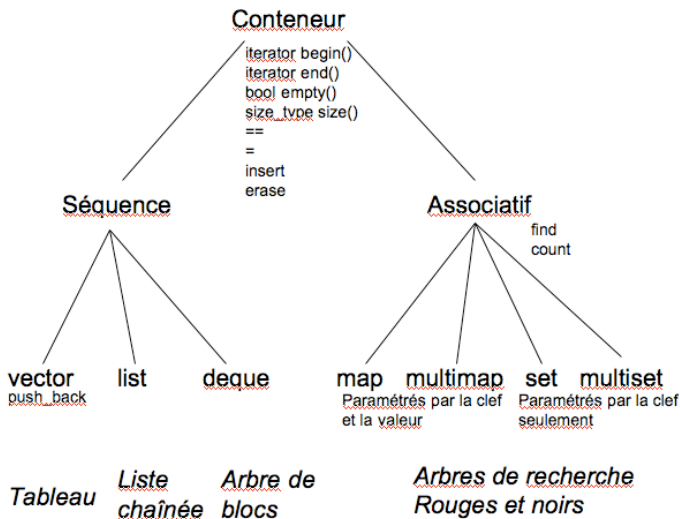


Figure : Les conteneurs avec quelques méthodes

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

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

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

Conteneurs existants

- map, multimap
- set, multiset
- hash_ versions dans SGI-STL
(<http://www.sgi.com/tech/stl/>)

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

```
#include<iostream>
#include<string>
#include<map>
#include<algorithm>
#include <pair.h>
class Date{
private:  int jour, mois, annee;
public:
    Date(int j=1, int m=1, int a=2001){jour=j; mois=m; annee=a;}
    virtual ~Date(){}
    virtual void affiche(ostream& os)const
    {os<<jour<<"/"<<mois<<"/"<<annee;}
};

ostream& operator<<(ostream& os, const Date& D)
{D.affiche(os); return os;}
ostream& operator<<(ostream& os,const pair<const string,Date>&p)
{os<<p.first<<" "<<p.second; return os;}
```

Associatifs : map

```
int main()
{
    map<string,Date > Anniversaires;
    Anniversaires["Julie"]=Date(8,5,1988);
    Anniversaires["Cecilia"]=Date(18,12,1991);
    Anniversaires.insert(
        pair<const string,Date>("Eloise", Date(11,4,1998)));
}
```

Associatifs : multimap

Variante dans laquelle les clefs sont multiples

```
#include<iostream>
#include<string>
#include<multimap>
#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).

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

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

Itérateurs

Cinq sortes :

- les `input iterator`, qui autorisent le déréférencement (`*` et `->`) pour la lecture seule, l'incrémement (`++`), les comparaisons (`==` et `!=`),
- les `output iterator`, qui autorisent le déréférencement (`*` et `->`) pour l'écriture seule, et l'incrémement (`++`),
- les `forward iterator`, qui autorisent en plus des opérations précédentes l'affectation (`=`),
- les `bidirectionnal iterator`, qui autorisent en plus des opérations précédentes la décrémement (`--`),
- les `random iterator`, qui autorisent en plus des opérations précédentes l'accès direct à une position (`[]`) les comparaisons (`<`, `>`, `>=`, `<=`), et l'avancée ou le recul de n positions.

Itérateurs

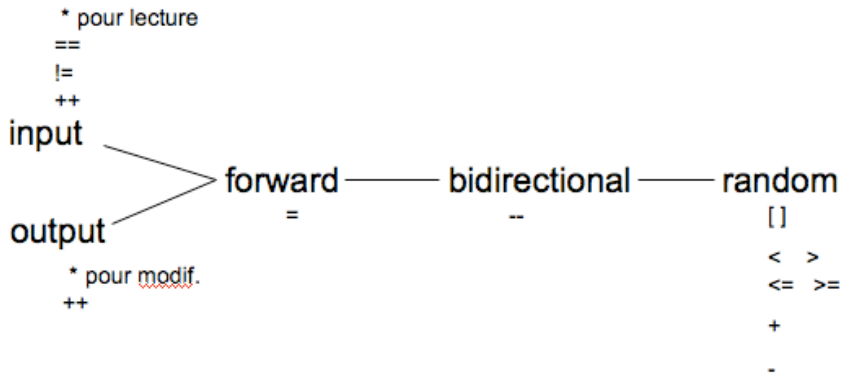


Figure : Les itérateurs et leurs opérations

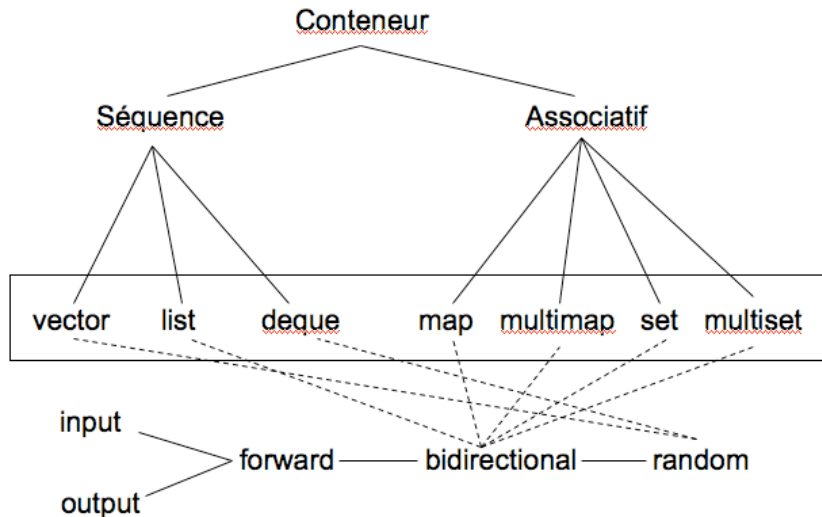
Liens entre itérateurs et conteneurs

Dans le conteneur `C::iterator` ou `C::const_iterator`.

- les `input iterator` et les `output iterator` servent de base pour les itérateurs de *flot* : `istream_iterator`, `ostream_iterator` ;
- les `bidirectionnal iterator` servent pour les *conteneurs associatifs* et les *listes*, un accès direct par le numéro d'élément ne serait en effet pas efficace dans une telle structure et il n'est donc pas disponible ;
- les `random iterator` servent pour les *vecteurs* et les *queues à double entrée (deque)*.

Liens entre itérateurs et conteneurs

Dans le conteneur `C::iterator` ou `C::const_iterator`.



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; // est un random iterator
    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; // est un bidirectionnal iterator
    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

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

// la fonction find retourne Anniversaires.end()
// si la cle n'est pas trouvee
// ce pourrait etre teste ici ...

    cout << (*(Anniversaires.find("Julie"))).first<<" "
        << (*(Anniversaires.find("Julie"))).second <<endl;
}
```

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

Présentation

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<algo.h>
#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.

On souhaite utiliser la multiplication.

Exemple : calcule $v[3] * (v[2] * (v[1] * (v[0] * 1)))$.

```
#include<iostream.h>
#include<vector>
#include<algo.h>
#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 $n \log(n)$)

`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)

```
cout <<"----- intersection de deux ensembles -----" << endl;
set<int> s1;
s1.insert(s1.end(),4); s1.insert(s1.end(),7);
s1.insert(s1.end(),3); s1.insert(s1.end(),8);
s1.insert(s1.end(),8); // sans effet !

cout << "s1 = "; ostream_iterator<int> outInt(cout, " ");
copy(s1.begin(),s1.end(),outInt); //s1 = 3 4 7 8
cout << endl;

set<int> s2;
s2.insert(s2.end(),4); s2.insert(s2.end(),9);
s2.insert(s2.end(),5); s2.insert(s2.end(),8);
cout << "s2 = ";
copy(s2.begin(),s2.end(),outInt); //s2 = 4 5 8 9
cout << endl;
```

Exemple d'algorithme modificateur (intersection d'ensembles)

```
.....  
    vector<int> s3(4);  
    vector<int>::iterator fin  
    = set_intersection(s1.begin(),s1.end(),  
                      s2.begin(),s2.end(),s3.begin());  
    cout << "partie utile de s3 = ";  
    copy(s3.begin(),fin,outInt); // partie utile de s3 = 4 8 taille=4  
    cout << "taille="<< s3.size()<<endl;  
    cout << "integralite de s3 = ";  
    copy(s3.begin(),s3.end(),outInt); //integralite de s3 = 4 8 0 0  
    cout << endl;
```

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;}
```

```
..... // dans le main ..... pour la queue precedente .....
        // chaque élément est augmenté de 2
```

```
for_each(li.begin(),li.end(),ajoute2);
```

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

Classes existantes

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

Il existe deux classes générales principales :

- `unary_function`, qui définit deux types associés `argument_type` et `result_type`
- `binary_function`, qui définit trois types associés `first_argument_type`, `second_argument_type` et `result_type`

Classes existantes

Les autres sont des sous-classes de celles-là, par exemple 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` prend comme arguments une fonction binaire et une valeur du même type que le deuxième argument de la fonction binaire.

Son effet est de fixer la valeur de ce deuxième argument.

Exemple : `bind2nd(less<int>(),5)` produit une fonction unaire qui répond vrai si son argument est inférieur à 5.

à partir de $(x,y) \rightarrow x < y$, crée $x \rightarrow x < 5$

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());
```

Sommaire

Présentation

Conteneurs

Itérateurs

Algorithmes

Classes-Fonctions

Adaptateurs

Adaptateur

Ce sont des spécialisations des conteneurs, des itérateurs et des classes fonctions.

- Adaptateurs de conteneurs

- `stack`
- `queue`
- `priority_queue`

- Adaptateurs d'itérateurs

Ils permettent de modifier le comportement des itérateurs existants en obligeant des insertions toujours en tête ou en queue d'un conteneur, ou en parcourant le conteneur de la fin vers le début, plutôt que dans le sens habituel qui est inverse.

Adaptateurs de classes-fonctions

not1 et not2 permettent d'obtenir la négation d'un prédicat unaire ou binaire.

ptr_fun permet de transformer une fonction ordinaire en instance de classe-fonction sans effort de programmation.

```
int affiche2(int i)
{
    cout << i+2;
    return i;
}
```

```
// ... dans le main, sur la deque li ...
for_each(li.begin(),li.end(),ptr_fun(affiche2));
```