TP : La librairie *STL*

(extrait d'un support de François Fleuret et Julien Ponge)

Partie 1 :

La *Standard Template Library* est une librairie du C++ qui évite au programmeur d'avoir à réinventer la roue. Une description complète de cette librairie existe à l'adresse: **http://www.sgi.com/tech/stl/**. Comme son nom l'indique, la *STL* se base sur la notion de **template**. Il existe deux types de *templates*: de classe ou de fonction. Dans les deux cas, leur but est de permettre de spécifier des classes ou des fonctions de manière générique. Ceci permettra alors au programmeur de les utiliser avec n'importe quel type de données. La librairie *STL* est divisée en trois parties distinctes:

- Les conteneurs
- Les itérateurs
- Les algorithmes

Exemple d'utilisation:

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main(int argc, char **argv) {
7 vector<int> v;
8
9 for(int i=0; i<10; i++)
10 v.push_back(i);
11
12 reverse(v.begin(), v.end());
13
14 vector<int>::iterator it;
15 for(it=v.begin(); it!=v.end(); it++)
16 cout << *it << " ";
17 cout << endl;
18 }
```

1) Conteneurs

Un *conteneur* est un objet permettant de stocker des données d'un certain type, déterminé par l'utilisateur lors de la création du conteneur. Il existe deux sortes principales de conteneurs: **séquentiels** ou **associatifs**.

Conteneurs séquentiels

Les valeurs d'un conteneur séquentiel sont ordonnées en fonction leur insertion dans le conteneur. Les conteneurs séquentiels sont : *vector, list et deque*

Conteneurs associatifs

Les conteneurs associatifs autorisent un accès à leurs données à l'aide d'une clé autre qu'un nombre entier : *set, multiset, map, multimap*

2) Itérateurs

Les itérateurs permettent de parcourir les conteneurs un peu comme on le ferait avec un pointeur sur un tableau. Ainsi, on peut parcourir un à un les éléments d'un conteneur. Le type d'élément qu'un itérateur va parcourir est spécifié à la déclaration de l'itérateur. Dans l'exemple introductif :

```
14 vector<int>::iterator it;
15 for(it=v.begin(); it!=v.end(); it++)
16 cout << *it << " ";
```

Crée un itérateur *it* sur un vecteur d'entiers, parcourt le vecteur *v* élément par élément et affiche la valeur de chaque élément. Les méthodes begin() et end() retournent des itérateurs correspondant au début et à la fin du vecteur.

3) Algorithmes

La STL fournit un certain nombre d'algorithmes permettant d'effectuer des opérations sur les conteneurs, telles que le tri ou l'inversion des valeurs. Ces algorithmes n'agissent en fait pas directement sur les conteneurs, mais sur des itérateurs, afin d'assurer la généricité sur n'importe quel type de conteneur.

Dans l'exemple introductif :

3 #include <algorithm>
12 reverse(v.begin(), v.end());

Inverse l'ordre des valeurs du vecteur *v*. Les méthodes begin() et end() retournent des itérateurs correspondant au début et à la fin du vecteur.

4) Méthodes communes aux conteneurs

- iterator **begin**(): retourne un itérateur qui pointe sur le début du conteneur.
- iterator **end**(): retourne un itérateur qui pointe après le dernier élément du conteneur.
- size type **size**() const: retourne la taille du conteneur.
- bool **empty**() const: retourne true si la taille du conteneur est 0.
- bool operator<(const container&, const container&): comparaison lexicographique de deux conteneurs
- container& operator**=**(const container&):opérateur d'assignation.
- bool operator**==**(const container&, const container&): teste l'égalité entre deux conteneurs.

5) La classe vector

La classe vector permet de représenter des vecteurs d'éléments. Cette classe est très similaire aux tableaux. Les éléments d'un vecteur sont tous du même type, mais le type des éléments peut varier d'un vecteur à l'autre. On peut accéder à n'importe quel élément du vecteur, mais l'insertion et l'effacement d'un élément sont optimisés pour être effectués à la fin du vecteur.

Méthodes de vector

- *void push_back(const Type &):* ajoute un élément de type *Type* à la fin du vecteur.
- *void pop_back():* enlève le dernier élément du vecteur.
- *Type& operator[](size type n):* retourne le n-ième élément du vecteur.

Exemple

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 int main(int argc, char **argv) {
5 int i;
6 vector<int> v1,v2;
7
8 for(i=0; i<10; i++){
9 v1.push_back(i);
10 v2.push_back(i);
11 }
12
13 cout << v1.size() << endl;
14
15 for(i=0; i<v2.size(); i++)
16 cout << v2[i] << << " ";
```

```
17 cout << endl;
18
19 if(v1 == v2)
20 v2.pop_back();
21
22 for(i=0; i<v2.size(); i++)
23 cout << v2[i] << " ";
24 cout << endl;
25 }
25
```

Le programme précédent affiche :
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8

Exercice 1 :
Ecrire un programme qui crée un vecteur *v1* de 10 *float* régulièrement répartis entre 0.0 (inclu) et 1.0 (non compris), créée un nouveau vecteur *v2* égal à *v1* et efface élément par élément la deuxième moitié de *v2*.

6) **La classe** list

La classe list représente les listes doublement chaînées, c'est-à-dire une séquence que l'on peut parcourir dans les deux sens. Une liste n'autorise pas l'accès direct à n'importe quel él ément (à l'inverse de vector), mais elle est optimisée pour l'insertion et l'effacement d'éléments à n'importe quelle position.

Méthodes de list

- void **push_back**(const Type&): ajoute un élément de type
- *Type* à la fin de la liste.
- void **pop_back**(): enlève le dernier élément de la liste.
- void **push_front**(const Type&): ajoute un élément de type*Type* au début de la liste.
- void **pop_front**(): enlève le premier élément de la liste.
- iterator **erase**(iterator pos): efface l'élément à la position *pos*.
- iterator **erase**(iterator first, iterator last): efface les élément entre les positions *first* et *last*.
- iterator **insert**(iterator pos, const Type& x): insert l'élément *x* à la position précédant *pos*.
- void **insert**(iterator pos, InputIterator f, InputIterator l): insert les éléments compris entre *f* et *l* à la position précédant *pos*.

Exemple

```
1 #include <iostream>
2 #include <list>
3 using namespace std;
4 int main(int argc, char **argv) {
5 int i;
6 list<int> l1, l2;
7 list<int>::iterator it;
8
9 for(i=0; i<10; i++){
10 l1.push_front(i);
11 l2.push_back(i);
12 }
```

```
13 for(it=l1.begin(); it!=l1.end(); it++) cout << *it << " ";
14 cout << endl;
15 for(it=l2.begin(); it!=l2.end(); it++) cout << *it << " ";
16 cout << endl;
17
18 if(l2 < l1){
19 it = l1.begin();
20 for(i=0; i<4; i++) it++;
21 l1.insert(it, l2.begin(), l2.end());
22 }
23 for(it=l1.begin(); it!=l1.end(); it++) cout << *it << " ";
24 cout << endl;
25 }
33
```

Le programme précédent affiche
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
9 8 7 6 0 1 2 3 4 5 6 7 8 9 5 4 3 2 1 0

Exercice 2
Ecrire un programme qui crée une liste *l1* contenant les 10 premières lettres de l'alphabet, créer une nouvelle liste *l2* égale à *l1* et effacer d'un bloc la première moitié de *l2*.

7) Les ensembles : set

Chaque valeur d'un ensemble est unique.

- Méthodes utiles:
  - insert(val), erase(val)
  - find(val) (renvoie un itérateur ...)
  - clear()
  - size(), empty().
- Opérateurs: =, ==, <. Les opérations sont toutes en O(n) (ajout, suppression, accès).
- En interne, les éléments sont toujours triés par ordre croissant.

Exemple :

```
1 #include <set>
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6 set<int> ens;
7 ens.insert(1); ens.insert(1);
8 ens.insert(2); ens.insert(2);
9 ens.erase(2);
10 // Affiche '1'
11 cout << ens.size() << endl;
12 return 0;
13 }
```

8) Les dictionnaires : map

- Association clé/valeur (clés uniques).
- Méthodes utiles:
  - size(), empty(), clear()
  - insert(val), erase(val), find(val).
- Opérateurs: =, ==, <, [].

● '[]' va nous permettre d'accéder à un élément par sa clé. Il va aussi nous permettre l'ajout.

Exemple :

```
1 #include <map>
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6 // Paramétre 1: type clé, paramètre 2: type valeur
7 map<string, string> agenda;
8 agenda[“Aristide”] = “06-12-34-56-78”;
9 agenda[“Paul”] = “03-86-38-12-34”;
10 cout << “Numéro de Paul: “ << agenda[“Paul”]<< endl;
11 return 0;
12 }
```

Remarque : 'set' garanti l'unicité des valeurs, 'map' garantie l'unicité des clés. Si l'on veut au contraire des valeurs ou clés multiples, il faut employer 'multiset' et 'multimap'. Les en-têtes sont les mêmes que 'set' et 'map' et l 'emploi est identique.

Exercice 3

Ecrire un programme qui crée un ensemble de nom d'élèves de la classe. Afficher les éléments de l'ensemble. Supprimer les 2 premiers noms suivant l'ordre lexicographique, puis afficher de nouveau l'ensemble.

Exercice 4

Ecrire un programme qui crée un dictionnaire avec des noms et des numéros de téléphone, le numéro de téléphone étant représenté par un entier (le premier 0 n'est pas stocké). Afficher le dictionnaire. Supprimer tous les numéros commençant par 5 et afficher votre dictionnaire.

9) Quelques algorithmes

Remarques

Les définitions des algorithmes suivants utilisent différents types d'itérateurs. Pour simplifier le problème, nous les considérerons tous comme des itérateurs généraux, notés *Itérateur*.

- *Itérateur* **find**(*Itérateur* first, *Itérateur* last, const *Type*& value); //Permet de trouver la position de la valeur *value* entre les positions *first* et *last*.

```
1 vector<int> x;
2 vector<int>::iterator it;
3
4 x.push_back(1); x.push_back(7); x.push_back(3);
5
6 it = find(x.begin(), x.end(), 7);
```

- *Itérateur* **search**(*Itérateur* first1, *Itérateur* last1, *Itérateur* first2, *Itérateur* last2); //Permet de trouver la position de la sous-séquence comprise entre *first2* et *last2* à l'intérieur de la séquence comprise entre *first1* et *last1*.

```
1 vector<int> v1, v2;
2 vector<int>::iterator it;
3
4 for(int i=0; i<10; i++)
5 v1.push_back(i);
6
```

```
7 v2.push_back(1); v2.push_back(2); v2.push_back(3);
8
9 it = search(v1.begin(), v1.end(), v2.begin(), v2.end());
```

- void **random shuffle**(*Itérateur* first, *Itérateur* last); //Réarrange de manière aléatoire les éléments entre *first* et *last* à l'intérieur d'un conteneur.

```
1 vector<int> v;
2
3 for(int i=0; i<10; i++)
4 v.push_back(i);
5
6 random_shuffle(v.begin(), v.end());
```

- void **sort**(*Itérateur* first, *Itérateur* last); //Trie par ordre croissant les éléments entre *first* et *last* à l'intérieur d'un conteneur.

```
1 vector<int> v;
2
3 for(int i=0; i<10; i++)
4 v.push_back(int(drand48()*100));
5
6 sort(v.begin(), v.end());
```

Exercice 5

Ecrire un programme qui crée un vecteur de 5 string de longueurs différentes, affiche ces chaînes, les trie par ordre lexicographique et réaffiche les chaînes triées.

Exercice 6

Ecrire un programme qui crée une liste de string représentant la phrase "il fait beau", trouver la position du mot "beau" et insérer le mot "tres" à la position précédente.

Partie 2 (question joker):

L'objectif est de définir une représentation des matrices creuses de réels efficace en utilisant la STL (une matrice creuse est une matrice constituée essentiellement de 0 et dont on ne stocke que les valeurs différentes de 0).

1) Créer une classe vecteur creux. On pourra utiliser une « map » pour représenter les éléments d'un vecteur non nuls.
    a. Définir les constructeurs intéressants.
    b. Définir les opérateurs [] et () permettant d'accéder au ième élément d'un vecteur. On s'intéressera entre au cas où l'on passe l'indice d'un élément du vecteur égal à 0 et donc en théorie non stocké.
    c. Définir l'opérateur <<
    d. Définir un ensemble d'opérateur de calcul : *, -, +=, etc
2) Définissez la classe matrice creuse en utilisant votre classe vecteur et la STL. On pourra définir la matrice creuse comme une « map » de vecteur non nul.
    a. Définir les constructeurs intéressants.
    b. Définir les opérateurs [] et () permettant d'accéder à l'élément à la i ème ligne et j ième colonne de la matrice. On s'intéressera entre au cas où l'on passe les indices d'un élément de la matrice égal à 0 et donc en théorie non stocké.
    c. Définir l'opérateur <<
    d. Définir des opérateurs de calcul : -, +=, * (on pourra considérer différentes multiplications vecteur-matrice, matrice-matrice, floattant-matrice, etc), etc