

Informatique III: Programmation en C++

Lundi 19 Décembre 2005

1

Rappel: string

- ➡ `char *c_str()` retourne un pointeur `char *` sur le tableau de caractères constituant la chaîne ;
- ➡ `size_type find(const string& str, size_type pos)` retourne l'indice où se trouve la chaîne recherchée à partir de `pos`, sinon retourne `string::npos`.
- ➡ `string& replace(size_type pos, size_type n, const string& str)` remplace la partie de la chaîne partant de la position `pos` et de longueur `n` par la chaîne `str`.

3

Rappel: fstream

- ➡ `void open(const char *nom_fichier, openmode mode)` ouvre le fichiers `nom_fichier` dans le mode `mode` (ifstream: par défaut en lecture, ofstream: par défaut en écriture).
- ➡ `bool good()` vérifie si le fichier prêt pour des opérations de lecture-écriture.
- ➡ `void close()` ferme le fichier.

Rappel: ofstream

- ➡ `ostream& write(const char* str, taille n)` écrit `n` caractères de `str` dans le fichier.

2

```
1 void remplacerChaine(string nomEntree, string nomSortie,  
2                       string chaine, string nouvelleChaine){  
3     string s;  
4     ifstream fin; ofstream fout;  
5  
6     fin.open(nomEntree.c_str());  
7     if(fin.good()){  
8         fout.open(nomSortie.c_str());  
9         while(fin.good()){  
10            getline(fin,s);  
11            int pos = s.find(chaine,0);  
12            while(pos != string::npos){  
13                s.replace(pos, chaine.length(), nouvelleChaine);  
14                pos = s.find(chaine, pos+nouvelleChaine.length());  
15            }  
16            fout.write(s.c_str(), s.length());  
17            fout.write("\n",1);  
18        }  
19        fin.close();  
20        fout.close();  
21    } else cout << "Fichier " << nomEntree << " inexistant." << endl;  
22 }
```

4

```
1 int main (int argc, char **argv) {
2     string ch, nouvelleCh;
3     string nameIn, nameOut;
4
5     cout << "Entrer un nom de fichier d'entree: ";
6     cin >> nameIn;
7     cout << "Entrer une chaine a remplacer: ";
8     cin >> ch;
9     cout << "Entrer une chaine de remplacement: ";
10    cin >> nouvelleCh;
11    cout << "Entrer un nom de fichier de sortie: ";
12    cin >> nameOut;
13
14    remplacerChaine(nameIn, nameOut, ch, nouvelleCh);
15 }
```

5

La librairie *STL*

La *Standard Template Library* est une librairie du C++ qui évite au programmeur d'avoir à réinventer la roue.

Nous ne ferons ici qu'un survol de cette librairie afin de vous permettre d'utiliser quelques unes de ses fonctionnalités.

Les personnes intéressées peuvent trouver toute la documentation nécessaire à l'adresse: <http://www.sgi.com/tech/stl/>.

7

Standard Template Library

6

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.

Nous ne nous intéresserons pas plus en détail à cette notion de *template*.

8

La librairie *STL* est divisée en trois parties distinctes:

- ➡ Les conteneurs
- ➡ Les itérateurs
- ➡ Les algorithmes

9

Exemple introductif

```
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 }
```

10

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** et **associatifs**.

Aujourd'hui nous ne traiterons que de deux exemples de conteneurs séquentiels.

11

Conteneurs séquentiels

Les valeurs d'un conteneur séquentiel sont ordonnées en fonction leur insertion dans le conteneur.

- ➡ vector
- ➡ list
- ➡ deque

12

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

13

Itérateurs

Nous avons vu qu'il était possible de parcourir des tableaux en utilisant les pointeurs.

Les itérateurs nous permettent de faire de même pour les conteneurs. Ainsi, nous pouvons 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.

Il existe plusieurs types d'itérateurs, mais nous ne considérerons par la suite que le cas général.

15

Dans l'exemple introductif

```
2 #include <vector>
```

```
7 vector<int> v;
```

Déclare un objet *v* comme un conteneur de type *vector*, contenant des entiers.

14

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.

16

Algorithmes

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.

17

Quelques conteneurs

19

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.

18

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.

20

- ➡ `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.

21

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.

23

La classe `vector`

La classe `vector` permet de représenter des vecteurs d'éléments. Cette classe est très similaire aux tableaux vus précédemment lors du cours. 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.

22

Exemple

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
```

24

```

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

```

10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8

```

26

Exercice

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

27

Correction

```

1 int main(int argc, char **argv) {
2     int i;
3     vector<float> v1,v2;
4
5     for(i=0; i<10; i++){
6         v1.push_back(0.1*i);
7     }
8
9     v2 = v1;
10
11    // NOTE: il faut sauver la taille car celle-ci
12    // va changer à l'appel de pop_back();
13    int s = v2.size();
14    for(i=0; i<s/2; i++)
15        v2.pop_back();
16 }

```

28

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 élément (à l'inverse de `vector`), mais elle est optimisée pour l'insertion et l'effacement d'éléments à n'importe quelle position.

29

- ➡ `iterator erase(iterator pos)`: efface l'élément à la position *pos*.
- ➡ `iterator erase(iterator first, iterator last)`: efface les éléments 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*.

31

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.

30

Exemple

```
1 #include <iostream>
2 #include <list>
3 using namespace std;
```

32

```

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

Exercice

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

35

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

```

34

Correction

```

1 int main(int argc, char **argv) {
2     int i;
3     char c[10] = {'a','b','c','d','e','f','g','h','i','j'};
4     list<char> c1, c2;
5     list<char>::iterator itc;
6
7     for(i=0; i<10; i++){
8         c1.push_back(c[i]);
9     }
10
11    c2 = c1;
12
13    itc = c2.begin();
14    for(i=0; i<c2.size()/2; i++){
15        itc++;
16    }
17    c2.erase(c2.begin(), itc);

```

36

Quelques algorithmes

37

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);
```

39

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*.

38

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());
```

40

```
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());
```

41

Exercice

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.

43

```
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());
```

42

Correction

```
1 int main(int argc, char **argv) {  
2   int i;  
3   string s[5] = {"abc", "bbaa", "bb", "jhfds", "aaa"};  
4   vector<string> v;  
5  
6   for(i=0; i<5; i++)  
7   {  
8     v.push_back(s[i]);  
9     cout << v[i] << " ";  
10  }  
11  cout << endl;  
12  
13  sort(v.begin(), v.end());  
14  
15  for(i=0; i<5; i++)  
16  cout << v[i] << " ";  
17  cout << endl;  
18 }
```

44

Exercice

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

45

Correction

```
1 int main(int argc, char **argv) {
2     list<string> l;
3     list<string>::iterator it;
4
5     l.push_back("il"); l.push_back("fait"); l.push_back("beau");
6
7     for(it=l.begin(); it!=l.end(); it++)
8         cout << *it << " ";
9     cout << endl;
10
11     it = find(l.begin(), l.end(), "beau");
12
13     if(it != l.end())
14         l.insert(it, "tres");
15
16     for(it=l.begin(); it!=l.end(); it++)
17         cout << *it << " ";
18     cout << endl;
19 }
```

46