Les conteneurs de la STL

M2103 - Programmation Objets en C++

IUT Bordeaux - 2015-2016

Table des matières

1 Qu'est-ce qu'un conteneur ?

Un **conteneur** est une structure de données qui sert à contenir des éléments.

La STL (Standard Template Library) est une bibliothèque (maintenant intégrée à la norme C++) qui en propose divers types.

Ce sont des conteneurs *génériques*, applicables à divers types d'objets. Exemple, le conteneur `list` apparaît ici deux fois, le type des élements est indiqué entre chevrons :

```
list<int> ages;        // liste de int
list<string> noms;     // liste de string
```

Histoire La première version de la bibliothèque STL a été développée par Alexander Stepanov (alors chez Hewlett Packard) et Meng Lee à partir de 1994, en faisant un usage intensif de la programmation générique (templates) permise par C++.

Depuis C++11 elle partie de la bibliothèque standard.

2 Typologie

On peut établir une classification des principaux conteneurs

- les tableaux (de taille connue à la compilation `array` (C++11), ou extensibles `vector`)
- les listes chainées simples `forward_list` (C++11) ou doubles `list`
- les ensembles `set` et les multi-ensembles `multiset` (ordonnés ou pas)
- les dictionnaires `map` (ordonnés ou pas, avec répétitions ou pas)

Les conteneurs forment une bibliothèque cohérente, les opérations similaires portent le même nom. Par exemple `cont.size()` retourne le nombre d'éléments d'un conteneur, quelque soit son type. `cont.clear()` le remet à vide. `cont.empty()` indique si il est vide, etc.

3 Tableaux (array)

Les éléments d'un tableau sont désignés par un indice (de 0 à la taille moins un). L'accès par indice, noté avec des crochets [], se fait en temps constant, indépendant de la valeur de l'indice.

Les tableaux sont des **conteneurs de séquences** avec un accès direct aux éléments (par l'indice). La STL fournit deux types de tableaux

- `array`, tableaux de taille constante
- `vector`, tableaux extensibles (on peut ajouter/enlever des éléments à la fin)

Note : l'opération `clear()` n'est pas applicable à un `array`.

3.1 Déclaration d'un tableau

```
#include <array>
#include <vector>

array<string, 4> beatles;    //taille=constante
vector<string>   couleurs(4);
```

Notes :

- pour `array` la taille est une **constante** connue à la compilation, elle fait partie du type, qui est "tableau de 4 chaines".
- les éléments sont initialisés par le constructeur par défaut (pour `string` : chaine nulle), ici on a un vecteur, qui est initialisé à 4 chaines vides, mais sa taille pourra changer.

3.2 Parcours d'un conteneur, "range-based for loop"

C++11 a introduit une manière simple de passer en revue les éléments d'un conteneur, qui se traduit par une forme spéciale de la boucle `for`, qu'on appelle "for each" dans d'autres langages.

Dans sa forme la plus simple

```
array<string, 4> beatles
   { "john", "paul", "georges", "ringo"};

for (string nom : beatles) {
  cout << nom << endl;
}
```

À chaque tour de la boucle, la variable `nom` est créée par appel au constructeur par copie.

Mieux : Déclarer `nom` comme *référence constante* évitera cette copie :

```
for (const string & nom : beatles) { // mieux
  cout << nom << endl;
}
```

De même, si il s'agit de modifier les valeurs du tableau, il faudra utiliser une référence

```
for (string & nom : beatles) { // indispensable
   nom = "Sir " + nom;
}
```

C'est indispensable, parce que sinon, on modifierait `nom` qui est une *copie* de l'élément du tableau, sans modifier le tableau.

Enfin, notez que depuis C++11 le compilateur peut déduire le type de la variable de contrôle, à partir du type du conteneur. On indique simplement le type `auto` (équivalent à `var` de C#) :

```
for (auto & nom : beatles) {
   nom = "Sir " + nom;
}
```

ce qui rendra service dans d'autres circonstances, quand les types sont compliqués à exprimer.

Conseil : utilisez `auto` systématiquement. Sinon, quand on relit le code, cela suggère qu'une subtilité interdisait de le faire, et qu'il faut en chercher la raison. On peut chercher longtemps.

3.3 Tableau, parcours par indice

Quand on a vraiment besoin des indices, on peut en revenir au parcours avec une variable de contrôle partant de 0 etc.

```
array<int,10> t;
for (int i=0; i < t.size(); ++i)
{
   t[i] = i;
}
```

3.4 `array` vs. tableaux traditionnels

Le type générique `array` a été introduit en C++11 pour plusieurs raisons :

- homogénéité avec les autres conteneurs
- pour réaliser un tableau de taille constante, aussi efficace que les tableaux de base, et plus efficace que `vector` (qui gère en plus l'extensibilité)
- un `array` est une variable, contrairement aux tableaux. On peut donc affecter un tableau dans un autre, le passer par valeur, etc.

Rappel : on ne peut pas écrire

```
int a[10], b[10];
a = b;    // erreur à la compilation
```

et quand on passe un tableau en argument d'une fonction, la fonction peut modifier son contenu. Alors qu'avec un `array`, c'est parfaitement légal.

```
array<int,10> a, b;
a = b;   // ok
```

et un `array` est passé par copie si on ne demande pas un passage par référence.

L'introduction des arrays dans C++11 rattrape donc un problème de conception du langage C, dans lequel **un tableau n'est pas une collection** d'objets, mais l'**adresse de début** d'une zone contenant ces objets.

4 Notion d'itérateur

La bibliothèque STL utilise intensivement les **itérateurs**, objets qui servent à désigner une **position** dans un conteneur. On peut positionner un itérateur sur le premier élément d'un conteneur l'avancer, etc.

4.1 Exemple

Ce code montre une troisième façon de parcourir un tableau

```
array<string,10> noms;

for (auto it = noms.begin();
          it != noms.end();
          ++it)
{
  cout << *it << endl;
}
```

- `noms.begin()` est la position du premier élément
- `auto it = noms.begin()` déclare `it` comme itérateur positionné sur le premier élément
- cet itérateur peut avancer (`operator++`) sur l'élément suivant
- l'itérateur désigne un élément, qui est accessible par déréférencement (`operator*`).
- enfin, on peut le comparer avec `noms.end()` qui indique la fin.

Attention : : `noms.end()` désigne la position **après** le dernier élément.

4.2 Remarques

1. Le type explicite de `it` est "itérateur sur un tableau de 10 chaines", soit `array<string, 10>.iterator`. D'où l'intérêt d'`auto`.
2. Les itérateurs *ne sont pas* des pointeurs, mais les opérateurs ++ et * ont été surchargés pour qu'ils se comportent apparemment comme tels.
3. On peut en particulier utiliser l'arithmétique des pointeurs, par exemple `noms.end()-1` désigne le dernier élément.

Important : quand la bibliothèque STL on utilise une paire d'itérateurs pour désigner une sous-séquence dans un conteneur, cette sous-séquence va du premier itérateur jusqu'au second **non compris**.

C'est cohérent avec le fait que `end()` soit *après* le dernier élément d'un intervalle.

4.3 Parcours à l'envers

Il est possible de parcourir le conteneur **en sens inverse** avec des "*reverse iterators*", avec les méthodes `rbegin()` (désigne le dernier élément) et `rend()` (fin du parcours). L'incrémentation ++ fait avancer cet itérator (qui va en sens inverse).

```
// parcours en ordre inverse

for (auto it  = noms.rbegin();
          it != noms.rend();
        ++it)
{
  cout << *it << endl;
}
```

4.4 Itérateurs sur éléments constants

Si le conteneur contient des éléments constants, on emploiera `cbegin()` et `cend()` de type `const_iterator` ("pointeurs" vers des objets qu'on s'interdit de modifier)

Et `crbegin()` et `crend()` de type `constant_iterator`, pour un parcours à l'envers.

5 Usage des itérateurs

La bibliothèque `algorithm` contient un grand nombre de fonctions qui font appel aux itérateurs.

Par exemple `find` cherche un élément dans une (sous-)séquence, et retourne sa position, sous forme d'un itérateur. La sous-séquence est elle-même indiquée par une paire d'itérateurs.

```
#include <algorithm>    // std::find
#include <array>

int main () {
  array<int,4> a  { 10, 20, 30, 40 };

  // on cherche l'endroit où il y a 30
  auto it = find (a.begin(), a.end(), 30);

  // si on l'a trouvé, on y met 33
  if (it != myvector.end()) {
    std::cout << "Trouvé" << endl;
    *it = 33;
  } else {
    std::cout << "Absent" << endl;;
  }
}
```

5.1 Algorithmes et lambda-expressions

Certains algorithmes prennent des fonctions comme paramètres. Par exemple :

- `for_each` qui applique une action à chaque élément
- `find_if` qui trouve le premier élément qui satisfait une condition

Une première façon de faire est de définir des fonctions

```
void afficher_element(int n) {
....
}

bool est_pair(int n) {
...
}
```

pour les utiliser

```
for_each(t.begin(), t.end(),
         afficher_element);

auto it = find_if(t.begin(), t.end(),
         est_pair);

if (it != t.end()) {
   cout << *it << " est pair";
}
```

Il est également possible de passer en paramètre des lambda-expressions (vous en avez vu le principe en C#)

```
for_each(t.begin(), t.end(),
            [](int n) { cout << n << endl; }
         );
```

Exercice : Ecrire la recherche du premier nombre pair à l'aide d'une lambda-expression.

Solution

```
auto it =
   find_if(t.begin(), t.end(),
           [](int n) { return n%2 == 0; }
           );
}
```

5.2 Capture de variables

Les crochets indiquent la liste des variables extérieures que l'expression doit *capturer*, c'est à dire dont elle a besoin pour être évaluée.

Exemple : Trouver si la liste contient au moins un élément qui soit supérieur à une certaine valeur `seuil`

```
int seuil = 12;
...
auto it =
   find_if(t.begin(), t.end(),
             [seuil](int n) { return n > seuil; }
           );
}
```

Ici c'est la **valeur** de `seuil` qui est capturée.

On peut aussi capturer une référence

```
int seuil = 0;

auto condition = [&seuil](int n){return n>v;};
...
seuil = 100;
auto it = find_if(t.begin(), t.end(),
                  condition);
```

Avec le passage par référence, c'est une comparaison avec 100 qui est effectuée.

5.3 Quelques algorithmes utiles

- `copy(debut, fin, destination)` : copie une séquence (délimitée par début et fin) à un certain endroit.
- `copy_if(debut, fin, destination, condition)` copie les éléments qui satisfont une condition

— `sort(debut, fin)`
— `min_element`, `max_element`, ...

Se référer à la documention à chaque fois que vous avez l'impression d'écrire un algorithme "bateau" : il est probablement dans la bibliothèque.

6 La classe générique `vector`

C'est un conteneur de séquence. Comme pour les tableaux, les éléments sont physiquement rangés les uns à côté des autres, qui garantit un accès rapide à partir de leur indice.

La taille du `vector` peut changer. Les éléments sont stockés dans un tableau alloué dynamiquement, qui peut être réalloué pour l'agrandir quand on ajoute des éléments.

6.1 Déclaration

Un `vector` peut être déclaré et initialisé de différentes façons

```
vector<int> v1;           // vide
vector<int> v2(10);       // 10 éléments (= 0)
vector<int> v3(10,3);     // 10 éléments (= 3)
vector<int> v4{10, 20, 30};
vector<int> v5{v4};
vector<int> v6{v3.begin(),
               v3.end()}; // autre séquence
```

Dans le dernier cas, les éléments peuvent provenir d'un autre conteneur que `vector`.

6.2 Opérations typiques

Outre les opérations communes (size, [], empty ...) et celles sur les itérateurs (`begin`, `end`, ...), on peut typiquement

— ajouter un élément à la fin par `push_back(valeur)`
— retirer le dernier élément `pop_back()`
— redimensionner le vecteur en ajoutant/enlevant des éléments `resize(n)`

6.3 Exercice

Ecrire un programme qui remplit un vecteur `premiers` avec les nombres premiers plus petits que 100, selon l'algorithme suivant

```
premiers = vide
pour n de 2 à 100
faire
   si dans premiers il n'y a pas de
      nombre qui divise n
  alors mettre n dans premiers
fin.
```

Solution

```
vector<int> premiers {};

for (int n = 2; n < 100; n++) {
   if (find_if(premiers.begin(),
               premiers.end(),
               [n](int d){ return n%d == 0; }
              )
              == premiers.end())
      premiers.push_back(n);
}
```

Note : on écrit `[n]` parce que la lambda-expression doit conserver ("capturer") la valeur de `n` pour tester sa divisibilité par `d`.

6.4 Autres opérations

Il est également possible d'insérer (`insert`) un élément à une position donnée (ou plusieurs venant d'une autre séquence, indiqués par des itérateurs), ou d'en retirer (`erase`), mais il faut savoir que ces opérations vont entrainer un décalage des éléments suivants :

```
vector<int> v(100);

v.erase(v.begin()); // décale les 99 autres
```

et le coût de ce décalage sera proportionnel au nombre d'éléments décalés.

Remarque ça ne veut pas dire qu'il ne faut jamais faire d'insertions/suppressions dans un vecteur, mais si elles reviennent souvent sur de gros vecteurs, cela aura un impact sur les performances. Il faut alors se poser la question : un autre conteneur serait-il mieux adapté ?

Heureusement, nous avons le choix : si nous voulons avoir une séquence, nous pouvons aussi utiliser les listes (`list` et `forward_list`) et sinon, nous avons aussi les ensembles (`set` et cie).

7 Classe générique `list`

Les listes (`list`) sont des conteneurs de séquence pour lesquels les opérations d'insertion et de suppression se font en temps constant à tout endroit. Ils permettent le parcours dans les deux directions.

Elles sont implémentées sous forme de listes doublement chaînées. Les données sont stockées dans des "cellules" qui contiennent également des pointeurs vers l'élément précédent et le suivant de la chaîne.

Elles sont préférables aux tableaux pour les programmes qui utilisent intensivement les ajouts et retraits d'éléments à une place désignée par un itérateur.

Deux inconvénients :

— il n'y pas d'accès direct : pour accéder au dixième élément il faut partir du premier, passer au second etc.
— chaque cellule comporte, en plue de la valeur stockée, deux pointeurs.

7.1 les opérations typiques

— `push_front(valeur)`, `push_back(valeur)`, pour insérer une valeur au début ou à la fin
— `front()`, `back()` accesseurs au premier et dernier élément
— `pop_front()`, `pop_back()`
— `insert(iterateur,....)` pour insérer avant une position donnée
— `erase(...)` pour retirer un élément ou plusieurs.

On retrouve bien sûr la boucle `for`, et les itérateurs : l'opération ++ fait avancer un itérateur sur le maillon suivant.

7.2 Invalidation des itérateurs

Attention il faut savoir que la suppression d'un élément désigné par un itérateur rend cet itérateur *invalide*. Il ne faut donc pas faire

```
// pour enlever les éléments impairs

for (auto it  = liste.begin();
           it != liste.end();
           ++it)
{
   if (*it % 2 == 1)
      erase(it);          // NON !!!
}
```

Ceci peut se réparer en utilisant un second itérateur

```
for (auto tmp  = liste.begin();
          tmp != liste.end();
           )
{
   auto it = tmp ++ ;
   if (*it % 2 == 1)
      erase(it);     // oui
}
```

mais une **bien meilleure solution** est d'employer la méthode `remove_if` des conteneurs `list` :

```
liste.remove_if(
   [](int n){ return n%2 == 1; }
);
```

Règles générales : - ne pas utiliser un itérateur sur lequel on a fait un `erase()`. - de préférence, utiliser les algorithmes qui font les choses correctement.

7.3 Listes simples

Dans les cas où on n'a pas besoin de parcourir en marche-arrière, on préfèrera la classe générique `forward_list` introduit par C++11.

8 Classe générique `set`

Ce conteneur contient des éléments uniques, stockés dans un ordre spécifique (par défaut l'ordre croissant pour les nombres, les chaines, les pointeurs, etc.)

Le `set` est une structure optimisée pour ranger/rechercher/supprimer des éléments dont on connait la valeur (contrairement aux listes et tableaux, qu'il faut parcourir séquentiellement).

Dans la plupart des implémentations de la STL, les sets sont représentés par des *arbres binaires de recherche*.

8.1 Opérations typiques

Les opérations typiques sont

— `insert(valeur)`
— `find(valeur)` qui retourne un itérateur sur la valeur cherchée
— `erase(...)` qui supprime une valeur, ou une donnée désignée par un itérateur, ou une paire d'itérateurs.

Et on peut faire des parcours, demander la taille etc. L'ordre de parcours est celui des valeurs.

9 Un conteneur associatif : `map`

Les `map`s, appelés aussi *dictionnaires* ou *tableaux associatifs*, sont des **conteneurs associatifs** qui stockent des éléments formés d'une clé et d'une valeur.

La clé sert à identifier, de manière unique, les éléments.

9.1 Accès à un élément

Pour accéder à un élément, on peut mettre la clé entre crochets

Exemple :

```
map<string,float> examen;

examen["anna"] = 6.5;
examen["bob"]  = 12;
examen["anna"] = 16.5;   // modification

cout << examen["bob"] << endl;
```

Attention, il est permis d'accéder en consultation par [] à un élément absent, qui va être créé et initialisé par le constructeur par défaut des valeurs :

```
cout << element["charlie"] << endl;
// maintenant charlie est dans la map....
```

Une autre possibilité pour consulter est d'utiliser la fonction `at()` qui retourne une référence vers la valeur si la clé existe

```
element.at("anna") = element.at("bob") + 1;
```

et qui déclenchera une *exception* (voir autre chapitre) si on tente d'accéder à un élément inexistant.

```
cout << element.at("charlie") << endl; // crac
```

9.2 Test de présence

pour tester la présence d'une clé, on utilise `find`, qui retourne un itérateur

Cet itérateur est soit `end()`, soit un *pointeur* sur une *paire* contenant la clé et la valeur

```
string nom = ....
auto it = examen.find(nom);

if (it == examen.end()) {
   cout << nom << " est absent" << endl;
} else {
   cout << nom << " a "
        << it->second << endl;
}
```

9.3 Parcours

La boucle `for` parcourt l'ensemble des paires (l'ordre de parcours est celui des clés).

```
for (auto & paire : examen) {
  cout << paire.first
       << " => "
       << paire.second
       << endl;
}
```

et de même pour les itérateurs

```
for(auto it  = examen.begin();
         it != examen.end();
         ++it)
{
  cout << it->first
       << " => "
       << it->second
       << endl;
}
```

9.4 Suppression

Attention, pour une raison bizarre (un oubli du comité de normalisation) il n'y a pas de `remove_if` pour les maps, mais la technique déjà présentée est applicable. La voici rédigée sous forme plus compacte :

```
for(auto tmp = m.begin(), it = tmp;
         (it = tmp++) != m.end(); )
{
    ...
    if (condition) {
       ....
       m.erase(it);
       }
    }
}
```