partie 7 : Bibliothèque standard STL

MOOC Langage C++ INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 Standard Template Library : la STL

Il existe une bibliothèque du domaine public très classique en C++. Elle a été développée initialement par Hewlett Packard puis portée et optimisée sur tous les systèmes et architectures. Elle est donc très efficace pour manipuler des données. On y retrouve toutes les structures de données classiques et tous les outils permettant de les manipuler. La STL est construite à l'aide des patrons et s'appuie sur trois concepts de base :

- les conteneurs représentent les structures de données,
- les itérateurs permettent de se déplacer dans les conteneurs,
- les algorithmes fournissent des fonctions évoluées de manipulation des données.

2 Les conteneurs

Il existe deux familles de conteneurs :

- les conteneurs en séquence ou conteneurs séquentiels : ils correspondent à des structures de données ordonnées comme, par exemple, une liste ou un vecteur. On peut les parcourir selon un ordre particulier, on peut insérer ou supprimer un élément à un endroit que l'on choisit,
- les conteneurs associatifs : on associe dans ce cas une valeur à une clé par exemple comme un répertoire téléphonique. À partir de la clé, on accède à la valeur. La notion d'emplacement n'a alors pas d'importance.

Les conteneurs sont bâtis grâce aux patrons, ils peuvent donc contenir n'importe quoi : des types de base ou bien des objets. Le choix de telle ou telle famille de conteneurs sera guidé par l'utilisation que l'on veut ensuite en faire, et surtout en fonction des méthodes d'ajout et d'accès aux valeurs contenues dans le conteneur. On définit notamment pour les conteneurs la complexité des algorithmes qu'on leur applique. On utilise la notation dite "de Landau" : O(x) qui se définit ainsi : le temps d'une opération t est dite en O(x) s'il existe une constante k telle que, dans tous les cas, on ait $t \leq kx$. On retrouve par exemple :

- O(1) : ceci signifie que le temps d'accès est borné et surtout indépendant du nombre d'éléments,
- O(N) : où N représente le nombre d'éléments du conteneur, dans ce cas le temps d'accès maximum est proportionnel au nombre d'éléments dans le conteneur,
- etc.

2.1 Notion d'itérateur

Un itérateur est un objet qui généralise la notion de pointeur. Un itérateur possède une valeur qui permet d'accéder à un élément du conteneur ; on dira qu'il "pointe" sur un élément du conteneur. Il existe des méthodes permettant de manipuler l'itérateur :

- $++$ permet de pointer sur l'élément suivant, $--$ sur l'élément précédent,
- $*$ permet d'accéder à la valeur pointée,
- on peut comparer deux itérateurs manipulant un même conteneur,
- les conteneurs possèdent des méthodes pour initialiser les itérateurs : pour le premier élément begin(), pour pointer après le dernier élément end(), etc.

2.2 Les conteneurs séquentiels

Il existe trois conteneurs séquentiels principaux : `vector, list` et `deque`. `Vector` généralise la notion de tableau, `list` correspond à la notion de liste doublement chaînée et `deque` est une classe intermédiaire entre les deux précédentes, permettant des optimisations d'accès dans certains cas. Ces trois conteneurs sont de taille dynamique, ils peuvent donc augmenter ou diminuer de taille en cours d'exécution du programme.

2.2.1 Fonctionnalités communes aux conteneurs séquentiels

constructeur

Il existe différents types de constructeurs : vide, avec un nombre d'élément initiaux, à partir d'un autre conteneur, etc.

```
list<float> tf; // liste vide
vector<int> vect1(5); /* tableau de 5 éléments initialisés
 par défaut*/
vector<float> vect2(4,0.4); /* tableau de 4 éléments
 initialisés avec la valeur 0.4*/
int t[4]={2,5,6,1};
vector<int> vect3(t,t+6); /* tableau de 6 éléments
initialisé avec les valeurs contenues dans le tableau t*/
deque<int> d1(4);
deque<int> d2(d1);
```

opérateur et fonctions

Voici quelques opérateurs et fonctions disponibles :

- les conteneurs possèdent par défaut l'opérateur d'affectation : $v1 = v2$,
- la méthode `clear()` permet de vider le conteneur de son contenu : `v1.clear()`,
- la méthode `swap` permet d'échanger le contenu de deux conteneurs de même type : `v1.swap(v2)` ,
- l'ensemble des opérateurs de comparaisons classiques : $==$, $\leq$, etc,
- manipulation des itérateurs .

```
list<int>::iterator it;
list<int> l1(5);
it=l1.begin();
```

- l'insertion et la suppression d'éléments :
  - `v1.insert(position, valeur)` //insertion de la valeur à une position donnée par un itérateur,
  - `v1.erase(position)` //détruit la valeur à cette position,
  - `v1.erase(debut, fin)` // détruit toutes les valeurs dans l'intervalle donné par les positions [début, fin],
  - `pop_back()` détruit le dernier élément,
  - `push_back(valeur)` insère en fin.

2.2.2 vector

Il reprend la notion classique de tableau avec un accès direct aux valeurs en O(1) avec l'opérateur []. Toutefois, il permet en plus :

- une dimension dynamique,
- une utilisation d'opérateur par exemple de comparaison,
- des insertions et suppressions avec une efficacité en O(N).

Il existe une méthode pour accéder directement au dernier élément : `back()`
Pour pouvoir utiliser ce conteneur, il faut inclure `vector`.

Exemple:

Utilisation de l'itérateur de parcours inverse

```
#include<vector>
...
int main (){
  vector<int> v(10);
  vector<int>::reverse_iterator rit;
  for (rit=v.rbegin();rit!=v.rend();rit++)
      *rit=1;
  // autre manière
  rit=v.begin();
  rit+=3;
  *rit=2;
  v[4]=3;
  v.back()=4;
  return 0;
}
```

Il existe un ensemble de méthodes permettant de manipuler l'espace de stockage du vecteur. Par exemple, la méthode `reserve()` va permettre de réserver à l'avance de l'espace mémoire pour mettre de futurs éléments.

2.2.3 deque

`deque` offre des fonctionnalités proches d'un vecteur, il permet toujours un accès en O(1) et des insertions ou suppressions en général en O(N). En plus de l'insertion/suppression en fin, il offre la même chose mais en début, le tout en O(1) : `pop_front()` et `push_front()`. Il faut inclure `deque`.

```
#include <deque>
...
int main(){
  deque<int> v1;
  deque<int> v2(3);
  v1.push_back(1);v1.push_back(2);v1.push_back(3);
  v2[0]=1;v2[1]=2;v2[2]=3;
  if (v1==v2)
    cout <<"OK"<<endl;
  else
    cout << "appeler l'asile"<< endl;
  return 0;
}
```

2.2.4 list

Ce conteneur correspond au concept de liste doublement chaînée. Les insertions et suppressions vont maintenant pouvoir se faire en O(1), mais on ne disposera plus d'un itérateur à accès direct. On dispose sinon des mêmes opérations pour insérer/supprimer en début ou en fin comme `deque`. On dispose en outre d'autres méthodes :

- `remove(valeur)` : supprime tous les éléments de la liste avec la valeur passée en paramètre ,
- `sort()` : permet d'ordonner la liste selon un ordre lexicographique avec un algorithme en O(Log N).,
- `unique()` : permet de supprimer les doublons ,
- `l1.merge(l2)` : permet de fusionner la liste l1 avec la liste l2,
- etc.

2.2.5 Autres conteneurs séquentiels

Il existe d'autres conteneurs bâtis à partir des trois conteneurs précédents, afin de les spécialiser. "queue" permet de générer des files d'attente en mode FIFO. "stack" permet de créer une pile gérée en LIFO. "priory_queue" définit une file d'attente où, à chaque insertion d'un élément, la file est ré-ordonnée en fonction d'une priorité définie par l'utilisateur à travers un prédicat binaire (notion d'algorithme).

2.3 Les conteneurs associatifs

Il y a là aussi deux conteneurs principaux et deux autres qui en dérivent : `map`, `mutimap` et `set`, `multiset` .

Ils associent une valeur et une clé et permettent donc des manipulations par l'utilisation de la clé.

2.3.1 map

Il faut inclure `map`. `map` impose l'unicité des clés. Ceci permet de disposer de l'opérateur [] où l'on fournit la valeur de la clé (efficacité en O(Log N)). Les données sont ordonnées dans `map` en fonction de l'ordre sur les clés. Pour représenter le couple "clé,valeur" on utilise une classe nommée `pair`.

```
#include<map>
...
map<char, int> tableascii;
tableascii['A']=65;
tableascii['B']=66;
tableascii['C']=67;
cout <<" code ascci de B:"<<tableascii['B'] <<endl;
...
```

On dispose en outre de méthodes pour manipuler les map :

- les itérateurs permettent de pointer sur une paire,
- `first` et `second` permettent pour une paire particulière de récupérer respectivement la clé et la valeur,
- `find(clé)` fournie un itérateur sur la paire avec la clé fournit en paramètre,
- `make_pair(clé,valeur)` permet de créer un objet de type pair pour ensuite pouvoir l'insérer par exemple avec la méthode insert,
- `erase(clé)` permet de détruire la paire avec la clé passée en paramètre.

2.3.2 multimap

Il faudra inclure `map`. Contrairement à `map`, `multimap` peut avoir une même valeur de clé représentant plusieurs valeurs. Il y a généralisation de presque toutes les méthodes associées à `map`. On ne peut toutefois pas utiliser l'opérateur []. Des méthodes sont ajoutées comme par exemple `lower_bound(clé)` et `upper_bound(clé)` qui renvoient un itérateur sur la première paire et sur la dernière avec la clé fournie en paramètre.

2.3.3 set et multiset

Il faudra inclure `set`. `set` et `multiset` sont des cas particuliers, le premier de `map` et le second de `multimap` dans lesquels il n'y a pas de clé associée aux valeurs. L'autre grande différence est que les valeurs stockées sont considérées comme des constantes. On ne peut pas les modifier.

3 Algorithmes standards

Les algorithmes standards se présentent sous forme de patrons de fonctions. Leur code est écrit sans connaissance précise des éléments qu'ils devront manipuler. Ils utilisent pour cela les itérateurs. Il faut inclure `algorithm`. Voici quelques algorithmes et leurs méthodes associées :

- `copy(itérateur début de séquence, itérateur fin de séquence, itérateur sur endroit où mettre la copie)` : permet de copier un intervalle de valeurs et de la mettre à partir d'un endroit particulier,
- `generate(itérateur début, itérateur fin, fonction de génération de valeur)` : permet d'initialiser un ensemble de valeurs avec le résultat d'une fonction,
- `max_element(début, fin)` et `min_element(début, fin)` recherchent respectivement le min et le max d'une séquence de valeurs,
- `replace(début, fin, valeur cherchée, nouvelle valeur)` : correspond au "find and replace",
- `random_shuffle(debut, fin)` : réalise une permutation aléatoire des valeurs entre début et fin,
- etc.

Il existe beaucoup d'algorithmes prédéfinis : suppression, tri, recherche, fusion, numérique, ensembliste ; etc.