

TD 9

Les itérateurs de la STL

Objectif

Cet exercice apporte quelques compléments sur l'utilisation « naïve » de la STL (*Standard Template Library*). En particulier nous récapitulons les collections (*containers*) disponibles et introduisons deux concepts clés de la STL : les *itérateurs* et les *algorithmes*.

9.1 Introduction rapide à la STL

Nous avons déjà introduit des éléments de la STL au cours du TD 2 (classe `string`, `iostream` et `vector`) ainsi qu'au TD 7 (classe `map`). La STL est aussi brièvement décrite dans le chapitre 5 du cours (section 5.7). Enfin on pourra se reporter au site web de ce cours, sur l'Intranet de l'ESSI, ainsi qu'à l'un des sites fournissant une documentation en ligne de la STL (par exemple SILICON GRAPHICS, www.sgi.com, ou STLPORT, www.stlport.org).

Nous rappelons ici les éléments fondamentaux nécessaires à la compréhension des exercices qui suivent.

9.1.1 Collections (*containers*) de la STL

La STL définit un ensemble de classes génériques qui permettent de représenter des *collections homogènes de valeurs*. « Homogène » signifie que tous les éléments de la collection sont de même type ; « valeur » indique que les éléments en question sont copiés dans la collection¹.

La STL définit plusieurs types de collections qui peuvent différer par leur propriétés logiques (celles vues de l'utilisateur) : ordonnées ou non, mode de recherche et d'ajout d'un élément, etc. Elles diffèrent aussi par leur structure interne d'implémentation : listes chaînées, éléments consécutifs ou non en mémoire, arbre binaire, etc. La version initiale de la STL définit dix types de collections résumés dans le tableau ci-après.

Collections de base (l'ordre des éléments dans collection est celui de leur insertion)	
<code>vector<T></code>	« consécutif » en mémoire ; indexation et insertion au milieu et à la fin ; pas d'insertion en tête
<code>list<T></code>	liste doublement chaînée ; insertion partout
<code>deque<T></code>	compromis entre <code>vector</code> (indexation facile) et <code>list</code> (insertion facile)
Adaptateurs de collections	
<code>stack<T></code>	la pile (LIFO) habituelle avec les opérations <code>push()</code> et <code>pop()</code>
<code>queue<T></code>	la file (FIFO) habituelle avec les opérations <code>put()</code> et <code>get()</code>
<code>priority_queue<T></code>	une file dans laquelle les éléments sont rangés dans l'ordre (celui de <code>operator<</code> sur <code>T</code>).

1. Si l'on ne souhaite pas copier, on peut bien entendu utiliser des collections de *pointeurs* : c'est alors la valeur du pointeur qui sera rangée dans la collection.

Collections associatives (nécessitent <code>operator<</code> sur T)	
<code>map<K, T></code>	tableau associatif : K est le type de la clé, T celui de la valeur associée (tableau de T indexé par K) ; pas de duplication des valeurs de K ; collection ordonnée en K
<code>multimap<K, T></code>	collection semblable à <code>map</code> , mais duplication possible des valeurs de K
<code>set<K></code>	ensemble (pas de duplication, donc) ordonné avec l'opérateur <code><</code> de K
<code>multiset<K></code>	semblable à <code>set</code> , mais duplication possible

La révision de la norme C++ ajoute à ce tableau quatre nouvelles collections associatives utilisant le « hachage » au lieu d'arbres binaires. Nous ne considérerons pas ici ces nouvelles collections.

Dans le tableau précédent, T désigne en général le « type de valeur » (*value type*) de la collection, c'est-à-dire le type de ses éléments. **Attention**, cependant : dans le cas des `map` et `multimap`, ce type de valeur n'est pas T, mais `pair<K, T>` où `pair` est une structure générique prédéfinie :

```
template <typename K, typename T>
struct pair
{
    K first;
    T second;
};
```

Pour pouvoir utiliser un type de collection, il suffit en général d'inclure le fichier d'en-tête qui porte son nom (`<list>`, `<deque>`, `<map>`, etc.) et de rendre accessible l'espace de noms `std`.

9.1.2 Itérateurs

L'opération la plus fréquente appliquée à une collection est de la parcourir, totalement ou partiellement. Pour cela, la STL propose une interface unique, un **modèle abstrait** auquel adhèrent toutes les collections, *quelle que soit leur implémentation interne et donc le détail de leur parcours*.

Modèle abstrait des collections de la STL Ce modèle abstrait est le suivant : toutes les collections apparaissent (logiquement) et sont manipulables comme une liste doublement chaînée de leur type de valeur. Cette liste a la structure indiquée sur la figure 9.1. Elle est constituée de cellules contenant chacune la valeur d'un élément et elle est bornée par deux « sentinelles » au début et à la fin (les sentinelles, hachurées sur la figure, ne font pas partie de la liste à proprement parler et ne contiennent donc pas de valeur utile). La liste du modèle abstrait respecte évidemment l'ordre de la collection de base, que ce soit celui d'insertion ou celui intrinsèque des éléments pour les collections ordonnées (`priority_queue`, `map`, `set`, `multimap` et `multiset`).

Itérateurs Pour permettre le parcours d'une collection à travers son modèle abstrait, la STL fournit, pour chaque type de collection, un certain nombre de types d'itérateurs, par exemple :

```
list<double> l; //une liste de réels
list<double>::iterator it; //un itérateur sur liste de réels
```

Notez qu'un itérateur comme `it` n'est pas directement lié à une collection au moment de sa création. C'est en fait comme *une sorte de pointeur*¹ qui pointerait sur un élément de la collection : ainsi `it` ressemble-t-il à un `double *`.

1. Ce n'est pas, en général, un vrai pointeur ; on parle donc souvent dans ce cas de « pointeur habile » (*smart pointer*).

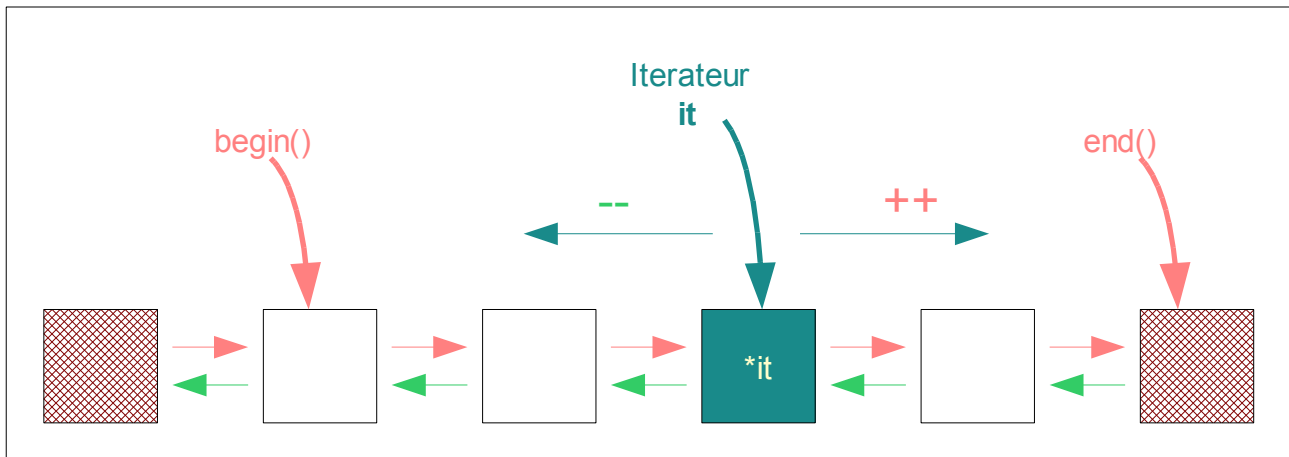


FIGURE 9.1 : Modèle abstrait des collections de la STL

Toutes les collections de la STL fournissent deux fonctions-membres, `begin()` et `end()`, qui retournent respectivement un itérateur sur le début (i.e., la première cellule utile) et la fin (i.e., la sentinelle finale) de la liste du modèle abstrait. L'opérateur `++`¹ appliqué à un itérateur permet de le « faire pointer » sur la cellule suivante, et `--` permet d'aller à l'élément précédent. Enfin, l'opérateur unaire `*` (`*it`) fournit une référence sur l'élément « pointé » par l'itérateur.

À titre d'exemple voici une boucle (tout à fait canonique) qui imprime tous les éléments de la liste de réels précédente en la parcourant grâce à l'itérateur `it` :

```
for (it = l.begin(); // it pointe sur le début de la liste
    it != l.end(); // tant que l'on n'est pas sur la sentinelle finale
    ++it) // aller à la cellule suivante
{
    cout << *it << endl; // imprimer l'élément courant
}
```

Quand on parcourt une map, il ne faut pas oublier que le type de valeur est une *paire* ; par exemple :

```
map<int, double> a_map;
...
for (map<int, double>::iterator it = a_map.begin();
    it != a_map.end(); ++it)
{
    cout << it->first << " " << it->second << endl;
}
```

L'opérateur `->` est bien sûr défini pour les itérateurs, à partir de l'identité habituelle

```
it->first ≡ (*it).first
```

Différents types d'itérateurs La STL dispose d'une riche classification des itérateurs. Ici nous nous contenterons de noter qu'il existe deux grands types d'itérateurs pour chaque collection :

- `const_iterator` : ces itérateurs ne permettent pas de modifier l'élément pointé par l'itérateur ; ainsi après
`list<double>::const_iterator cit;`
l'affectation
`*cit = 12`
est-elle incorrecte (ne compile pas) ;
- `iterator` : ces itérateurs permettent de modifier l'élément pointé.

1. Il en existe évidemment une forme préfixe et une forme postfixe.

Fonctions-membres principales Toutes les collections sont constructibles par défaut et copiables (copie des éléments à l'initialisation ou l'affectation). Elles disposent toutes d'un certain nombre de fonctions-membres communes dont les principales sont rappelées dans le tableau suivant (voir cours, section 5.7.3.2). Dans ce tableau *VT* désigne le type de valeur de la collection (noté *T* en général, mais rappelez-vous que c'est une paire dans le cas des maps) et *IT* désigne le type itérateur; enfin, *container* désigne bien entendu le nom du type de collection considéré.

Interface pour les itérateurs	
<i>IT</i> begin()	retourne un itérateur sur le début ou la fin de la collection
<i>IT</i> end()	(voir ci-après)
Accesseurs	
<i>VT</i> & front()	retourne une référence sur le premier ou le dernier élément
<i>VT</i> & last()	de la collection
<i>VT</i> & at(int i)	indexation avec ou sans vérification (voir 2.3.3).
<i>VT</i> & operator[] (int i)	
Insertions	
void push_front(const <i>VT</i> & e)	Insère un élément au début ou à la fin (<i>non disponible pour les collections ordonnées</i>)
void push_back(const <i>VT</i> & e)	
void pop_front()	retire le premier ou le dernier élément
void pop_back()	
insert(<i>IT</i> pos, const <i>VT</i> & e)	insère la valeur e devant l'élément « pointé » par pos
Opérations globales	
void clear()	détruit toutes les cellules (la collection devient vide)
void erase(<i>IT</i> pos)	détruit la cellule « pointée » par pos
void erase(<i>IT</i> beg, <i>IT</i> end)	détruit toutes les cellules entre beg (inclus) et end (exclus)
unsigned size()	nombre d'éléments dans la collection
bool empty()	la collection est-elle vide ?
bool operator==	égalité et plus généralement opérateurs relationnels
et != < <= > >=	(comparaison lexicale, <i>VT</i> doit posséder l'opérateur équivalent)
Constructeurs et destructeur	
<i>container</i> ()	constructeur par défaut (collection vide)
<i>container</i> (int n)	collection avec n éléments initialisés au zéro de leur type (<i>VT</i> ())
<i>container</i> (int n, const <i>VT</i> & x)	collection avec n éléments tous initialisés à x
<i>container</i> (<i>IT</i> first, <i>IT</i> last)	copie de la collection itérée par [first, last[(les deux collections doivent avoir le même type de valeur)
<i>container</i> (<i>other_container</i> cont)	copie de la collection (les deux collections doivent avoir le même type de valeur)
~ <i>container</i> ()	destructeur : détruit toutes les cellules
Opérations de copie	
constainer(const <i>container</i> &)	Constructeur de copie
<i>container</i> & operator=(Const <i>container</i> &)	Affectation de copie

9.1.3 Algorithmes

La STL fournit sous le nom d'*algorithmes* une soixantaine de fonctions génériques réalisant en un seul appel une opération sur tous les éléments d'une (ou plusieurs) collection(s). Pour utiliser les algorithmes, il vous faut inclure le fichier d'en-tête <algorithm>.

La collection sur laquelle opère un algorithme est fournie par deux itérateurs, disons *itf* et *itl*, qui définissent le début (inclus) et la fin (exclue) de la collection de travail (on désignera ceci par le terme « intervalle d'itération »). Par exemple, pour une liste *l*, *l.begin()* et *l.end()*. Bien

sûr, `itf` et `itl` doivent pointer sur des éléments de la même collection.

Voici un exemple. L'algorithme `find` prend trois paramètres : les deux premiers constituent l'intervalle d'itération, et le dernier est une valeur dont le type est le même que celui des valeurs de la collection ; il retourne un itérateur sur le premier élément qui a la valeur indiquée ; si un tel élément n'existe pas, il retourne la borne supérieure de l'intervalle d'itération. Quand à l'algorithme `copy`, il copie les valeurs de tous les éléments d'un intervalle d'itération vers une autre collection, dont le début est fourni par l'itérateur qui constitue son dernier paramètre.

```
list<int> l;
// ...
// recherche de la première occurrence de la valeur 5
list<int>::iterator itf = find(l.begin(), l.end(), 5);
if (itf == l.end())
    cerr << "première valeur non trouvée" << endl;
// recherche de la première occurrence de la valeur 10 à partir de itf
list<int>::iterator itl = find(itf, l.end(), 10);
if (itl == l.end())
    cerr << "deuxième valeur non trouvée" << endl;
else
{
    // copie de toutes les cellules entre les deux itérateurs précédents dans un vecteur
    // notez que si itf est l.end() rien n'est copié (l'intervalle est vide)
    vector<int> vec(l.size());
    copy(itf, itl, vec.begin());
}
```

Bien entendu, le vecteur et la liste doivent avoir le même type de valeur pour que `copy` fonctionne correctement. Notez aussi que la collection destinatrice doit être correctement dimensionnée : en effet, **aucun algorithme de la STL ne modifie le nombre d'éléments d'une collection**¹. Ici la dimension de `vec` est la même que celle de `l`, donc plus grande que nécessaire en général².

Certains algorithmes prennent des fonctions en paramètres. Ces fonctions seront appliquées sur chacun des éléments de la collection, et donc doivent avoir un unique paramètre dont le type est celui des valeur de la collection (pour éviter des ennuis avec certaines réalisations de la STL, passez toujours ce paramètre par *valeur*). Voici un exemple avec l'algorithme `for_each` qui applique une fonction (son troisième argument) aux éléments de son intervalle d'itération :

```
list<double> l;
// ...
// cette fonction sera appliquée sur chaque élément de la liste
void print(double)
{
    cout << x << endl;
}
// ...
for_each(l.begin(), l.end(), print);
```

Voici un exemple d'algorithme prenant une fonction à retour booléen, `count_if`. Il compte le nombre d'éléments dans l'intervalle d'itération qui satisfont la fonction booléenne :

1. Nous verrons, à la fin du cours, des moyens plus élégants de dimensionner dynamiquement la collection cible.

2. Quand aux petits malins qui, essayant de pousser un peu trop loin l'analogie entre pointeurs et itérateurs, penseraient que l'expression `itl - itf` est le nombre d'éléments à copier, qu'ils sachent que ce serait faux ici (cela ne compilerait même pas) ! Les itérateurs n'ont pas, en général, d'opérations arithmétiques, à part l'incréméntation et la décréméntation (`++` et `--`).

```
list<double> l;
//...
//une fonction à résultat booléen, applicable sur chaque élément de la liste
bool square_is_gt_10(double x)
{
    return x * x > 10.0;
}
int n = count_if(l.begin(), l.end(), square_is_gt_10);
```

Ici `n` sera donc le nombre d'éléments de la liste dont le carré est supérieur à 10. (Pour ceux d'entre vous qui se demanderaient comment faire `square_is_gt_y`, pour `y` quelconque donné en paramètre, il faudra attendre les dernières séances du cours...)

9.2 Manipulation des mots d'un texte

Il est temps de passer à l'exercice. Pour le faire vous vous reporterez avec profit aux solutions des TDs où nous avons déjà mis en oeuvre des collections de la STL : TD 2 et TD 7 (7.1.2) pour les maps.

On suppose que vous disposez d'un texte quelconque composé de mots séparés par des espaces. N'importe quel texte peut convenir, par exemple le contenu du fichier `elevation.txt` situé dans le répertoire correspondant à ce TD (`Word_Freq`) :

*au-dessus des étangs, au-dessus des vallées,
des montagnes, des bois, des nuages, des mers,
par delà le soleil, par delà les éthers,
par delà les confins des sphères étoilées,*

Fréquence des mots d'un texte Votre première mission est de lire ce texte depuis l'entrée standard mot par mot (un mot est défini par ce que lit l'opérateur `<<` de `istream` lorsqu'on lui demande de lire une `string` — on a déjà fait cela au TD 2), de compter le nombre d'occurrences de chaque mot, et d'imprimer le résultat. Pour compter le nombre d'occurrence, vous utiliserez une map de la STL associant à un mot un compteur entier. Pour imprimer le résultat, vous pouvez bien sûr utiliser une boucle d'itération, mais pourquoi ne pas essayer l'algorithme `for_each` ? Il vous faudra alors définir une fonction annexe.

Voici le résultat que vous devez obtenir avec le texte précédent (l'exécutable s'appelle `word_freq.exe` et `l_kheops%` est le message de sollicitation du **shell**) :

```
l-kheops% ./word_freq.exe < elevation.txt
Occurrences per word
-----
au-dessus : 2
bois, : 1
confins : 1
delà : 3
des : 7
le : 1
les : 2
mers, : 1
montagnes, : 1
nuages, : 1
par : 3
soleil, : 1
```

```
sphères : 1
vallées, : 1
étangs, : 1
éthers, : 1
étoilées, : 1
1-kheops%
```

Notez qu'avec la définition que nous avons pris pour un mot, la ponctuation fait partie du mot. Notez aussi que le tri s'effectue sur le code numérique et pas à la manière d'un dictionnaire français (voyez où sont placées les lettres accentuées !)¹. Nous nous contenterons de cela pour l'instant.

Ensuite, trouvez et imprimez tous les mots qui ont plusieurs occurrences. Pour cela vous écrirez une boucle ordinaire (**for**) utilisant l'algorithme `find_if`. Cet algorithme a trois paramètres : un intervalle d'itération et une fonction booléenne applicable à chaque élément de l'intervalle. Il cherche le premier élément qui satisfait la fonction booléenne et retourne un itérateur (non constant) dessus (ou la borne supérieure de l'intervalle d'itération si aucun élément n'est trouvé). Au passage vous sauvegarderez les cellules correspondantes (les paires, donc) dans une liste.

Enfin (pour cette question), calculez la longueur moyenne m des mots du textes, donnée par la formule

$$m = \frac{\sum_i n_i \times |w_i|}{\sum_i n_i}$$

où n_i est le nombre d'occurrences et $|w_i|$ la longueur du mot w_i . Ne finassez pas, utilisez une boucle d'itération directe (**for**).

Transformation d'une collection Nous souhaitons maintenant copier la liste des cellules correspondant aux mots qui ont plus d'une occurrence (ceux que nous avons obtenus plus haut) et les mettre dans un vecteur de mots (oubliant le nombre d'occurrences). Nous pourrions évidemment faire cela grâce à une boucle d'itération ordinaire (**for**), mais ne serait-il pas élégant d'utiliser un algorithme ? Le problème, ici, est que les deux collections n'ont pas le même type de valeur : la map contient des paires, le vecteur que nous voulons obtenir ne contient que des `string`. Un seul algorithme de la STL est capable de faire ce travail de transformation du type de valeur : il s'appelle justement `transform`. Il s'utilise de la manière suivante :

```
transform(itf, itl, itdest, ftranf);
```

où `itf` et `itl` constituent l'habituel intervalle d'itération, `itdest` est un itérateur pointant sur le début de la collection résultat, et `transf` est la fonction de transformation qui sera appliquée à chaque élément de la première collection et dont les résultats seront successivement rangés dans la seconde. Cette dernière doit prendre un paramètre (unique) du type de valeur de la première collection (dans notre cas une paire) et retourner une valeur du type de la collection résultat (dans notre cas une `string`).

Destruction des éléments dans une collection Enfin, dans le vecteur obtenu, on vous demande d'enlever tous les mots dont la longueur est inférieure ou égale à trois caractères. Pour cela vous utiliserez l'algorithme `remove_if`, dont nous vous laissons lire avec soin la documentation. En effet, comme aucun algorithme de la STL ne modifie le nombre d'éléments d'une collection, il doit y avoir un truc !

1. Il se peut que le tri soit différent dans votre environnement.