
TD 10

Algorithmes de la STL

Objectif

Les deux exercices suivants ont pour but de continuer l'initiation à l'utilisation de la bibliothèque STL de C++. Ils donnent l'occasion d'utiliser des objets-fonctions dans les algorithmes de la STL ainsi que des fonctionnelles associées. Ils montrent également comment on peut étendre la STL avec ses propres algorithmes.

Le premier exercice explore en outre certains des problèmes posés par les collections de pointeurs, indispensables avec des types polymorphes. Quant au second, il montre au passage comment les instances de `string` peuvent être aussi considérées comme des collections de la STL et se voir appliquer des algorithmes.

Pour ces deux exercices, quelques fichiers utiles sont fournis dans le répertoire `Algorithms`, situé à l'endroit habituel¹. Il sera sans doute utile de consulter la documentation de la STL, par exemple sur <http://www.sgi.com>.

La « règle du jeu » utilisée ici est d'éviter, autant que faire se peut et autant qu'il est élégant, les boucles d'itération explicites et de les remplacer par l'utilisation des algorithmes de la STL. Essayez de la respecter. Cela vous introduira au style très particulier mais, à mon avis, très plaisant de la programmation avec la STL.

10.1 Collections polymorphes

Dans le répertoire `Algorithms` mentionné précédemment, on trouvera le fichier `Figure.h` contenant la définition d'une hiérarchie de classes représentant des figures géométriques élémentaires (rectangle, carré, ellipse et cercle) avec quelques fonctions associées.

10.1.1 Utilisation des fonctionnelles

Écrire un programme principal qui utilise cette hiérarchie en créant une liste de pointeurs (pourquoi des pointeurs ?) sur quelques figures de différents sous-types (ces figures seront créées par `new`). Imprimer les figures de cette liste (noter que la classe `Figure` et ses dérivées disposent d'un opérateur `<<` pour les `ostream`). Appliquer une rotation de $\pi/2$ à toutes les figures de la liste et imprimer la liste en résultant; faire de même avec une homothétie (méthode `scale()`) d'un facteur 2.

10.1.2 Destruction des figures d'une collection

Ensuite on cherche non seulement à retirer de la liste toutes les figures qui sont des ellipses ou dérivées de `Ellipse`, mais aussi à les détruire définitivement.

Pour cela vous commencerez par définir une fonction (ou un objet-fonction) *template*, disons `is_kind_of`, tel que `is_kind_of<T>(p)`, où `p` est un pointeur sur `Figure`, retourne vrai si le type dynamique de l'objet pointé par `p` est en fait `T`. Pour cela vous aurez certainement besoin de l'opérateur `dynamic_cast`.

1. [/net/public/c/cm/C++EPU2](http://net/public/c/cm/C++EPU2)

Ensuite vous utiliserez cette fonction dans le cadre d'un simple `remove_if` et vous imprimerez la totalité (de `begin()` à `end()`) de la liste résultante. Que constatez-vous ? Avec un petit effort de mémoire, ceci devrait vous rappeler la fin de l'exercice 9.2.

Pour faire correctement cette dernière question, il faut tout d'abord détruire les objets pointés par les éléments de la liste. Pour cela vous définirez une fonction ou un objet-fonction `delete_Figure` qui, recevant un pointeur en paramètre, lui applique l'opérateur `delete` et met le pointeur à 0 (**attention à cette dernière spécification !**). Ensuite vous définirez un algorithme qui n'existe pas dans la STL, bien qu'il soit souvent pratique, `for_each_if`. L'appel

```
f = for_each_if(first, last, uf, pred);
```

applique la fonction `uf()` à tous les éléments de l'intervalle d'itération `[first, last[` qui satisfont le prédicat `pred()` ; comme pour `for_each`, la valeur retournée est une copie de `uf`, après traversée de toute la collection. Tout ceci devrait vous permettre de détruire tous les éléments de la liste qui sont des sortes d'ellipses et de mettre à zéro les pointeurs correspondants ; pour achever la question, il suffira de détruire effectivement (méthode `erase()`) les éléments ainsi annulés.

Note importante Dans tout cette exercice, **vous n'avez droit qu'à une seule boucle d'itération explicite**, celle qui permet d'implémenter l'algorithme `for_each_if`. Tout le reste doit être réalisé à l'aide d'algorithmes de la STL. Vous devrez également utiliser, chaque fois que faire se peut, les objets-fonctions et les fonctionnelles prédéfinis de la STL.

10.2 Index et références croisées

10.2.1 Affichage des éléments d'une collection de la STL

Votre première mission est d'écrire une fonction *template*, disons `display()`, dont le prototype est le suivant :

```
template <typename Container>
void display(ostream& os, const Container& cont,
             const string& sep = " ");
```

Cette fonction affiche sur la *stream* de sortie `os` tous les éléments de la collection `cont` (de type `Container`), séparés par la chaîne `sep`. `Container` doit pouvoir être le type de n'importe quelle collection itérable de la STL, y compris les *maps*.

Vous supposerez bien entendu que le `value_type` de `Container` est doté d'un opérateur d'affichage `<<`. Comme les paires (`struct pair`, le `value_type` de `map`) n'a pas d'opérateur `<<`, vous serez conduit à en définir un.

10.2.2 Constitution d'un index des mots d'un texte

Notre but est maintenant de construire l'index des mots d'un texte, c'est-à-dire que nous voulons connaître pour chaque mot, le numéro des lignes où ce mot apparaît. Notre index sera donc une *map* ou une classe encapsulant une *map* (une *map* de quoi donc, au fait ?).

Vous constituerez l'index de la manière suivante :

1. Vous lirez le texte ligne par ligne depuis l'entrée standard, grâce à la fonction `istream::getline()` (voir plus loin en 10.2.5) ; le fichier `camille.txt` fourni dans le répertoire `Algorithms` donne un exemple de texte utilisable ;
2. Dans la chaîne ainsi obtenue représentant une ligne, vous remplacerez tout caractère non alphabétique par un espace (vous pouvez ici utiliser l'algorithme `transform()` sur les `string` considérées comme des collections de `char`) ;
3. Vous transformerez la ligne ainsi obtenue en `istringstream` (voir plus loin en 10.2.5)

et vous extraieriez tous les mots de cette ligne, que vous placerez dans l'index, mais uniquement si leur longueur est strictement supérieure à 1 ;

4. Vous afficherez le contenu de l'index ainsi obtenu, c'est-à-dire, pour chaque mot, les numéros des lignes où ce mot apparaît.

10.2.3 Constitution de la table des références croisées

À partir de l'index obtenu précédemment, essayez de constituer le plus élégamment possible et sans recourir massivement aux boucles d'itération explicites, un table de références croisées : une telle table donne, pour chaque numéro de ligne, la liste (triée alphabétiquement) des mots que la ligne contient.

10.2.4 Recherche des mots apparaissant dans plusieurs lignes

Pour cette partie de l'exercice, vous ne devez utiliser aucune boucle d'itération explicite.

Revenez à l'index que vous avez constitué précédemment. Transformez-le un vecteur de paires, chaque paire comportant un mot et le nombre de lignes où ce mot apparaît. Triez ce vecteur par nombre de lignes croissant et affichez le résultat. Comme les paires n'ont pas d'opérateur <, vous en écrirez un. **Attention** : placez cet opérateur vous-même dans le *namespace std*, sinon vous aurez des ennuis...

Enfin copiez dans une liste les paires où le mot apparaît dans strictement plus d'une ligne et affichez le résultat. Il se peut qu'un algorithme comme `copy_if()` puisse ici vous faciliter la vie. Vous devinerez facilement, d'après le nom et l'analogie avec `copy()`, le rôle et le prototype de cet algorithme. Malheureusement, il n'existe pas dans la STL, alors... écrivez-le donc.

10.2.5 Quelques compléments

Fonction-membre `istream::getline()` Elle s'utilise de la manière suivante :

```
const unsigned MAXL = ...;
char buff[MAXL];
if (cin.getline(buff, MAXL))
{
    // ici on a pu lire une ligne de longueur maximale MAXL
    // et le résultat est disponible dans le tableau de caractères buff
}
else
{
    // ici la lecture de ligne a échoué pour quelque raison, par exemple une fin de fichier
}
```

Notez que le résultat de la lecture est dans un tableau ordinaire de C. Il convient de s'empresse de le convertir en `string` pour l'utiliser plus facilement. Ceci se fait sans problème.

Manipulation des `string` comme des collections Les `string` de la STL sont en fait utilisables comme des collections. Elles ont toutes les propriétés des collections ordonnées, en particulier, elles disposent de l'interface d'itération. Leur `value_type` est, bien entendu, `char`. On peut donc sans problème leur appliquer les algorithmes de la STL.

Manipulation des `istream` Les *stringstreams* permettent de simuler des entrées-sortie à partir (`istream`) ou vers (`ostream`) un buffer en mémoire. Pour les utiliser, vous devez inclure le fichier d'en-tête `<sstream>`.

Une `istringstream` peut se construire aisément à partir d'une `string`, et s'utilise ensuite comme une `istream` (dont elle dérive) :

```
string s = "12 3.14159";  
istringstream istr(s); // construction d'une istringstream  
int i;  
double x;  
istr >> i >> x; // i = 12 et x = 3.14159, maintenant
```