PROGRAMMATION PAR OBJETS

LANGAGE C++

COMPLEMENT SUR LA *STANDARD TEMPLATE LIBRARY*

Notes de Cours

N. Castagné, M. Desvignes, F. Portet

Version 2016.01

INTRODUCTION A LA STL - COMPLEMENTS

Ces notes de cours apportent des compléments à la partie du polycopié consacré à la Standard Template Library (STL) du C++. L'objectif n'est pas de faire une présentation complète de la STL, mais de positionner les connaissances essentielles à retenir dans le cadre du module.

➢ Notion de type abstrait et de structure de donnée

Lorsqu'on a besoin de stocker en mémoire *plusieurs éléments de même type* dans un programme, les tableaux C++ ne sont souvent pas adéquats.
Par exemple :

- La taille d'un tableau doit être connue à sa creation
- Ajouter un élément à un tableau plein est une opération couteuse : il faudrait réallouer le tableau pour augmenter sa taille !
- Un tableau ne garantit pas que les éléments qu'il contient sont uniques, ou qu'ils sont ordonnés suivant une relation d'ordre.
- Un tableau est indexé par un entier... mais ne permet pas d'accéder à une case à partir d'autre chose qu'un entier (accéder à une *définition* à partir d'un *mot* par exemple).
- Etc.

Il n'y a pas de solution unique pour tous ces besoins que l'on peut rencontrer. Mais, en fonction des besoins, il est possible de choisir un *type abstrait de donnée* adequate, qui permettra d'organiser les elements dans une *structure de donnée* adequate.

Structure de donnée : collection finie d'éléments organisés d'une certaine manière en mémoire, qui réalise (implante) un type abstrait, au moyen d'un langage de programmation.

Type abstrait : concept qui définit une vision abstraite d'une catégorie de structures de données similaires. Définit une *sémantique* indépendante de l'implantation (de la structure de donnée choisie pour l'implanter).

Un type abstrait est définit égalemnet des **opérations** que l'on peut faire sur la collection de données. Ces opérations sont implantées, dans les structures de donnée au moyen, de fonctions ou méthodes.

Exemples :

- Le type abstrait *Liste* : collection d'éléments ranges séquentiellement (le premier, le second, etc.). Le *tableau* est une structure de donnée qui réalise (implante) le type abstrait Liste. Exemple d'opérations: ajouter un element à la fin, accéder au $i^{ème}$ élément, etc.
- Le type abstrait Ensemble: collection d'éléments dans laquelle chaque element est unique (il n'apparait qu'une fois).

INTRODUCTION A LA STL - COMPLEMENTS

Pour en savoir plus sur la notion de « type abstrait » et la façon de les implanter, consulter le cours Phelma 1A-S2: http://tdinfo.phelma.grenoble-inp.fr/1Apet/cours.html

➢ Quelques grandes catégories de types abstraits et structures de données

Les listes (« *list* »)

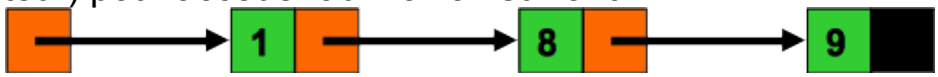
- **collection séquentielle d'éléments (le premier, le second...)**
  - la notion de $i^{ème}$ élément a un sens (on peut y accéder, etc.)

- **Principales opérations :**
  - Ajouter un élément en tête, à la fin (parfois aussi : à la $i^{ème}$ place)
  - Supprimer un élément : le premier, le dernier (ou parfois : le $i^{ème}$...)
  - Récupérer le nombre d'éléments
  - Accéder au premier ou au dernier élément (parfois : au $i^{ème}$ élément)
  - Parcourir tous les éléments (dans l'ordre séquentiel)

- **Quelques variantes possibles :**
  - Les tableaux :
    - taille fixée à l'avance !
    - les données sont contigues en mémoire
    - accès au $i^{ème}$ élément immédiat, en O(1)

  - Les tableaux redimensionnables :
    - on peut ajouter autant d'éléments que l'on veut...
    - Mais l'ajout (ou la suppression) d'un élément peut être couteux (complexité en O(n)), car il faut faire augmenter (ou diminuer) la taille du tableau redimensionnable en mémoire
    - les données sont contigues en mémoire
    - accès au $i^{ème}$ élément immédiat, en O(1)

  - Les listes chainées :
    - on peut ajouter autant d'éléments que l'on veut.
    - les données sont stockées dans des « maillons » : chaque maillon contient la donnée, et un moyen (par exemple : un pointeur) pour accéder au maillon suivant.



    - Accès au premier (ou au dernier élément) en O(1)
    - accès au $i^{ème}$ élément : pas possible directement
    - ajout en tête (ou en queue) en O(1)

INTRODUCTION A LA STL - COMPLEMENTS

➢ Quelques grandes catégories de types abstraits et structures de données (suite)

Les ensembles (« *set* ») :

- **collection d'éléments dont on assure l'unicité**
  - il n'y a pas deux fois « le même » élément dans la collection
  - un élément donné est présent dans l'ensemble, ou non
  - les éléments ne sont pas rangés de manière séquentielle. L'ordre d'ajout n'est pas conservé.
  - nécessite qu'il soit possible de comparer deux éléments entre eux

- **Principales opérations :**
  - Insérer un élément (si il n'est pas déjà dans l'ensemble)
  - Supprimer un élément (si il est dans l'ensemble)
  - Vérifier si un élément est dans l'ensemble ou non
  - Récupérer le nombre d'éléments
  - Parcourir tous les éléments (dans quel ordre ?)

- **Principales variantes possibles :**

  - Les ensembles basés sur une relation d'ordre
    - les données sont rangées en mémoire dans un arbre rouge-noir, ou dans un ABR simple
    - nécessite qu'une relation d'ordre sur les éléments soit définie
    - le parcours se fait dans l'ordre défini par la relation d'ordre (du « plus petit » au « plus grand »)
    - ajout / suppression d'un élément en O(log(n)) en moyenne
    - rechercher la présence d'un élément en O(log(n)) en moyenne

  - Les ensembles basés sur une table de hachage :
    - les données sont rangées dans une « table de hachage »
    - nécessite qu'une fonction de hachage existe sur les éléments
    - le parcours se fait dans un ordre d'apparence aléatoire
    - ajout / suppression d'un élément en O(1) en moyenne
    - rechercher la présence d'un élément en O(1) en moyenne
    - tend à occuper beaucoup de mémoire

Note : il peut exister des « ensembles » dans lequels un élément peut figurer plusieurs fois (multi-ensembles ou « multi-set »)

INTRODUCTION A LA STL - COMPLEMENTS

➢ Quelques grandes catégories de types abstraits et structures de données (suite)

Les tables associatives (« *map* ») :

- **Un table associative stocke en mémoire des associations clé -> valeur**
  - Le but d'une table associative est de permettre d'accéder rapidement à une valeur à partir de sa clé.
  - Les clés sont uniques
- **Exemple** : un dictionnaire de francais contient des mots (les « clés », chaines de caractères) et leurs définitions (les « valeurs », chaines de caractères)

- **Principales opérations :**
  - Ajouter une entrée (un couple clé-valeur)
  - Récupérer la valeur associée à une clé
  - Supprimer une entrée (par la clé)
  - Modifier la valeur associée à une clé
  - Vérifier si une clé est dans la table associative
  - Récupérer le nombre d'entrées
  - Parcourir tous les paires clé-valeur (dans quel ordre ?)

- **Principales variantes possibles :**
  - Les tables associativés basés sur une relation d'ordre sur les clé
    - les couples clé-valeur sont rangées en mémoire dans un arbre rouge-noir des clés, ou dans un ABR
    - nécessite qu'une relation d'ordre soit définie sur les clés
    - le parcours se fait dans l'ordre des clés, défini par la relation d'ordre
    - ajout / suppression d'un couple clé-valeur en O(log(n)) en moyenne
    - récupération de la valeur à partir d'une clé en O(log(n)) en moyenne
  - Les tables associativés basés sur une table de hachage des clés :
    - les données sont rangées par clé dans une table de hachage
    - nécessite qu'une fonction de hachage existe sur les clés
    - le parcours se fait dans un ordre d'apparence aléatoire
    - ajout / suppression d'un couple clé-valeur en O(1) en moyenne
    - récupération de la valeur à partir d'une clé en O(1) en moyenne
    - tend à occuper beaucoup de mémoire

Note : il peut exister des « tables associatives » dans lequels plusieurs valeurs peuvent être associées à la même clé.

INTRODUCTION A LA STL - COMPLEMENTS

➢ Implantations génériques d'un type abstrait

Une fois qu'on a identifié le type abstrait dont on a besoin pour manipuler des éléments, il est possible de l'implanter à la main dans une structure de donnée *ad hoc*.

Dans la vraie vie, on le fait parfois, par exemple lorsqu'on a besoin d'une structure de donnée très optimisée pour un problème spécifique.

Toutefois, les diverses implantations d'un type abstrait se ressemblent toutes plus ou moins, quel que soit le type des éléments...

Il existe donc des implantations génériques toutes prêtes, dans chaque langage.

La POO est particulièrement adaptée à des implantations génériques, notamment grâce au mécanisme d'encapsulation qui permet de ne voir que l'extérieur mais cache les détails d'implémentation (la « mécanique » de la structure de donnée).

En java : les « collections ».
En C++ : les « conteneurs ».

➢ Les conteneurs STL

Chacun des conteneurs C++ est une classe qui réalise une implantation générique d'un des types abstraits.

De la meme manière que, en C++, on indique le type des elements que contiendra un tableau lorsqu'on le declare, il faut indiquer le type des éléments que contiendra le conteneur lorsqu'on le declare : on créer une variable de type "liste de chaîne de caractères" ou "tableau associatif associant des entiers à des double", par exemple.

Cela est fait en C++ au moyen du mécanisme dit de généricité. Il ne vous est pas demandé de le connaître en detail – juste de savoir l'utiliser dans les conteneurs.

INTRODUCTION A LA STL - COMPLEMENTS

Le tableau suivant résume les types de conteneurs C++ les plus utiles et liste, pour chacun d'eux, les principales méthodes.

Pour utiliser un conteneur, remplacer le type générique **T entre crochet <>** par le type des elements que vous voulez mettre dans le conteneur. Par exemple :

```
vector<string> v; //un vecteur de chaînes de caractères
```

➢ Les listes STL

Type abstrait concerné	Type C++	Principales méthodes

Liste - Tableau redimensionnable	vector<T>	`int size() ;`
		
		`void push_back(const T &) ;`
		`void pop_back(const T &) ;`
		
		`T& front() ;`
		`T& back() ;`
		`T& at (int) ;`
		`T& operator[](int ) ;`
		
		`vector<T>::iterator erase(`
		
		`vector<T>::iterator);`
		
		`vector<T>::iterator begin() ;`
		`vector<T>::iterator end() ;`

Liste - Liste chainée	list<E>	`int size() ;`
		
		`void push_back(const T &) ;`
		`void pop_back(const T &) ;`
		`void pop_front(const T &) ;`
		`void push_front(const T &) ;`
		
		`T& front() ;`
		`T& back() ;`
		
		`vector<T>::iterator erase(`
		`vector<T>::iterator);`
		
		`vector<T>::iterator begin() ;`
		`vector<T>::iterator end() ;`

INTRODUCTION A LA STL - COMPLEMENTS

➢ Les ensembles STL

Type abstrait concerné	Type C++	Principales méthodes

Ensemble basé sur une relation d'ordre	set<E>	`int size() ;`
		
		`pair<iterator, bool> insert(const T&) ;`
		`int erase(const T&) ;`
		
		`int count(const T&) ;`
		
		`iterator begin() ;`
		`iterator end() ;`

Ensemble basé sur une table de hachage	unordered_set<E>	

➢ Les tables associatives STL

Type abstrait concerné	Type C++	Principales méthodes

Table associative basée sur une relation d'ordre	map<K, V>	`int size() ;`
		
		`pair<iterator, bool> insert(`
		`const pair<K,V> &) ;`
		
		`V& operator[](const K & ) ;`
		
		`int erase(const K &) ;`
		
		`map<K, V>::iterator begin() ;`
		`map<K, V>::iterator end() ;`

Table associative basée sur une table de hachage	unordered_map<K, V>	`int size() ;`
		
		`pair<iterator, bool> insert(`
		`const pair<K,V> &) ;`
		
		`V& operator[](const K & ) ;`
		
		`int erase(const K &) ;`
		
		`unordered_map<K, V>::iterator begin() ;`
		`unordered_map<K, V>::iterator end() ;`

➢ Autres conteneurs STL

La STL définit aussi d'autres types de conteneurs : piles, files, etc.
Il n'est pas demandé de connaître ces conteneurs.

INTRODUCTION A LA STL - COMPLEMENTS

➢ Notion d'iterateur

En bon français (... ?), « Itérer sur une paquet de choses », c'est « parcourir un à un les choses contenus dans le paquet ».

Ainsi, « itérer sur un tableau », c'est « parcourir les éléments d'un tableau ». Pour itérer sur un tableau, on peut utiliser une variable entière (dont la valeur est l'index dans le tableau de l'élément qu'on est en train de consulter), ou un pointeur (sur un élément). A chaque tour de boucle, on incrémente cette variable entière (ou ce pointeur) pour passer à l'élément suivant.

Dans ce cas, on peut dire que la variable entière (ou le pointeur) utilisé(e) a fonction « d'itérateur ».

Dans le cas des conteneurs STL, une simple variable entière n'est pas suffisante pour itérer sur un conteneur. Ca pourrait suffir pour un vector<T> (puisqu'un vecteur permet d'accéder au ième élément en O(1) avec l'opérateur [int] ou la méthode at(int)... Mais ce n'est pas adapté pour parcourir, par exemple, une list<T> (puisqu'une list<T> ne permet pas d'accéder facilement au $i^{ème}$ élément), ou un ensemble.

En fait, le parcours de chaque type de conteneur nécessite un mécanisme spécifique, adapté à la structure de donnée utilisée dans le conteneur.

La notion d'itérateur en C++ (et plus généralement en programmation...) permet d'unifier la façon dont on parcoure un conteneur dans son code (même si, dans notre dos, des choses diverses peuvent se passer).

En C++, on peut se représenter un itérateur comme un petit objet qui :

- fait référence à un des éléments du conteneur
- dispose d'un opérateur ***** qui permet de récupérer l'élément courant (comme un déréférencement de pointeur)
- dispose d'un opérateur **++** qui permet de passer à l'élément suivant (comme une incrémentation d'entier)
- *ainsi que de quelques autres méthodes*

STL - EXEMPLE : VECTOR<COMPLEXE>

Un conteneur peut contenir des objets. Voici par exemple un vector de nombres complexes...

```
#include <iostream>
#include <vector>
using namespace std;

class Complexe {
public:
   Complexe(double, double){
     //...
   }
   //...
} ;

main() {
// un vecteur de Complexe, vide pour le moment.
   vector<Complexe> v;

   v.push_back(Complexe(3,3)) ;
   v.push_back(Complexe(2,3)) ;
   v.push_back(Complexe(2,9)) ;
   //cout << v[1] ;
   v[2] = Complexe(4, 5) ;
   v.at(0) = Complexe(2,2) ;

   // on supppose que l'opérateur + est défini
   // pour les vecteurs
   v[0] = v[0] + v[1];

    v.pop_back();

   vector<Complexe>::iterator itr ;
   for(itr = v.begin() ; itr != v.end() ; ++itr) {
     Complexe c = *itr ; // déréférencement de
l'iterateur
       //cout << c << " " ;
       // ou directement :  cout << *itr << " " ;
   }
}
```

STL - EXEMPLE : LIST<VEHICULE *>

Un suppose dans cet exemple qu'il existe une hiérarchie de classes de véhicules : classe mère abstraite Vehicule ; sous-classe concrête Camion et Voiture par exemple... On veut regrouper plusieurs véhicules (des voitures, des camions) dans une liste chainée.

On ne peut pas créer une liste de Vehicule, car Vehicule est une classe abstraite :

```
list<Vehicule > l; // pas possible : on ne peut pas
    // manipuler des objets de type dynamique Vehicule !
```

On a donc recours, comme lorsqu'on utilisait des tableaux, à une liste de pointeurs sur des Vehicule :

```
list<Vehicule *> l; // liste de pointeurs sur Vehicule.
                    // Les Vehicule pointes peuvent être
                    // des camion ou des voitures (type dynamique)
```

```
#include <iostream>
#include <list>
using namespace std;
class Vehicule {
  //...
} ;
class Voiture : public Vehicule {
  //...
} ;
class Camion : public Vehicule {
  //...
} ;

main() {
  list<Vehicule *> l; // une liste chainée de Vehicules.
  l.push_back(new Voiture()) ;
  l.push_back(new Camion()) ;
  l.push_front(new Voiture()) ;

   // on suppose que l'opérateur << est défini sur les Vehicule
   cout << "premier vehicule " << * l.front();
   cout << "dernier vehicule " << * l.back();
   cout << "tous les vehicules : " ;
  list<Vehicule *>::iterator itr ;
  for(itr = l.begin() ; itr != l.end() ; ++itr) {
    Vehicule * ptrVehicule = *itr ;
     cout << *ptrVehicule << " " ;
    // ou directement :  cout << **itr << " " ;
  }
}
```

STL - EXEMPLE : SET<STRING>

Cet exemple affiche tous les mots contenus dans une liste de chaines de caractères *une seule fois* (en supprimant tous les doublons) et dans l'ordre lexicographique. Il a recours, pour ce faire, un en ensemble `set<string>`.

```
#include <iostream>
#include <list>
#include <set>
#include <string>
using namespace std;

int main() {
   list<string> l;
   l.push_back(string("contrepetrie : "));
   l.push_back(string("il"));
   l.push_back(string("fait"));
   l.push_back(string("beau"));
   l.push_back(string("et"));
   l.push_back(string("chaud"));
   l.push_back(string("chaud"));
   l.push_back(string("chaud"));

   set<string> s;
   list<string>::const_iterator it ;
   for(it = l.begin() ; it!=l.end() ; ++it) {
      s.insert(*it); // insertion du mot dans le set
   }

   cout <<"la liste contenait les mots suivants : "<< endl;

   set<string>::const_iterator itSurSet ;
   for(itSurSet = s.begin(); itSurSet!=s.end(); ++itSurSet){
      cout << "'" << *itSurSet << "' ";
   }
   cout << endl;

   return 0 ;
}
```

Ce programme affiche dans le Terminal :

```
la liste contenait les mots suivants :
'beau' 'chaud' 'contrepetrie : ' 'et' 'fait' 'il'
```

STL - EXEMPLE : UNORDERED_MAP<STRING , INT>

Cet exemple compte le nombre d'occurrence de chaque mot d'une liste de mots, et affiche ce nombre d'occurrence (sans respecter l'ordre lexicographique des mots : une unoredered_map n'est pas ordonnée).

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <string>
using namespace std;
int main() {
   list<string> l;
   l.push_back(string("si"));
   l.push_back(string("six"));
   l.push_back(string("scies"));
   l.push_back(string("scient"));
   l.push_back(string("six"));
   l.push_back(string("cyprès"));
   l.push_back(string("si"));
   l.push_back(string("près"));

   // une table associative qui associe chaque clé
   // de type "chaine de caractères" à un entier
   unordered_map<string, int> m;

   list<string>::const_iterator it ;
   for(it = l.begin() ; it!=l.end() ; ++it) {
      string motConcerne = *it;
      if(m.count(motConcerne) != 0) {
         m[motConcerne] ++; // mot pas encore dans la map
      } else {
         m[motConcerne] = 1;
         m.insert(pair<string, int>(motConcerne, 1));
      }
   }
   cout << "Occurence des mots dans la liste : " << endl;
   unordered_map<string, int>::const_iterator itMap ;
   for(itMap = m.begin() ; itMap!=m.end() ; ++itMap) {
      cout << "'" << itMap->first
      << "' est contenu "
      << itMap->second
      << " fois " <<endl;
   }
   cout << endl;
   return 0 ;
}
```

Ce programme affiche dans le Terminal :

```
Occurence des mots dans la liste :
'près' est contenu 1 fois
'scient' est contenu 1 fois
'scies' est contenu 1 fois
'cyprès' est contenu 1 fois
'six' est contenu 2 fois
'si' est contenu 2 fois
```

La complexité globale de ce programme est en 0(nombre de mots contenus dans la liste), car l'accès aux éléments dans une `unordered_map` est en O(1).

Notez qu'en remplaçant `unordered_map` par `map`, les mots s'afficheraient dans l'ordre lexicographique (ordre sur les clés d'une map).
Par contre, la complexité passerait en O(n * log(n)) car l'accès aux éléments d'une `map` est en O(log(n)).

REMARQUE SUR LES SET, LES MAP ET LES UNORDERED_MAP

Comme indiqué prédément :

Un `set<T>` nécessite qu'il existe une relation d'ordre sur le type T.
Une `map<K, V>` nécessite une relation d'ordre sur le type des clés K.
Une `unordered_map<K, V>` nécessite une fonction de hachage sur le type des clés K.

Il ne vous est **pas** demandé de savoir écrire en C++ une relation d'ordre ou une fonction de hachage si elle n'existe pas encore (par exemple : pour des classes que vous définissez vous même).

On conséquence, pour les ensembles `set<T>` et pour les clés des tableaux associatifs, on se limitera aux types basiques du C++, pour lesquels le C++ fourni une relation d'ordre et une fonction de hachage.

Ainsi par exemple :
`set<string>, set<int>, set<double>` sont OK dans le cadre de ce cours.
`map<int, Vehicule *>, map<string, string>` sont OK dans le cadre de ce cours.
*Mais il ne vous est **pas** demandé, par exemple, de savoir implanter une*

```
unordered_map< MesComplexe, int>
```

car le type `MesComplexe` n'est pas un type basique du C++.