ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

TP 6
Standard Template Library (Les conteneurs)

Objectifs : Utilisation de quelques conteneurs de la STL et des itérateurs associés. Implémentation d'algorithmes basés sur les itérateurs.
Durée : 2 séances

Ce TP comporte 5 exercices. Vous traiterez chaque exercice dans un fichier séparé (code demandé et fonction `main`). Huit questions vous sont posées. Les réponses seront fournies dans un fichier README.

1 Les itérateurs version STL

Le type itérateur de la STL est une abstraction de la notion de pointeur. Sa fonction principale est de décrire une position quelconque à l'intérieur d'un conteneur, quel que soit son type (vecteur, ensemble, liste, etc.). Il a aussi comme intérêt de permettre de parcourir les éléments d'un conteneur de manière générique, indépendante du conteneur associé. Il devient donc possible de programmer des algorithmes génériques, c.-à-d. des modèles de fonctions paramétrés par des itérateurs.

Dans sa version de base, un itérateur offre essentiellement deux services :
- L'incrémentation, via l'opérateur ++ qui le fait *pointer* sur le successeur de sa position courante.
- Le déréférencement, via l'opérateur * qui retourne une référence sur l'élément pointé.

Toutes les collections de la STL possèdent les deux méthodes `begin()` et `end()` qui retournent respectivement un itérateur pointant sur le premier élément et sur un élément fictif qui se trouverait juste après le dernier élément. Autre exemple : les conteneurs `map`, `multimap`, `set` et `multiset` possèdent une méthode `find()` qui retourne un itérateur pointant sur l'élément recherché ou bien `end()` s'il n'est pas trouvé.

a) Écrivez une fonction qui remplit un vecteur avec 40 entiers tirés aléatoirement entre 0 et 20 et parcourt ensuite le vecteur pour en afficher le contenu. Les entiers sont ajoutés au vecteur à l'aide de la méthode `vector<int>::push_back(const int &)`. On parcourt les éléments du vecteur à l'aide d'un objet de la classe `vector<int>::iterator`.

b) Écrivez une fonction similaire à la précédente mais qui utilise un *ensemble* d'entiers (type `set<int>`) dans lequel 1000 entiers, toujours aléatoires et compris entre 0 et 20, sont insérés.

Q 1. Que remarquez vous concernant l'affichage et la taille finale de l'ensemble ?

Q 2. Pour quelle raison une relation d'ordre est elle utilisée par les `(multi)set` et `(multi)map` ?

c) Écrivez un modèle de fonction qui affiche le contenu d'une séquence donnée par deux itérateurs (début et fin). Le type d'itérateur est donc le paramètre de type de ce modèle. Modifiez (au choix) une des deux questions précédentes en conséquence.

d) Vérifiez, par un message d'erreur du compilateur, que le modèle `stack<T>` n'est pas une collection mais bien un adaptateur qui ne dispose pas d'itérateur et donc pas non plus de méthode `begin()`. (Voir le diagramme d'héritage en page du support de cours.)

2 Taille et capacité d'un vecteur

La méthode `vector<T>::capacity()` retourne la taille du tableau utilisé par le conteneur pour stocker ses éléments. Il ne faut pas confondre cette taille avec le nombre d'éléments du vecteur (méthode `size()`). Pour tester la façon dont ce conteneur est implémenté on se propose de programmer un test montrant l'évolution de la capacité en fonction du nombre d'éléments contenus.

Programmez une boucle insérant 5000 éléments à la fin d'un vecteur de doubles (par la méthode `vector<T>::push_back(const T &)`) tout en détectant les changements de capacité du vecteur. La capacité s'affichera à chaque changement.

Q 3. Qu'en déduisez vous de la façon dont un vecteur est implémenté ?

3 Tri à l'aide d'une priority_queue

Rappel 1 : éviter les récopies inutiles (par exemple d'un conteneur) est une « obligation ».

Rappel 2 : la version `const` de la méthode `begin()` retourne un `const_iterator`.

a) L'adaptateur `priority_queue<T>` [1] implémente une file de priorité qui garantit l'insertion d'un élément et l'extraction de l'élément de plus forte priorité en $O(\log(n))$ où n est le nombre d'éléments dans la file (classiquement grâce à une structure de tas). Programmez à l'aide de ce conteneur une fonction de tri qui prend en argument un vecteur d'entiers et retourne une copie triée du vecteur. La démarche est simple : insérer tous les éléments du vecteur dans une file puis les retirer un à un. Vous aurez essentiellement besoin des méthodes suivantes :
   — `void vector<T>::push_back(const T &)` qui ajoute un élément en fin de vecteur ;
   — `void priority_queue<T>::push(const T &)` qui insère un élément dans une file ;
   — `T priority_queue<T>::top()` qui retourne la valeur de l'élément de plus grande priorité [2] ;
   — `void priority_queue<T>::pop()` qui retire de la file son élément de plus forte priorité.
   La relation d'ordre utilisée par défaut par le modèle `priority_queue` (cf. listing 1) est telle que l'élément le plus prioritaire est le plus grand. Faites en sorte que le tri soit bien en ordre croissant (cf. `std::greater<T>`).

b) Faites de la fonction précédente un modèle de fonction paramétré par le type des éléments du vecteur.

```
template<typename T,
         typename Sequence = vector<T>,
         typename Compare  = less<typename Sequence::value_type> >
class priority_queue {
     [...]
};
```

Listing 1 – Modèle `priority_queue`.

1. Défini dans l'entête **<queue>**.
2. Au sens de la « relation d'ordre » utilisée.

4 Tableaux associatifs

4.1 Parcours

Le modèle `map<K,V>` implémente les tableaux associatifs qui mettent en relation des clés de type `K` avec des valeurs de type `V`. Une `map` est en fait un ensemble de paires (modèle `pair<K,V>`, cf. page du support de cours) ordonné selon les valeurs des clés. De même que pour un ensemble, il faut qu'un comparateur, par défaut `less<K>` (c.-à-d. <), s'applique sur le type `K`.

L'insertion d'éléments dans un tableau associatif se fait comme dans l'exemple du listing 2.

```
#include <map>
// ...
{
  std::map<int,int> triple;
  triple[2] = 6;
  triple[5] = 15;
}
```

Listing 2 – Exemple d'utilisation d'une `map`.

Construisez un tableau « age » associant à une chaine (classe `string`) un entier. Remplissez ce tableau avec quelques éléments et affichez son contenu (sous la forme clé/valeur) à l'aide d'un itérateur.

Q 4. Qu'en déduisez vous sur le fonctionnement de `less<string>` ?

4.2 (Ordre sur) le modèle pair

On propose de vérifier que l'ordre lexicographique est bien défini sur les `pair<U,V>` dès lors qu'une relation d'ordre existe sur les U et sur les V.

Testez la vérité des relations suivantes en utilisant le modèle de fonction `make_pair<U,V>(U,V)` :

- `(1,10) < (2,3)`
- `(1,10) < (1,20)`
- `(1,10) < (1,5)`
- `(1,10) < (0,8.5)`
- `(0,0) != (0,8)`
- `(0,5) > (0,5)`
- `("pomme",10) < ("tomate", 40)`

Q 5. Quelle est l'utilité du modèle de fonction `make_pair()`, en tout cas avant la norme de 2017 (C++17) ?

4.3 Comportement de l'opérateur `[]`

```
#include <iostream>
#include <map>

int main() {
  std::map<int,double> tab;
  tab[10] = 5.0;
  tab[5] = 2.5;
  if (tab[0] == 0.0) {
    std::cout << tab.size() << std::endl;
  }
}
```

Listing 3 – Comportement de l'opérateur `[]` d'une map.

Q 6. En vous aidant éventuellement de la documentation disponible sur le site `cppreference.com`, expliquez pourquoi, dans le code du listing 3 :
- le test est toujours vrai ;
- la valeur affichée est "3".

5 Algorithmes

a) Écrivez un algorithme (c.-à-d. un modèle de fonction) qui, étant données deux séquences précisées par 4 itérateurs, vérifie que la première séquence est incluse dans la seconde, au sens ensembliste. (On rappelle si besoin que $\emptyset \subset S$, pour tout ensemble S.)

b) Programmez une variante du modèle de fonction de l'exercice 3, mais qui s'applique à une séquence d'un conteneur quelconque (type de conteneur comme type de contenu). Le conteneur trié ne sera pas retourné mais modifié sur place et la fonction sera applicable à toute séquence décrite par deux itérateurs.

c) Écrivez un modèle de fonction qui, étant donné deux itérateurs désignant chacun un élément d'une même séquence, échange les valeurs de ces deux éléments. Attention, ce modèle de fonction ne devra appeler aucune autre fonction. Vous aurez recours aux `traits` d'un itérateur pour obtenir le type des éléments du conteneur associé (cf. cours sur les traits et polycopié, page 123). Quelle serait l'alternative, à partir de `C++11` ?

6 Bonus : Comparaison `vector`, `deque` et `list`

On se propose ici de vérifier expérimentalement la complexité algorithmique de l'ajout d'un élément en fin de vecteur (ajout qui, rappelons le, est susceptible de provoquer un « realloc » quand la capacité est atteinte, comme observé en section 2).

a) Programmez un modèle de fonction paramétré par un type de conteneur `C` (supposé contenir des entiers et posséder une méthode `push_back()`) qui mesure le temps, au sens de la fonction `clock`, nécessaire pour ajouter n entiers aléatoires à la fin d'un conteneur de type `C` [3].

3. Attention : ici le conteneur n'est pas un paramètre de la fonction mais une variable locale.

b) À partir du modèle écrit en (a), mesurez par programme les temps nécessaires pour ajouter à un vecteur initialement vide (resp. une liste et une « `deque` ») de 10 000 à 1 000 000 d'entiers, par pas de 10 000. Ces temps seront écrits dans un fichier texte [4] dont chaque ligne suit le format ci-dessous :

```
nb_éléments  nb_clocks_vector  nb_clocks_deque  nb_clocks_list
```

À l'aide du programme Gnuplot (disponible sur cybele, cf. commandes fournies ci-après) tracez les courbes qui représentent les différents temps en fonction du nombre d'éléments ajoutés.

Q 7. Que constatez-vous concernant l'ajout de n éléments en fin de vecteur ? Que pouvez-vous en déduire sur la complexité algorithmique de l'ajout d'un élément en fin de vecteur ?

Q 8. Que peut-on dire de l'efficacité de l'ajout en fin de vecteur comparé à la même opération en fin de liste ?

Exemples de commandes Gnuplot pouvant être utiles (sans les deux premières, l'affichage du graphe se fera dans une fenêtre) :

```
set terminal png truecolor size 800,600
set output "graph.png"
plot "plot.txt" using 1:2 with line title "vector<>" linecolor "red"," \
     "plot.txt" using 1:3 with line title "deque<>" linecolor "green", \
     "plot.txt" using 1:4 with line title "list<>" linecolor "blue"
```

4. **`std::ofstream`**