TP 6

1 Utilisation de la STL

Cet exercice a pour but de vous faire réviser comment utiliser les `Containers` en lien avec les `Algorithms` de la `STL`. L'idée est de répondre en une unique ligne concise de code à chacune des questions. Ne pas rester plus d'une demi-heure sur cet exercice.

- Construisez un `deque d` de 5 entiers
- Remplissez `d` avec des multiples de 4 (on utilisera le générateur du TP 04)
- Affichez le contenu de `d` sur la sortie standard (on construira pour cela un `ostream_iterator` associé à la sortie standard et on aura recours à l'algorithme `copy` de la `STL` pour copier le contenu de `d` sur la sortie standard via les itérateurs adéquats).
- Construisez un ensemble `s1` d'entiers à partir de `d`
- Videz `d` de son contenu
- Construisez un ensemble `s2` d'entiers par insertion successive de 10 multiples de 2 (cette fois, vous avez le droit d'utiliser une boucle `for`)
- Construisez un ensemble vide `s3`
- Remplissez `s3` avec 6 multiples de 3 (on utilisera un `insertor` sur cet ensemble, ainsi que l'algorithme `generate_n`, voir site web de la `STL`)
- Affichez le contenu des ensembles `s1`, `s2` et `s3` en séparant les éléments par des `"; "`
- Recherchez sur le site web de la `STL` quelles sont les algorithmes permettant de réaliser des opérations ensemblistes
- Rangez le résultat de l'intersection de `s1` et de `s2` dans une liste `l` d'entiers
- Affichez le contenu de `l`
- Affichez directement le résultat de l'union des ensembles `s1` et `s3`

2 Itérateur sur le type abstrait générique : Tableau

Reprenez votre classe template Tableau et finissez-la.

Définissez une classe itérateur permettant d'accéder aux éléments d'un Tableau par déréférencement, et de parcourir tous les éléments de ce Tableau. Outre les possibilités d'incrémentation (versions pré et post) et de déréférencement, vous munirez cet itérateur d'un test d'égalité.

Dans un premier temps, vous pourrez commencer par définir la classe itérateur comme une classe extérieure à la classe template Tableau et tester le comportement de ces 2 classes dans un programme du type :

```
#include <iostream>
#include "tableau.hpp"

int main()
{
  Tableau<int,6> A(5);
  std::cin >> A;
  TabIterator<int,6> it=A.begin();
  TabIterator<int,6> ite=A.end();
  for(;it!=ite;++it)
```

```
    std::cout << *it << std::endl;
  return 0;
}
```

Que faire de plus pour définir un type itérateur interne à la classe template Tableau ? Cela permettra de parcourir les éléments d'un Tableau<int,6>A avec des instructions du type :

```
Tableau<int,6>::iterator it=A.begin();
Tableau<int,6>::iterator ite=A.end();
for(;it!=ite;++it)
  std::cout << *it << std::endl;
```

Faites une nouvelle classe d'itérateur qui permet de ne faire que le parcours des cases d'indice pair d'un Tableau. Dans le programme précédent, on pourra ainsi utiliser des itérateurs de type `Tableau<int,6>::oddIterator`.

3 Type abstrait générique : Liste Triée chaînée

Si vous avez terminé votre `Tableau` *templaté* par le type `T` de ses éléments et par la taille de ses agrandissements, nous vous proposons à présent de faire la mise en oeuvre des `ListesTriee`s chaînées. Vous aurez deux classes : une classe `Cellule` qui contiendra une information de type `T` ainsi que l'adresse de la `Cellule` suivante (`nullptr` sinon), et une classe `ListeTriee` qui contiendra l'adresse de la première `Cellule` et l'adresse de la dernière `Cellule` (`nullptr` si la liste est vide). La classe `ListesTriee` sera *templatée* par le type `T` de ses éléments qui devront disposer de l'opérateur de comparaison. Pour le moment, vous implémenterez uniquement les fonctionnalités d'initialisation par défaut, par copie (et éventuellement par déplacement), d'affectation (avec copie et éventuellement aussi par déplacement), d'ajout d'un élément, d'affichage, ainsi qu'un destructeur. On gardera pour plus tard les fonctionnalités de recherche d'un élément.