L3 MIASHS
TP C++ N° 3

Exercice 1 : les listes doublement chaînées et les modèles

Lors du TP précèdent, vous avez créées les classes `CNoeud` et `CListe`. Ces classes sont capables de stocker uniquement de pointeurs vers des objets du type `CPersonne`. L'objectif de cet exercice est d'implémenter une version **générique** de ces classes.

1. Modifiez la déclaration et la définition de `CNoeud` et de `CListe` afin de pouvoir stocker un pointeur d'un type de donnée quelconque `T`, à l'aide de *modèles de classe*. Par exemple, comme pour la `std:list`, le but est de pouvoir créer de listes pour stocker des types entiers (`CListe<int>`), des types personnes (`CList<CPersonne>`), etc.
2. Vous devez (au moins) :
   — créer une liste chaînée de `int`
   — créer une liste chaînée de `CPersonne`
   — imprimer le contenu des deux listes

S'il vous semble nécessaire, vous pouvez modifier la méthode `bool ajoute(T*)` de `CListe` afin d'ajouter un type `<T>` au lieu d'un pointeur `<T*>` de type T.

Exercice 2 : les conteneurs de la STL et les fonctions modèles

Pour cet exercice, vous allez utiliser les conteneurs fournis par la bibliothèque standard ainsi que les fonctions modèles pour manipuler ces conteneurs :

1. Pour les deux listes créées dans l'exercice précèdent, remplacez votre type `CListe` par `std:list`
2. Utilisez la fonction `sort` définie dans `std:list` pour trier les deux listes. Le tri d'objets du type `CPersonne` se fait par l'ordre croissant de noms
   — dans un premier temps, utilisez la fonction `sort` sans paramètre (prédicat)
   — dans un deuxième temps, la fonction `sort` reçoit en paramètre la comparaison à être prise en compte (> ou <). Pour cela, définissez deux fonctions : `croisant(const T &t1, const T &t2)` et `décroissant(const T &t1, const T &t2)`, qui seront utilisées en tant que prédicats pour `sort`. Finalement, remplacez les prédicats par des fonctions lambdas
3. Affichez le contenu des deux listes à l'aide des itérateurs. Pour optimiser le code, implémentez une fonction modèle `affiche` qui reçoit en paramètre une `std:list` d'un type quelconque `T` et affiche son contenu
4. Utilisez le conteneur associative approprié pour stocker un ensemble de `CPersonne` associé à une clé donnée (par exemple, l'ensemble des étudiants du groupe 1 de TP et l'ensemble des étudiants du groupe 2 de TP). Vous pouvez, à l'aide du type `pair`, stocker soit une `std:list` associée à une clé ('Gr1', par exemple) soit un objet `CPersonne` associé à une clé :

```
#include <utility>
...
pair<string,list> groupes;
// ou
pair<string,personne> groupes;
```

5. À l'aide de la fonction `affiche` que vous avez créée précédemment, affichez les personnes (stockées dans le conteneur associative !) qui sont associées à une clé donnée (n'utilisez aucune variable auxiliaire !!)

Exercice 3 : les algorithmes de la STL

Pour cet exercice, vous allez manipuler les algorithmes de la STL. **Pour chaque cas, vous devez fournir une version utilisant une fonction nommée et une version utilisant une fonction lambda. Vous devez utiliser uniquement les algorithmes fournis par la STL** :

1. Affichez la liste d'entiers créée dans l'exercice 2 à l'aide de `for_each`
2. Contez le nombre d'éléments impairs de la liste d'entiers
3. Remplacez les éléments impairs par une valeur donnée
4. Affichez la liste de personnes à l'aide de `for_each` (n'utilisez qu'une fonction pour afficher des types `int` et `CPersonne`, pour la version en utilisant les fonctions nommées)
5. Supprimez de la liste de `CPersonne`, les personnes ayant le nombre de caractères de leur nom un nombre impair (pour cela, utilisez `remove_if` de la `std:list` ou `remove_if` de la STL avec `erase` de la `std:list`)
6. Faites une recherche binaire pour vérifier si une personne donnée est sur la liste