TP Liste

Le but du TP est de réaliser l'implémentation du type abstrait Liste par une *liste chaînée circulaire avec sentinelle*. Les fichiers mis à votre disposition se trouvent dans le répertoire habituel.

Remarque : À chaque étape du TP, vous devrez valider votre gestion de mémoire avec la commande valgrind (voir 5).

1 Liste chaînée

Le chaînon utilisé dans ce TP vous est fourni ; il s'agit d'une classe générique, CyclicNode, paramétrée par le type des éléments qu'il contient. Il vous est recommandé d'étudier cette classe afin de bien comprendre son fonctionnement.

On vous demande d'implémenter une liste *générique* ; pour simplifier les notations, vous définirez (dans la partie *protected* de la classe Liste) le type du chaînon utilisé dans la liste comme suit :
typedef DataStructure::cyclicNode<T> Chainon;

Vous programmerez la totalité de la classe générique Liste en un seul fichier Liste.h.

Voici sous forme informelle la spécification de la liste :

1.1 opérations de base

- initialiser une liste vide
- détruire une liste
- empty : déterminer si la liste est vide
- size : donner le nombre d'éléments de la liste
- front, back : accès (modifiable et non modifiable) au premier, au dernier élément
- push_front, push_back : insérer un élément en tête, en fin
- pop_front, pop_back : supprimer le premier, le dernier élément

Tous les paramètres et tous les résultats seront passés *par référence* (sauf les types scalaires prédéfinis du langage).

Bien sûr, pour toutes ces opérations, il vous est demandé *une spécification précise* sous forme de commentaire ; les pré-conditions, judicieusement définies, seront testées avec assert.

1.2 test des opérations

Pour tester cette première partie, utilisez le programme de test unitaire donné.

2 Opérations de parcours

On veut parcourir la liste au moyen d'itérateurs externes ; un *itérateur* est un objet associé à une liste qui permet de désigner les éléments de la liste et d'y accéder (en consultation ou en modification).

Les opérations définies sur un itérateur sont similaires à celles d'un pointeur : incrémentation, décrémentation pour passer d'un élément au suivant (précédent), indirection (*) pour accéder à l'élément désigné, indirection (->) pour obtenir l'*adresse* de l'élément désigné.

La liste fournit des méthodes pour créer des itérateurs..

2.1 class const_iterator : constructeur, destructeur

Définissez dans la classe Liste une classe interne const_iterator avec les attributs nécessaires.

Attention : cet itérateur ne doit permettre l'accès aux éléments de la liste qu'en *consultation seule*.

Programmez le constructeur et le destructeur.

Attention : ce constructeur doit être *privé* (ou *protégé*) afin de ne pas permettre la création d'itérateur autrement que par le mécanisme expliqué plus bas.

2.2 classe Liste : création d'itérateurs

Programmez les deux méthodes (publiques) :

```
/** renvoie un itérateur sur le début de liste
 *  cet itérateur désigne le premier élément de la liste si elle n'est pas vide ;
 *  sinon, il désigne la même position que l'itérateur renvoyé par end()
 */
const_iterator begin(void) const;

/** renvoie un itérateur qui désigne une position située après le dernier élément
 */
const_iterator end(void) const;
```

Afin de donner à la classe Liste (à l'exclusion de toute autre) la possibilité de créer un itérateur, on va déclarer la classe Liste *amie* de la classe const_iterator en ajoutant la déclaration suivante dans la partie privée de la classe const_iterator : **friend class Liste;**

2.3 class const_iterator : opérations

1. Complétez la définition de la classe avec les opérateurs suivants, sous forme de méthodes :

```
/** opérateur ++ préfixé
 *  positionne l'itérateur sur l'élément suivant
 *  @pre l'itérateur désigne une position valide dans la liste(≠ end())
 *  @return nouvelle valeur de l'itérateur
 */
const_iterator & operator ++(void);

/** opérateur -- préfixé
 *  positionne l'itérateur sur l'élément précédent
 *  @pre l'itérateur ne désigne pas l'élément de tête (≠ begin())
 *  @return nouvelle valeur de l'itérateur
 */
const_iterator & operator --(void);

/** opérateur d'indirection * (accès NON modifiable)
 *  @pre l'itérateur désigne une position valide dans la liste(≠ end())
 *  @return valeur de l'élément désigné par l'itérateur
 */
const T & operator * (void) const;


/** opérateur d'indirection -> (accès NON modifiable)
 *  @pre l'itérateur désigne une position valide (≠ end())
 *  @return adresse de l'élément désigné par l'itérateur
 */
const T * operator -> (void) const;
```

2. Surchargez les *opérateurs de comparaison d'itérateurs* == et != sous forme de méthodes.

Testez le fonctionnement de votre itérateur.

2.4 classe iterator

Programmez la classe iterator sur le même modèle que la classe const_iterator (voir les parties 2.1, 2.2 et 2.3) ; cet itérateur doit permettre un accès *modifiable* aux éléments de la liste.

Testez.

3 Autres opérations sur la liste

3.1 opérations avec itérateur

Programmez les opérations suivantes :

1. *fonction* find qui cherche une valeur dans une séquence définie par deux itérateurs :

```
/**
   3.1 chercher un élément dans la séquence [premier, dernier[
   @param premier : début de la séquence
   @param dernier : fin de séquence
   @param x       : valeur cherchée
   @return itérateur qui désigne x s'il est trouvé ;
           cet itérateur est égal à dernier si x est absent
*/
template <class InputIterator, class T>
InputIterator
find(   InputIterator premier,
        InputIterator dernier,
        const T & x);
```

2. *méthode* insert qui insère un nouvel élément, dont la valeur est donnée en paramètre, *avant* l'élément désigné par l'itérateur donné en paramètre ; si l'itérateur donné vaut end(), l'ajout se fait en fin de liste ; cette méthode rend un itérateur qui désigne l'élément inséré.

```
iterator insert ( iterator position, const T & x );
```

3. *méthode* erase qui supprime l'élément désigné (qu'on suppose exister) par l'itérateur passé en paramètre ; cette méthode rend un itérateur qui désigne l'élément qui *suit* celui supprimé.

Testez ces nouvelles opérations.

3.2 opérations sur des listes

Programmez les fonctions suivantes dans le fichier copier.h :

1. une fonction qui cherche une valeur dans une liste triée par valeurs croissantes ; cette fonction doit rendre un itérateur sur le premier élément de valeur $\geq$ à la valeur cherchée, si un tel élément existe ; dans le cas contraire, l'itérateur rendu est l'itérateur end().
2. une fonction qui rend l'adresse d'une copie triée par valeurs croissante d'une liste passée en paramètre (par référence).
   Remarque : il n'est pas stupide d'utiliser ici la fonction précédente.

Testez ces nouvelles opérations.

4 Mécanique et opérateurs

Programmez les méthodes et opérateurs suivants dans la classe Liste :

1. *constructeur de copie* qui initialise l'instance courante avec une copie profonde de la liste paramètre.
2. *opérateur d'affectation*
3. *opérateur de concaténation* : surchargez l'opérateur + pour lui donner la signification suivante : l1 + l2 est une nouvelle liste qui contient les éléments de l1 suivis de ceux de l2.

   Attention : Il doit être possible d'effectuer la concaténation d'un nombre quelconque de listes sans fuite de mémoire... Exemple : l4 = l1 + l2 + l3.

   Remarque : En réfléchissant, on s'aperçoit que ces trois méthodes, ainsi que le constructeur et le destructeur font appel à des traitement communs ; plutôt que de programmer plusieurs fois la même fonctionnalité, il vous est demandé de créer des méthodes privées qui seront judicieusement utilisées par les méthodes citées ci-dessus.
4. *opérateur d'affichage* $\ll$ qui affiche une liste.

Testez ces nouvelles opérations.

5 Détection des erreurs de gestion de la mémoire

valgrind est un outil puissant de débogage de programme ; il possède en particulier un outil efficace de détection des erreurs de gestion de mémoire (memcheck).

Pour l'utiliser, il faut ouvrir un terminal, se placer dans le répertoire de votre projet et taper la commande :

valgrind --tool=memcheck --leak-check=yes ./Release/p

en supposant que votre programme s'appelle p et est placé dans le sous-répertoire Release de votre projet.