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. À chaque étape du TP, vous devrez valider votre gestion de mémoire, soit avec l'allocateur mémoire du premier TP soit 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;

Pour simplifier la réalisation, vous programmerez la totalité de la classe générique Liste en un seul fichier liste.hh.

La spécification de la liste diffère quelque peu de celle utilisée au premier semestre ; la voici sous forme informelle :

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 (avec la syntaxe « normalisée » vue au premier semestre) ; les pré-conditions, judicieusement définies, seront testées avec assert.

1.2 test des opérations

Dans le fichier testListe.cc qui vous est donné se trouve un programme destiné à tester toutes les opérations qu'il vous sera demandé de programmer ; pour l'utiliser, vous devrez ôter les commentaires des tests qui correspondent aux parties programmées.

Pour tester cette première partie, utilisez la procédure testPartie1.

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 qui désignent certains éléments.

2.1 class const_iterator : constructeur, destructeur

Définissez dans la classe Liste une classe interne const_iterator à deux attributs :
- pointeur vers la liste associée à l'itérateur,
- pointeur vers l'élément courant désigné par l'itérateur.

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

Programmez les deux méthodes suivantes :

1. constructeur (à deux paramètres) qui initialise un itérateur ; 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. destructeur

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;
```

Ces deux méthodes font appel au constructeur de la classe const_iterator qui est en accès privé (ou protégé).

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 désigne une position valide dans la liste(≠ end())
 * @return nouvelle valeur de l'itérateur
 */
const_iterator & operator --(void);

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

/** 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);
```

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

2.4 client de test

Testez le fonctionnement de votre itérateur avec la procédure testPartie2.

2.5 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 avec la procédure testPartie2bis.

3 Autres opérations sur la liste

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 T>
typename  Liste<T>::iterator
find(    typename Liste<T>::iterator premier,
         typename Liste<T>::iterator dernier,
         const T & x);
```

   Remarque : lorsqu'une fonction générique, une méthode générique ou une classe générique utilise un type interne à une classe elle-même générique (exemple : Liste<T>::iterator), la norme du langage exige de faire précéder l'utilisation de ce type du mot clé typename.

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.

3.1 client de test

Testez ces nouvelles opérations (testPartie3), puis ajoutez-y les fonctions suivantes :

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).

Complétez votre programme client pour tester ces 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 résultat de la concaténation des 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.

Il ne vous reste plus qu'à tester 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 ./Debug/p en supposant que votre programme s'appelle p et est placé dans le sous-répertoire Debug de votre projet.