## Partie 2 : std::shared\_ptr & Liste simplement chaînée

Nous nous intéressons à un containeur que nous implantons comme une liste simplement chaînée.

#include<exception>  
#include<iostream>  
#include<memory>  
  
template<typename T>  
class List  
{  
private:  
 struct Node  
 {  
 private:  
 T m\_value;  
 Node\* m\_next\_node;  
  
 public:  
 Node(T aValue): Node(aValue, NULL)   
 {}  
 Node(T aValue, Node\* theNextNode):  
 m\_value(aValue), m\_next\_node(theNextNode)  
 {}  
   
 void insert\_after(T aValue)  
 {  
 m\_next\_node = new Node(aValue, m\_next\_node);  
 }  
 Node\* next() { return m\_next\_node; }  
 T& value() { return m\_value; }  
 T value() const { return m\_value; }  
 };  
  
 Node\* m\_front;  
 Node\* m\_back;  
  
public:  
 List(): m\_front(NULL), m\_back(NULL) {}  
 void push\_front(T value)  
 {  
 if(m\_front == NULL)  
 {  
 m\_front = new Node(value);  
 m\_back = m\_front;  
 }  
 else  
 m\_front = new Node(value, m\_front);  
 }  
 void push\_back(T value)  
 {  
 if(m\_back == NULL)  
 {  
 m\_front = new Node(value);  
 m\_back = m\_front;  
 }  
 else  
 {  
 m\_back->insert\_after(value);  
 m\_back = m\_back->next();  
 }  
 }  
};

### Question n°1 :

Proposez un destructeur qui détruit bien tous les éléments stockés dans la liste avant de détruire la liste.

Correction

Nous devons parcourir l’ensemble des nœuds qui ont été créés et les détruire un par un.

~List()  
 {  
 for(auto m\_current = m\_front; m\_current != NULL; )  
 {  
 auto m\_next = m\_current->next();  
 delete m\_current;  
 m\_current = m\_next;  
 }  
 }

### Question n°2 :

Nous aimerions ajouter des itérateurs. Pour ce faire, nous proposons la classe suivante d’itérateurs qui fait référence au nœud de la liste.

class List  
{  
 public:  
 ...  
 class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 Node\* m\_current;  
 public:   
 iterator(Node\* node): m\_current(node)  
 {}  
 iterator& operator++()  
 {  
 if(m\_current != NULL)  
 m\_current = m\_current .next();  
 return \*this;  
 }  
 iterator operator++(int)  
 {  
 auto result = iterator(\*this);  
 if(m\_current != NULL)  
 m\_current = m\_current->next();  
 return result;  
 }  
 reference operator \*()  
 {  
 return m\_current->value();  
 }  
 pointer operator ->()  
 {  
 return &m\_current.value();  
 }  
 bool operator == (const iterator& another) { return m\_current == another.m\_current; }  
 bool operator != (const iterator& another) { return m\_current != another.m\_current; }  
 };  
...  
}

Ajouter les méthodes begin et end à la liste.

Correction

Le code est relativement simple et se réduit à 👍

iterator begin() { return iterator(m\_front); }  
 iterator end() { return iterator(NULL); }

### Question n°3 :

Est-il possible d’utiliser des std::unique\_ptr en lieu et place des pointeurs natifs de C++, expliquer pourquoi ?

Correction

Il n’est pas possible d’utiliser les std::unique\_ptr pointeurs puisque nous avons deux références sur le dernier élément de la liste, la référence venant du nœud précédent le dernier et la référence stockée dans le champ m\_back.

De même, les itérateurs font références à des nœuds de la liste, ce qui veut dire que pour un nœud, nous pouvons avoir plusieurs nœuds de la liste.

### Question n°4 :

Est-il possible d’utiliser des std::shared\_ptr en lieu et place des pointeurs natifs de C++, expliquer pourquoi ?

Correction

Les std::shared\_ptr supportent plusieurs références concurrentes et peuvent faire références aux éléments.

### Question n°5 :

Réécrivez la classe avec des std::shared\_ptr. Avons-nous toujours besoin d’un destructeur ?

Correction

Il n’est plus besoin de constructeurs sachant que la destruction de toutes les références à un nœud va entrainer la destruction de ce noeud.

### Question n°6 :

Nous aimerons passer à une structure bidirectionnelle. Expliquer la difficulté d’une structure bidirectionnelle et proposez une solution basée sur les std::shared\_ptr.