## 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 de la liste, nous pouvons avoir plusieurs références sur ce nœud.

### 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 à un élément. En conséquence, il est possible d’utiliser des std::shared\_ptr au lieu et place de pointeurs natifs de C++. Ces pointeurs autorisent un accès aussi rapide que les pointeurs standards de C++ et garantissent que l’élément référencé ne sera détruit uniquement au moment de la destruction de la dernière référence sur celui-ci.

### Question n°5 :

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

Correction

Il suffit de remplacer dans le code précédent chacune des références à un pointeur sur un élément de T par un std::shared\_ptr<T>.

De plus, il n’est plus besoin de destructeur sachant que la suppression de toutes les références à un nœud va entrainer automatiquement la destruction de ce nœud.

Il est cependant à remarquer que cette stratégie de gestion de la mémoire peut induire un changement de comportement du logiciel. Si un itérateur faisant référence à un élément de la liste continue à exister après la destruction de la liste, l’élément de la liste référencée par cet itérateur continuera à être présent en mémoire tant que l’itérateur ne sera pas détruit.

La suppression du destructeur ainsi que le remplacement des pointeurs natifs T\* par des std::shared\_ptr<T> conduit au code suivant :

template<typename T>  
class List  
{  
private:  
 struct Node  
 {  
 private:  
 T m\_value;  
 std::shared\_ptr<Node> m\_next\_node;  
  
 public:  
 Node(T aValue):   
 m\_value(aValue), m\_next\_node()  
 {}  
 Node(T aValue, std::shared\_ptr<Node>& theNextNode):  
 m\_value(aValue), m\_next\_node(theNextNode)  
 {}  
   
 void insert\_after(T aValue)  
 {  
 m\_next\_node = std::make\_shared<Node>(aValue, m\_next\_node);  
 }  
 std::shared\_ptr<Node>& next() { return m\_next\_node; }  
 T& value() { return m\_value; }  
 T value() const { return m\_value; }  
 };  
  
 std::shared\_ptr<Node> m\_front;  
 std::shared\_ptr<Node> m\_back;  
  
 class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 std::shared\_ptr<Node> m\_current;  
 public:   
 iterator(): m\_current() {}  
 iterator(std::shared\_ptr<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; }  
 };  
public:  
 List(): m\_front(NULL), m\_back(NULL) {}  
 iterator begin() { return iterator(m\_front); }  
 iterator end() { return iterator(); }  
 void push\_front(T value)  
 {  
 if(m\_front == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 m\_front = std::make\_shared<Node>(value, m\_front);  
 }  
 void push\_back(T value)  
 {  
 if(m\_back == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 {  
 m\_back->insert\_after(value);  
 m\_back = m\_back->next();  
 }  
 }  
};

### Question n°6 :

Nous souhaitons désormais imposer une sémantique qui est assez habituelle lorsque nous manipulons des itérateurs. Un itérateur est invalidé quand la collection sous-jacente est modifiée. La technique la plus classique pour déterminer si un containeur a été modifiée, consister à ajouter un numéro de version au containeur, souvent représenté par un nombre entier non-signé, qui est incrémenté chaque fois que le containeur est modifié. Lors que l’itérateur cherche à accéder à un élément, il vérifie que le numéro de version correspond au numéro de version qu’il a obtenu lors de sa création, si ce n’est pas le cas, l’itérateur déclenche une exception.

### Question n°6.1 :

Modifier le code du containeur List pour intégrer les fonctionnalités suivantes :

* List doit définir un champ qui contient le numéro de version du containeur. Le champ est supporté être à zéro lors de l’initialisation du containeur.
* Chacune des opérations qui modifient le containeur doit incrémenter le numéro de version.
* Chaque fois qu’un itérateur essaye d’accéder à un élément de la collection, il doit vérifier que le numéro de version est bien défini, sinon, il doit générer une exception que nous avons défini comme étant de type invalid\_iterator et dérivant de std::exception.

Correction

Nous définissons dans un premier temps une nouvelle classe invalid\_iterator :

class invalid\_iterator: std::exception  
{  
public:  
 invalid\_iterator() {}  
 invalid\_iterator(const char\* const &aMessage):  
 exception(aMessage) {}  
};

Ensuite, nous ajoutons à la classe List, le numéro de version :

template<typename T>  
class List:  
{  
private:  
 ...  
 using version\_type = unsigned;  
 ...  
 version\_type m\_version = 0ul;  
  
public:  
  
};

Les méthodes modifiant la liste List doivent désormais incrémentée le numéro de version m\_version.

template<typename T>  
class List:  
{  
 ...  
public:  
 void push\_front(T value)  
 {  
 if(m\_front == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 m\_front = std::make\_shared<Node>(value, m\_front);  
 m\_version ++;  
 }  
 void push\_back(T value)  
 {  
 if(m\_back == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 {  
 m\_back->insert\_after(value);  
 m\_back = m\_back->next();  
 }  
 m\_version ++;  
 }  
};

Une fois ces modifications faites, il faut modifier la classe itérateur List<T>::iterator. Désormais cette classe doit stocker une référence à la liste ainsi que doit initialiser un champ m\_version avec le numéro de version définie par la liste au moment de la création de l’itérateur.

Il faut en conséquence ajouter les deux champs et modifier les constructeurs comme suit :

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 std::shared\_ptr<Node> m\_current;  
 const List<T>& m\_list;  
 typename List<T>::version\_type m\_version;  
 ...  
 public:  
 iterator(const List<T>& theList):  
 m\_list(theList), m\_current(),   
 m\_version(theList.m\_version) {}  
 iterator(const List<T>& theList,   
 std::shared\_ptr<Node>& node):  
 m\_list(theList), m\_current(node),   
 m\_version(theList.m\_version) {}  
 {}  
 ...  
 };

Ensuite, chaque méthode de l’itérateur doit vérifier que l’itérateur est valide. Pour factoriser le code, nous proposons de définir une méthode check\_if\_is\_valid qui vérifie que l’itérateur est bien valide et si ce n’est le cas lève l’exception invalid\_iterator.

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 ...  
 void check\_if\_is\_valid()  
 {  
 if(m\_version != m\_list.m\_version)  
 throw invalid\_iterator();  
 }  
 ...  
 public:  
 ...  
 };

Enfin pour chacun que méthodes qui manipulent l’itérateur, comme les opérateurs d’accès à l’élément, les opérateurs d’incrémentation, il est nécessaire d’appeler la méthode privée check\_if\_is\_valid pour s’assurer que l’itérateur est valide.

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 ...  
 iterator& operator++()  
 {  
 check\_if\_is\_valid();  
 if(m\_current != NULL)  
 m\_current = m\_current->next();  
 return \*this;  
 }  
 ...  
 reference operator \*()  
 {  
 check\_if\_is\_valid();  
 return m\_current->value();  
 }  
 ...  
 };

Enfin, nous devons modifier les appels aux constructeurs dans les méthodes begin et end de la liste List.

template<typename T>  
class List:  
{  
 ...  
public:  
 ...  
 iterator begin() { return iterator(\*this, m\_front); }  
 iterator end() { return iterator(\*this); }  
 ...  
};

Ce qui termine l’ensemble des modifications nécessaires pour implanter ce nouveau comportement.

Ceci nous donne en conséquence le code consolidé suivant :

class invalid\_iterator: std::exception  
{  
public:  
 invalid\_iterator() {}  
 invalid\_iterator(const char\* const &aMessage):  
 exception(aMessage) {}  
};  
  
template<typename T>  
class List  
{  
private:  
 struct Node  
 {  
 private:  
 T m\_value;  
 std::shared\_ptr<Node> m\_next\_node;  
  
 public:  
 Node(T aValue):   
 m\_value(aValue), m\_next\_node()  
 {}  
 Node(T aValue, std::shared\_ptr<Node>& theNextNode):  
 m\_value(aValue), m\_next\_node(theNextNode)  
 {}  
   
 void insert\_after(T aValue)  
 {  
 m\_next\_node = std::make\_shared<Node>(aValue, m\_next\_node);  
 }  
 std::shared\_ptr<Node>& next() { return m\_next\_node; }  
 T& value() { return m\_value; }  
 T value() const { return m\_value; }  
 };  
  
 using version\_type = unsigned;  
  
 std::shared\_ptr<Node> m\_front;  
 std::shared\_ptr<Node> m\_back;  
 version\_type m\_version;  
  
 class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 std::shared\_ptr<Node> m\_current;  
 const List<T>& m\_list;  
 typename List<T>::version\_type m\_version;  
 void check\_if\_is\_valid()  
 {  
 if(m\_version != m\_list.m\_version)  
 throw invalid\_iterator();  
 }  
  
 public:   
 iterator(const List<T>& theList):  
 m\_list(theList), m\_current(),   
 m\_version(theList.m\_version) {}  
 iterator(const List<T>& theList,   
 std::shared\_ptr<Node>& node):  
 m\_list(theList), m\_current(node),   
 m\_version(theList.m\_version) {}  
 iterator& operator++()  
 {  
 check\_if\_is\_valid();  
 if(m\_current != NULL)  
 m\_current = m\_current->next();  
 return \*this;  
 }  
 iterator operator++(int)  
 {  
 check\_if\_is\_valid();  
 auto result = iterator(\*this);  
 if(m\_current != NULL)  
 m\_current = m\_current->next();  
 return result;  
 }  
 reference operator \*()  
 {  
 check\_if\_is\_valid();  
 return m\_current->value();  
 }  
 pointer operator ->()  
 {  
 check\_if\_is\_valid();  
 return &(m\_current->value());  
 }  
 bool operator == (const iterator& another)   
 {   
 return &m\_list == &another.m\_list   
 && m\_version == another.m\_version   
 && m\_current == another.m\_current;   
 }  
 bool operator != (const iterator& another)   
 {   
 return &m\_list == &another.m\_list   
 && m\_version == another.m\_version   
 && m\_current != another.m\_current;   
 }  
 };  
public:  
 List(): m\_front(), m\_back(), m\_version(0) {}  
 iterator begin() { return iterator(\*this, m\_front); }  
 iterator end() { return iterator(\*this); }  
 void push\_front(T value)  
 {  
 if(m\_front == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 m\_front = std::make\_shared<Node>(value, m\_front);  
 m\_version ++;  
 }  
 void push\_back(T value)  
 {  
 if(m\_back == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 {  
 m\_back->insert\_after(value);  
 m\_back = m\_back->next();  
 }  
 m\_version ++;  
 }  
};

### Question n°6.2 :

Cependant, le code précédent n’est pas optimal. En effet, quand le containeur est détruit et qu’un itérateur faisant référence au containuer existe encore, l’itérateur va pour pouvoir d’abord vérifier le numéro de version. Cependant si la liste a été supprimé, le champ correspondant au numéro de version n’est plus accessible et nous aurons une erreur d’accès qui va se produire à ce moment.

Pour ce faire, nous souhaitons pouvoir garantir deux comportements :

* détruire l’ensemble des éléments stockés dans la liste (ou le containeur) quand le containeur est détruit.
* déterminer si la liste (ou le containeur) a été détruit.

C++ introduit une classe dénommée [std::weak\_ptr](https://en.cppreference.com/w/cpp/memory/weak_ptr). À la différence de std::unique\_ptr ou std::shared\_ptr, un std::weak\_ptr ne garantit pas que l’élément auquel il fait référence est accessible. Pour pouvoir accéder à un objet, il faut verrouiller l’accès en utilisant la méthode [lock](https://en.cppreference.com/w/cpp/memory/weak_ptr/lock)/ Cette méthode retourne un std::shared\_ptr qui soit fait référence à l’élément si cette référence est encore actuelle ou fait référence à NULL si la référence est devenue obsolète. La méthode [expired](https://en.cppreference.com/w/cpp/memory/weak_ptr/expired) indique si la référence à l’élément est devenue obsolète.

Désormais, si nous souhaitons implanter un itérateur qui détermine si la collection n’a pas été modifiée, cela impose :

1. Tester si la référence est devenue obsolète. Si la référence est devenue obsolète, cela signifie que l’élément référencé a été retiré de la collection ou a été modifié. Dans ce cas, l’itérateur n’est plus valide.
2. Tester si le numéro de version de la liste est différent de celui obtenu lors de la création de la liste. Si tel est le cas, l’itérateur n’est plus valide.

Correction

La solution consiste à utiliser dans la classe iterator un std::weak\_ptr en lieu et place d’un std::shared\_ptr pour faire référence au nœud de la liste.

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 std::weak\_ptr<Node> m\_current;  
 const List<T>& m\_list;  
 typename List<T>::version\_type m\_version;  
 ...  
 };

Nous modifions ensuite le code de la méthode check\_if\_is\_valid pour tester en premier si la référence au nœud est toujours valide. Si elle n’est plus valide, cela signifie soit que le nœud a été supprimé, soit que la collection a été détruite.

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
{  
private:  
 ...  
 void check\_if\_is\_valid()  
 {  
 if(m\_current.expired() || m\_version != m\_list.m\_version)  
 throw invalid\_iterator();  
 }  
 ...  
};

Il faut ensuite modifier les méthodes qui accèdent au nœud pour obtenir un std::shared\_ptr le temps nécessaire pour effectuer l’opération. Ce std::shared\_ptr sera automatiquement détruit au plus tard à la fin de la méthode.

Ceci donne par exemple les méthodes suivantes :

class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 ...  
 public:  
 ...  
 iterator operator++(int)  
 {  
 check\_if\_is\_valid();  
 auto result = iterator(\*this);  
 auto current\_ptr = m\_current.lock();  
 if(current\_ptr != NULL)  
 m\_current = current\_ptr->next();  
 return result;  
 }  
 ...  
 reference operator \*()  
 {  
 check\_if\_is\_valid();  
 return m\_current.lock()->value();  
 }  
 ...  
 };

Et cela termine l’ensemble des modifications à réaliser pour implanter le nouveau comportement.

Ceci donne le code consolidé suivant :

class invalid\_iterator: std::exception  
{  
public:  
 invalid\_iterator() {}  
 invalid\_iterator(const char\* const &aMessage):  
 exception(aMessage) {}  
};  
  
template<typename T>  
class List  
{  
private:  
 struct Node  
 {  
 private:  
 T m\_value;  
 std::shared\_ptr<Node> m\_next\_node;  
  
 public:  
 Node(T aValue):   
 m\_value(aValue), m\_next\_node()  
 {}  
 Node(T aValue, std::shared\_ptr<Node>& theNextNode):  
 m\_value(aValue), m\_next\_node(theNextNode)  
 {}  
   
 void insert\_after(T aValue)  
 {  
 m\_next\_node = std::make\_shared<Node>(aValue, m\_next\_node);  
 }  
 std::shared\_ptr<Node>& next() { return m\_next\_node; }  
 T& value() { return m\_value; }  
 T value() const { return m\_value; }  
 };  
  
 using version\_type = unsigned;  
  
 std::shared\_ptr<Node> m\_front;  
 std::shared\_ptr<Node> m\_back;  
 version\_type m\_version;  
  
 class iterator:   
 std::iterator<std::forward\_iterator\_tag, T>  
 {  
 private:  
 std::weak\_ptr<Node> m\_current;  
 const List<T>& m\_list;  
 typename List<T>::version\_type m\_version;  
 void check\_if\_is\_valid()  
 {  
 if(m\_current.expired() || m\_version != m\_list.m\_version)  
 throw invalid\_iterator();  
 }  
  
 public:   
 iterator(const List<T>& theList):  
 m\_list(theList), m\_current() {}  
 iterator(const List<T>& theList,   
 std::shared\_ptr<Node>& node):  
 m\_list(theList), m\_current(node)  
 {}  
 iterator& operator++()  
 {  
 check\_if\_is\_valid();  
 auto current\_ptr = m\_current.lock();  
 if(current\_ptr != NULL)  
 m\_current = current\_ptr->next();  
 return \*this;  
 }  
 iterator operator++(int)  
 {  
 check\_if\_is\_valid();  
 auto result = iterator(\*this);  
 auto current\_ptr = m\_current.lock();  
 if(current\_ptr != NULL)  
 m\_current = current\_ptr->next();  
 return result;  
 }  
 reference operator \*()  
 {  
 check\_if\_is\_valid();  
 return m\_current.lock()->value();  
 }  
 pointer operator ->()  
 {  
 check\_if\_is\_valid();  
 return &(m\_current.lock()->value());  
 }  
 bool operator == (const iterator& another)   
 {   
 return &m\_list == &another.m\_list   
 && m\_version == another.m\_version   
 && m\_current == another.m\_current;   
 }  
 bool operator != (const iterator& another)   
 {   
 return &m\_list != &another.m\_list   
 && m\_version != another.m\_version   
 && m\_current.lock() != another.m\_current.lock();   
 }  
 };  
public:  
 List(): m\_front(), m\_back(), m\_version(0) {}  
 iterator begin() { return iterator(\*this, m\_front); }  
 iterator end() { return iterator(\*this); }  
 void push\_front(T value)  
 {  
 if(m\_front == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 m\_front = std::make\_shared<Node>(value, m\_front);  
 m\_version ++;  
 }  
 void push\_back(T value)  
 {  
 if(m\_back == NULL)  
 {  
 m\_front = std::make\_shared<Node>(value);  
 m\_back = m\_front;  
 }  
 else  
 {  
 m\_back->insert\_after(value);  
 m\_back = m\_back->next();  
 }  
 m\_version ++;  
 }  
};