TP2 - List Implementation and Iterator sub-class

Introduction

In this report, I will treat in parallel 2 different implementations of a list in c++. One implementation is a recursive implementation with a simple chaining and the second is an iterative implementation with a double chaining. To improve the reusability of these classes, I will create a template and write all my functions in the header file. Effectively, it is not possible to link the header file using a template to an implementation file when using some compilators in c++.

Members of the class

Recursive version

In the recursive implementation, there is 2 the members are the value of the node `val` and a pointer on the rest of the list `reste`.

Header file :

```
template <class T>
class Liste {
    T val; // tete de liste = element
    Liste* reste; // suivi d'une liste
    [...]
};
```

Node version

In the iterative version, we use a basic class called `Maillon` which represent a node of the list. The `Liste` class is then composed of a pointer on a `Maillon` which is the head of the list. It is then composed of a pointer on another node which is the end of the list.

Header file :

```
template <class T>
template <class T>
struct Maillon {
    T val;
    Maillon* suiv;
    Maillon* prec;//si double chainage

    ~Maillon(){
        delete suiv;
    }
};
```

```
// Classe Liste
template <class T>
class Liste {
    Maillon<T>* _tete;
    Maillon<T>* _fin;
    [...]
```

Coplien form

I firstly wrote the constructors.

Constructor by default

Recursive version

Header file :

```
Liste() : reste(NULL) {}
```

Node version

Header file :

```
Liste() : _tete(NULL), _fin(NULL) {}
```

Application

Client file :

```
Liste<int> l1;// liste de integer
Liste<float> lf;// liste de float

cout << l1 << endl;
cout << lf << endl;
```

Output : For the moment, we just create empty list but we can display these list in the console.

```
( )
( )
```

Constructor by copy

Recursive version

Header file :

```
Liste(const Liste& l) {
    if(l.reste == NULL)
        reste = NULL;
    else {
        val = l.val;
        reste = new Liste(*l.reste);
    }
}
```

Node version

Header file :

```
Liste(const Liste& l){
    Maillon<T>* it_this;
    Maillon<T>* it_l;

    //Création de la tête de liste
    _tete = new Maillon<T>();
    _tete->val = l._tete->val;
    _tete->prec = NULL;

    it_this = _tete;
    it_l = l._tete->suiv;

    while(it_l != l._fin) {
        it_this->suiv = new Maillon<T>();
        it_this->suiv->val = it_l->val;
        it_this->suiv->prec = it_this;
        it_this = it_this->suiv;
        it_l = it_l->suiv;
    }
    //Création de la fin de liste
    it_this->suiv = new Maillon<T>;
    it_this->suiv->val = l._fin->val;
    _fin = it_this->suiv;
    _fin->prec = it_this;
    _fin->suiv = NULL;
}
```

The constructor by copy was far harder to design in iterative than in recursive. I created in different step :

1. I created 2 pointer on node (`it_this` and `it_l`). they will iterate on the elements of the lists without altering them.

2. I created the node that will be at the beginning of the list.
3. I went through the list `l` in an iterative way and copy all the value in the new list.
4. Finally, I created the final node and set the end pointe of the lst to this node.

Application

Client file :

```
cout << "l1 : " << l1 << endl;
Liste<int> l2 = l1;
cout << "copie de l1 (l2) : " << l2 << endl;
cout << "address of the head (l1) : " << &l1 << endl;
cout << "address of the head (l2) : " << &l2 << endl;
```

I chose to display the head of these 2 lists in order to be sur that the 2 lists are not stored in the same memory space.

Output :

```
l1 : ( 6 5 3 )
copie de l1 (l2) : ( 6 5 3 )
address of the head (l1) : 0x7ffd1ee22880
address of the head (l2) : 0x7ffd1ee22860
```

The addreses are effectively different, so we can assume that we have created a real copy.

Destructor

Recursive version

Header file :

```
~Liste() {
    delete reste;
}
```

This method work with recursivity. When we destroy the list, we will destroy the rest, then this is also a list so the destructor will be called, etc.

Node version

Header file :

```

```

```
~Liste(){
    delete _tete;
}
```

In this case the destructor is really similar to the recursive one. In fact, this destructor is also recursive but it call the destructor of the `Maillon` class :

```
~Maillon(){
    delete suiv;
}
```

We do not delete the `prec` field because by eliminating the pointer recursively, the pointer on `prec` will no longer exist. It is for the same reason that we do not delete the `_fin` field in the `Liste` class.

Affectation operator

Recursive version

Header file :

```
Liste& operator=(const Liste& l) {
    if(this != &l){
        delete reste;
        reste = NULL;
        if(l.reste == NULL)
            reste = NULL;
        else {
            val = l.val;
            reste = new Liste(*l.reste);
        }
    }
    return *this;
}
```

The affectation operator has the same behavior as the constructor by copy. We just added a check to be sure that we don't try to affect the list her own value.

Node version

Header file :

```
Liste& operator=(const Liste& l) {
    if(this != &l) {
        delete this->_tete;
        Liste l_copy(l);
```

```
        this->_tete = l_copy._tete;
        this->_fin = l_copy._fin;
        l_copy._tete = NULL;
    }
    return *this;
}
```

To avoid rewriting a big function we use a small trick. We use an instance created by copy on l with the constructor by copy. We then put the head and the end of the copy respectively in the head and the end of the `(*this)`. Finally, we set `NULL` to the head of the copy. In this way, we avoid that the destructor, called at the end of the method, destroy the list.

Application

Client file :

```
l3 = l2;

cout << "l2 : " << l2 << endl;
cout << "copie de l2 (l3): " << l3 << endl;
cout << "address of the head (l2) : " << &l2 << endl;
cout << "address of the head (l3) : " << &l3 << endl;
```

I chose to display the head of the lists for the same reasons as the constructor by copy.

Output :

```
l2 : ( 6 5 3 )
copie de l2 (l3): ( 6 5 3 )
address of the head (l2) : 0x7fffc346e730
address of the head (l3) : 0x7fffc346e720
```

Display

In order to see the result of our operations, I chose to write the operator<< method.

Recursive version

Header file :

```
friend ostream& operator<<(ostream &o, Liste &l) {
    // This flag is only here for aesthetic purpose.
    static bool flag = true;
    if (flag == true)
        o << "( ";
```

```
    if(l.reste == NULL) {
        flag = true;
        return o << ")";
    } else {
        flag = false;
        o << l.val <<" ";
        return o << *l.reste;
    }
}
```

During the elaboration of this function, I encountered an issue. Effectively, it was impossible to insert the character (at the beginning of the stream using only pure recursivity. Thanks to the functionalities of the c++ language, I managed to create a static flag which will be initialized at true at the first creation of an instance and then it will be set to true or false during the execution. The drawback of this method is the possibility to adapt it for parallelism : with multiple instances executing at the same time, this static variable can be a real problem.

Node method

Header file :

```
friend ostream& operator<<(ostream &o, Liste &l) {
    Maillon<T>* it = l._tete;
    o << "( ";
    while(it != NULL) {
        o << it->val << " ";
        it = it->suiv;
    }
    return o << ")";
}
```

It is maybe one of the easiest method in iterative because we just need to go through the list and display each element with a while loop.

Application

Client file :

```
cout << "liste l1 " << l1 << endl;
```

Output :

```
liste l1 ( 6 5 4 3 )
```

Modificator operator

Recursive version

Header file :

```
T& operator[](int pos) {
    return (reste == NULL || pos == 0) ? val : (*reste)[pos-1];
}
```

If we access the first place of the list, we display the `val` attribute, otherwise, we access to he `pos- 1` recursively.

Node version

Header file :

```
T& operator[](int pos) {
    Maillon<T>* it = _tete;
    int index = 0;
    while(index != pos) {
        it = it->suiv;
        index++;
    }
    return it->val;
}
```

We just go throgh the list with a while loop until we reach the index indicated by the `pos` parameter.

Application

Client file :

```
cout << "l3 : " << l3 << endl;
// modifier un element a une position donnee
l3[2] = 8;

// Lire un element a une position donnee
int elt = l3[2];

cout << "element " << elt << endl;
cout << "l3 : " << l3 << endl;
```

Output :

```

```

```
l3 : ( 6 5 3 )
element 8
l3 : ( 6 5 8 )
```

Length method

Recursive version

Header file :

```
int longueur(){
    return reste == NULL ? 0 : 1+reste->longueur();
}
```

If the list is empty, then we return a length of zero, otherwise, we return 1 + a recursive call to the method on the rest.

Iterative version

Header file :

```
// Longueur
int longueur() {
    int l = 0;
    Maillon<T>* it_this = _tete;
    while(it_this != NULL) {
        ++l;
        it_this = it_this->suiv;
    }
    return l;
}
```

We just go through the list in an iterator-fashion way.

Application

Client file :

```
int l = l1.longueur();
cout << "longeur de l1 " << l << endl;
cout << "liste l1 " << l1 << endl;
```

Output :

```
longeur de l1 3
liste l1 ( 6 5 3 )
```

Addition and deletion

Recursive version

These two functions were the most difficult to design because we must to fully understand the behavior of the addition and the deletion.

Addition

Header file :

```
// Ajout
void ajouter(T& val, int pos) {
    if(reste == NULL || pos == 0){
        Liste* l = new Liste();
        l->val = this->val;
        l->reste = reste;
        reste = l;
        this->val = val;
    } else
        reste->ajouter(val,pos-1);
}
```

Here if we are at the wanted position, we will create a new pointer on a list. This pointer will take the value of the current element and the `reste` attribute will point on the `reste` attribute of the current element. Then, the `reste` of the current element will point on the pointer we just created and his value will be changed by the value given in argument of the method `ajouter`. Here, if we give an index greater than the size of the list, the element will be inserted at the end of the list.

Deleletion

Header file :

```
// Suppression
void supprimer(int pos) {
    if(reste != NULL) {
        if(pos == 0) {
            val = reste->val;
            Liste* l = reste;
            reste = reste->reste;
            l->reste = NULL;
            delete l;
        } else
            reste->supprimer(pos-1);
```

```
    }
}
```

Here, we go through the list while we don't reach the pos index. Then, we will replace the value of the `val` attribute of the current element. After that, we will create a pointer which will act as a buffer for keeping the `reste` field. We then replace the address pointed by `reste` by `reste->reste`. Finally, we set the rest of our buffer to `NULL` and we delete l. We set the rest to `NULL` because otherwise deleting l, will also delete the rest and all the element after our buffer (due to the deffinition of our destructor).

Node version

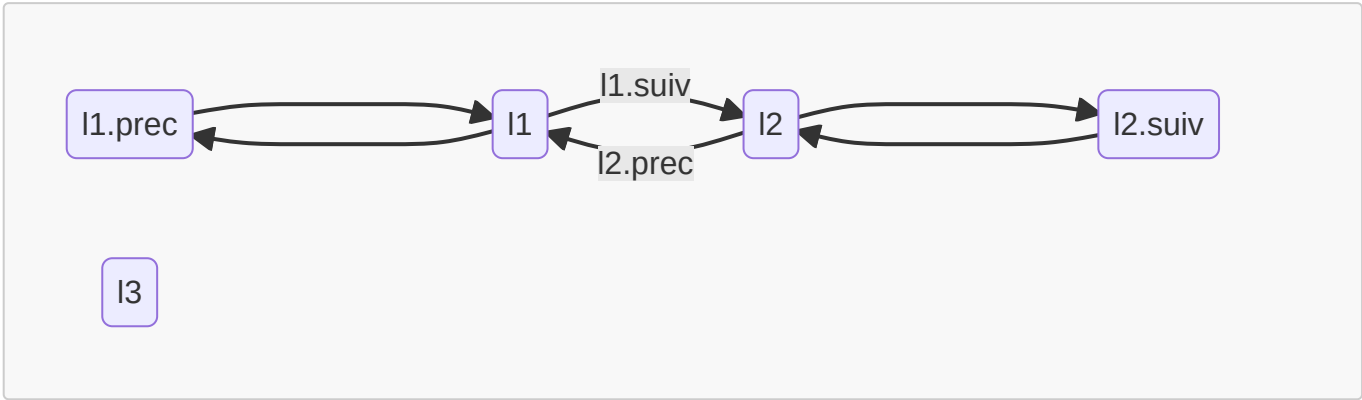
Addition

Header file :

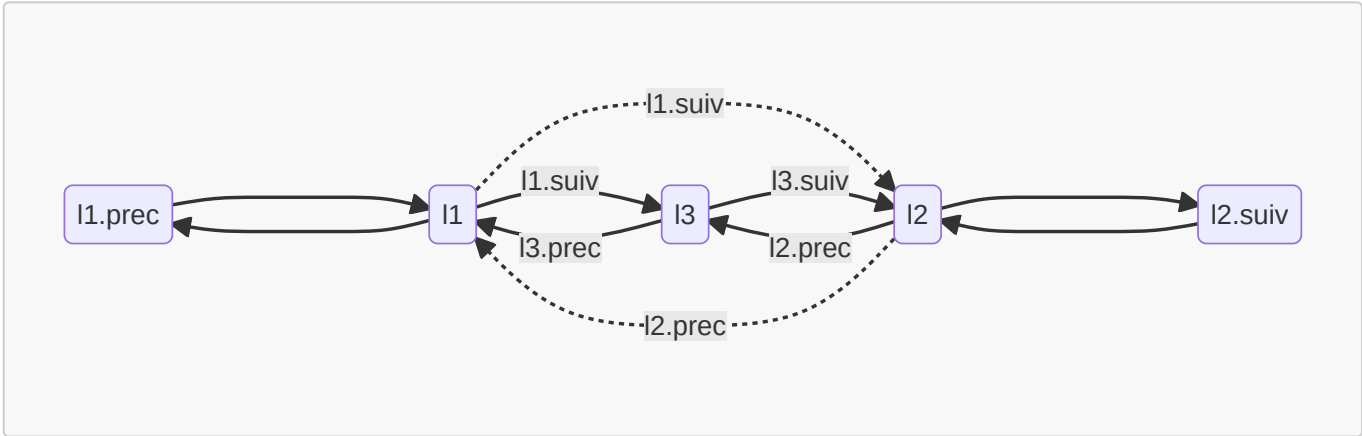
```
// Ajout
void ajouter(T& val, int pos) {
    int i = 0;
    Maillon<T>* it_this = _tete;
    Maillon<T>* nMaillon = new Maillon<T>;
    nMaillon->val = val;
    while(it_this != NULL && it_this->suiv != NULL && i != pos) {
        it_this = it_this->suiv;
        i++;
    }
    if(it_this == NULL) {
        // Case where the list is empty, we must initialize it with one
element
        this->_tete = nMaillon;
        this->_tete->prec = NULL;
        this->_tete->suiv = NULL;
        this->_fin = this->_tete;
    } else if(pos == 0) {
        // Case where we insert the element at the first position.
        nMaillon->suiv = it_this;
        nMaillon->prec = NULL;
        it_this->prec = nMaillon;
        this->_tete = nMaillon;
    } else if(it_this->suiv == NULL) {
        // Case where we reach the end of the list.
        it_this->suiv = nMaillon;
        it_this->suiv->prec = it_this;
        it_this->suiv->suiv = NULL;
        this->_fin = it_this->suiv;
    } else {
        // Case where we insert the element at the 'pos' position.
        nMaillon->suiv = it_this;
        nMaillon->prec = it_this->prec;
        it_this->prec->suiv = nMaillon;
        it_this->prec = nMaillon;
```

```
    }
}
```

To summarize what we do in this function, we can draw the following graph :



Then, when we insert l3, we must do all this links (the dotted arrows are links that are removed).



We then just treat some sepcific cases. By example, we cannot take the `suiv` element of the `NULL` pointer, that is why we cannot use the `it_this->prec->suiv` call in this specific case.

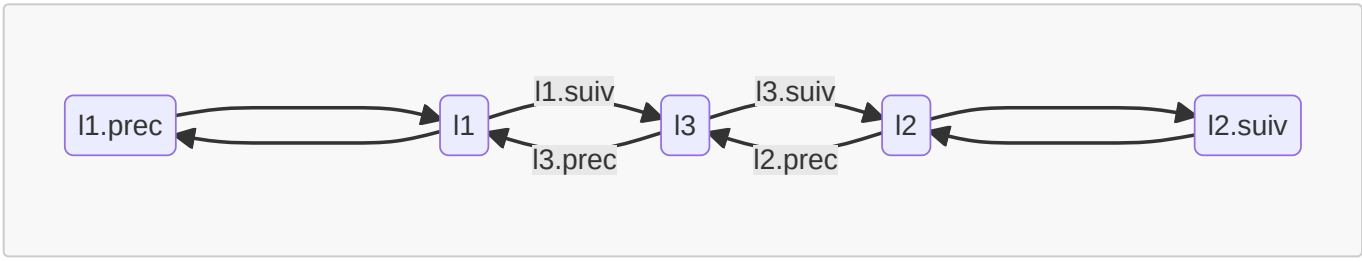
Deleletion

Header file :

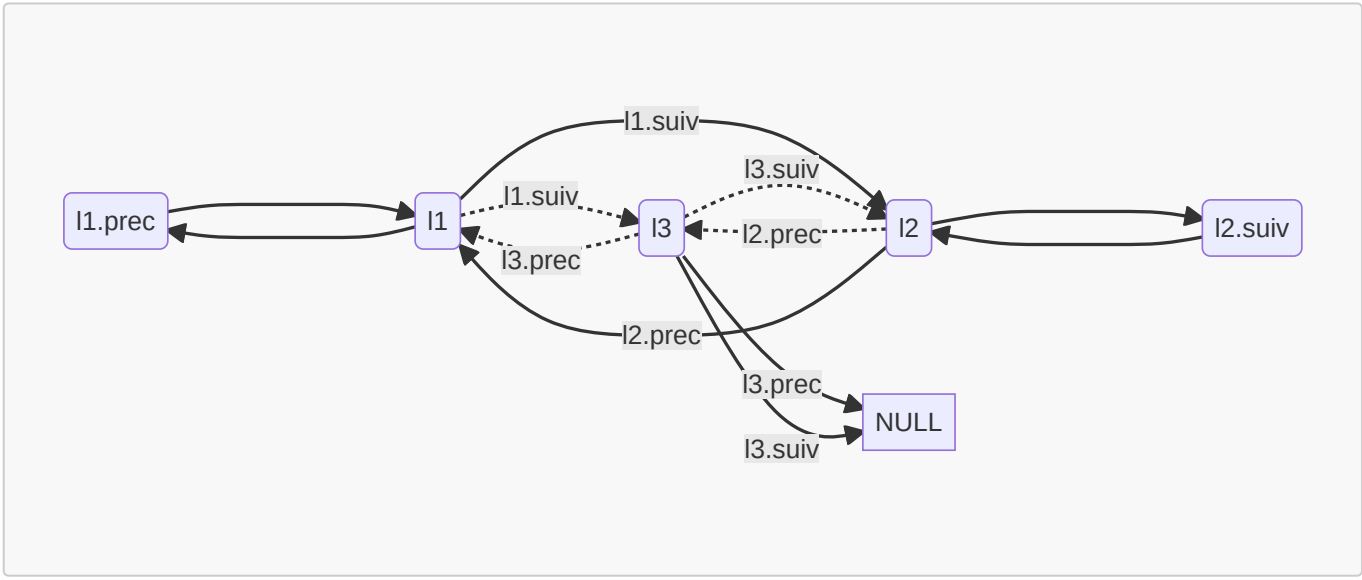
```
// Suppression
void supprimer(int pos) {
    int i = 0;
    Maillon<T>* it_this = _tete;
    while(it_this != NULL && it_this->suiv != NULL && i != pos) {
        it_this = it_this->suiv;
        i++;
    }
    if(pos == 0) {
        // Case where we insert the element at the first position.
        it_this->suiv->prec = NULL;
        _tete = it_this->suiv;
        it_this->suiv = NULL;
        delete it_this;
```

```
    } else if(it_this->suiv == NULL) {
        // Case where we reach the end of the list.
        it_this->prec->suiv = NULL;
        delete it_this;
    } else if(i == pos) {
        // Case where we insert the element at the 'pos' position.
        it_this->prec->suiv = it_this->suiv;
        it_this->suiv->prec = it_this->prec;
        it_this->suiv = NULL;
        it_this->prec = NULL;
        delete it_this;
    }
}
```

The behavior of this function is very similar to the addition method. We just do the same reasoning but reverse it.



We replace the link in the way described in the picture. We do not forget to nullify the `prec` and `suiv` elements of the element to delete or the call to the destructor will erase the end of the list.



Application

Client file :

```
Liste<int> l1;// liste de integer
int i = 3;
l1.ajouter(i, 0);
cout << "liste l1 " << l1 << endl;
```

```
i = 4;
l1.ajouter(i, 0);
cout << "liste l1 " << l1 << endl;
i = 5;
l1.ajouter(i, 0);
cout << "liste l1 " << l1 << endl;
i = 6;
l1.ajouter(i, 0);
cout << "liste l1 " << l1 << endl;
l1.supprimer(2); // suppression à une position donnee
cout << "liste l1 " << l1 << endl;
```

Output :

```
liste l1 ( 3 )
liste l1 ( 4 3 )
liste l1 ( 5 4 3 )
liste l4 ( 3 4 5 6 )
liste l1 ( 6 5 4 3 )
liste l1 ( 6 5 3 )
```

Concatenation

Recursive version

Auto-Concatenation

Header file :

```
// Auto-concatenation
Liste& autoConcat(Liste& l) {
    if(reste == NULL) {
        val = l.val;
        reste = new Liste(*l.reste);
    } else {
        reste->autoConcat(l);
    }
    return *this;
}
```

This function is pretty simple :

- while we doid not reach the end of the list, we call recursively the method on the rest.
- when we reach the empty list, we put `l` list in parameter in the rest `reste`.

Concatenation

Header file :

```
// Concatenation
Liste concat(Liste& l) {
    Liste l_f(*this);
    return l_f.autoConcat(l);
}
```

We simply use the auto-concatenation function on a new list created by copy.

iterative version

Auto-Concatenation

Header file :

```
// Auto-concatenation
Liste& autoConcat(Liste l) {
    Liste l_copy(l);
    _fin->suiv = l_copy._tete;
    l_copy._tete->prec = _fin;
    _fin = l_copy._fin;
    l_copy._tete = NULL;
    return *this;
}
```

In this method, we use the same trick as used in the affectation operator. In this case, we just replace the end of the current list by the beginning of the copy of `l`. When we affected all the pointers correctly, we nullify the head of the `l_copy` variable to avoid the call to the destructor.

Concatenation

Header file :

```
// Concatenation
Liste concat(Liste& l) {
    Liste this_copy(*this);
    return this_copy.autoConcat(l);
}
```

We simply use the auto-concatenation function on a new list created by copy.

Application

Client file :

```
cout << "l1 : " << l1 << endl;
cout << "l2 : " << l2 << endl;
cout << "l3 : " << l3 << endl;
l1.autoConcat(l2);
cout << "l1.autoConcat(l2) : " << l1 << endl;

l3 = l1.concat(l2);
cout << "l3 = l1.concat(l2) : " << l3 << endl;
cout << "l1 : " << l1 << endl;
```

Output :

```
l1 : ( 6 5 3 )
l2 : ( 6 5 3 )
l3 : ( 6 5 8 )
l1.autoConcat(l2) : ( 6 5 3 6 5 3 )
l3 = l1.concat(l2) : ( 6 5 3 6 5 3 6 5 3 )
l1 : ( 6 5 3 6 5 3 )
```

We can see that the `autoConcat` and the `concat` methods did not act exactly the same. The `concat` method will create a new instance with the result of the function, but the `autoConcat` method will also alter the list that called the function.

Class Iterator

Finally, to design a fully working list, I designed an iterator subclass named `ListeIterateur`.

Recursive version

Header file :

```
class ListeIterateur {
    Liste* tete;
    Liste* it;
public:
    ListeIterateur(Liste& l): tete(&l),it(&l){}
    void reset() {
        it = tete;
    }
    bool hasNext() {
        return it->reste != NULL;
    }
    T& get(){
        T& val = it->val;
        it = it->reste;
        return val;
    }
};
```

We can observe different elements in this iterator :

- *The attributes* : We have 2 attributes. One of them will be used to always point on the head of the list (the pointer `tete`). The second will iterate through the list.
- *The constructor* : To build an iterator, we must specify in the parameters of the constructor to which list the iterator will be attached. We then build the head and the iterator attributes pointing on the address of the list.
- *The reset method* : This method only make the `it` parameter point on the `tete` parameter.
- *The hasNext method* : This function check if the current element is `NULL` or not.
- *The get method* : Ths function has 2 goals. The first one is to get the value of the current element. The second one is to iterate in the list by assigning the `it->reste` pointer to the `it` pointer.

Node version

```
class ListeIterateur {
    Liste* liste;
    Maillon<T>* it;
public:
    ListeIterateur(Liste& l): liste(&l),it(liste->_tete){}
    void reset() {
        it = liste->_tete;
    }
    bool hasNext() {
        return it != NULL;
    }
    T& get(){
        T& val = it->val;
        it = it->suiv;
        return val;
    }
};
```

The functioning is really similar to the recursive version. We just replace the pointer on the head of the list by a `Liste` because it already contains a pointer on the head. We also replace the `it` attribute by a pointer on a node.

Application

Client file :

```
Liste<int>::ListeIterateur iter(l3);
int r = 0;
iter.reset();
cout << "l3 jusqu'à l'élément 3: ";
while(iter.hasNext() && r < 4) {
    cout << iter.get() << " ";
    r++;
```

```
}
cout << endl;
iter.reset();

cout << "l3 : ";
while(iter.hasNext()) {
    cout << iter.get() << " ";
}
cout << endl;
```

I use the iterator in two different case.

- In the first case, we iterate until a certain position.
- In the second case, we go through the entire list. **Output :**

```
l3 jusqu'à l'élément 3: 6 5 3 5
l3 : 6 5 3 5 3 6 5 3
```

Additional constructor

In the subject, it was ask to build a new constructor which build a list of int from a string.

Recursive version

Header file :

```
Liste(const string& s) : reste(NULL) {
    stringstream s_str(s);
    string token;
    int toAdd;

    while(getline(s_str, token, '-')){
        toAdd = stoi(token);
        ajouter(toAdd, longueur());
    }
}
```

In this method we use the `getline` function in the iostream library. It lets us tokenize our string and lets us parse this string. We just convert the token into int to add them to the list.

Node version

Header file :

```
Liste(const string& s) : _tete(NULL),_fin(NULL) {
    stringstream s_str(s);
```

```
    string token;
    int toAdd;

    while(getline(s_str, token, '-')){
        toAdd = stoi(token);
        ajouter(toAdd, longueur());
    }
}
```

It is more or less the same function than the recursive one.

Application

Client file :

```
iste<int> l4("3-4-5-6");
cout << "liste l4 " << l4 << endl;
```

Output :

```
liste l4 ( 3 4 5 6 )
```

Annexes

Makefile

To compile, I used the qmake software. It is a software developped by the Qt Company which let us generate a makefile from a `.pro` file, easier to write than a full makefile. **liste.pro :**

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

TARGET = ../tp

SOURCES += \
        main.cpp

HEADERS += \
    ListeMaillon.h \
    ListeRecursif.h
```

To compile, simply run :

```
```

```
qmake ; make
```

or simply `make` if the makefile has already been generated.

To erase all the unnecessary file run :

```
make distclean
```

or only `make clean` if you want to keep the makefile and the qmake stash.

ListeRecursif

Header file

```
#ifndef __LISTE_RECURSIF_H
#define __LISTE_RECURSIF_H
/*
 * Creer une classe liste generique (avec template)
 * avec une implentation en recursif.
 *
 * Auteur : JC. Creput - A. Viala
 * 23/03/2022
 */
#include <iostream>
#include <new>
#include <sstream>
#include <string>
#include <cstring>

using namespace std;

// Classe liste
template <class T>
class Liste {
    T val; // tete de liste = element
    Liste* reste; // suivi d'une liste

public:
    // Forme canonique de Coplien (4 operations)
    // Cons. par defaut
    Liste() : reste(NULL) {} //liste vide, val est bidon
    // Cons. copie
    Liste(const Liste& l) {
        if(l.reste == NULL)
            reste = NULL;
        else {
            val = l.val;
            reste = new Liste(*l.reste);
```

```
        }
    }
    // Destructeur
    ~Liste() {
        delete reste;
    }
    // Affectation
    Liste& operator=(const Liste& l) {
        if(this != &l){
            delete reste;
            reste = NULL;
            if(l.reste == NULL)
                reste = NULL;
            else {
                val = l.val;
                reste = new Liste(*l.reste);
            }
        }
        return *this;
    }

    //Constructeur par un string d'une liste d'entiers
    Liste(const string& s) : reste(NULL) {
        stringstream s_str(s);
        string token;
        int toAdd;

        while(getline(s_str, token, '-')){
            toAdd = stoi(token);
            ajouter(toAdd, longueur());
        }
    }
    // Modificateur
    T& operator[](int pos) {
        return (reste == NULL || pos == 0) ? val : (*reste)[pos-1];
    }
    // Longueur
    int longueur(){
        return reste == NULL ? 0 : 1+reste->longueur();
    }
    // Ajout
    void ajouter(T& val, int pos) {
        if(reste == NULL || pos == 0){
            Liste* l = new Liste();
            l->val = this->val;
            l->reste = reste;
            reste = l;
            this->val = val;
        } else
            reste->ajouter(val,pos-1);
    }
    // Suppression
    void supprimer(int pos) {
        if(reste != NULL) {
```

```
            if(pos == 0) {
                val = reste->val;
                Liste* l = reste;
                reste = reste->reste;
                l->reste = NULL;
                delete l;
            } else
                reste->supprimer(pos-1);
        }
    }

    // Concatenation
    Liste concat(Liste& l) {
        Liste l_f(*this);
        return l_f.autoConcat(l);
    }

    // Auto-concatenation
    Liste& autoConcat(Liste& l) {
        if(reste == NULL) {
            val = l.val;
            reste = new Liste(*l.reste);
        } else {
            reste->autoConcat(l);
        }
        return *this;
    }
    // Entrees-sorties
    friend ostream& operator<<(ostream &o, Liste &l) {
        // This flag is only here for aesthetic purpose.
        static bool flag = true;
        if (flag == true)
            o << "( ";

        if(l.reste == NULL) {
            flag = true;
            return o << ")";
        } else {
            flag = false;
            o << l.val <<" ";
            return o << *l.reste;
        }
    }

    class ListeIterateur {
        Liste* tete;
        Liste* it;
    public:
        ListeIterateur(Liste& l): tete(&l),it(&l){}
        void reset() {
            it = tete;
        }
        bool hasNext() {
            return it->reste != NULL;
```

```
        }
        T& get(){
            T& val = it->val;
            it = it->reste;
            return val;
        }
    };
};

#endif //__LISTE_RECURSIF_H
```

ListeMaillon

Header file

```
#ifndef __LISTE_MAILLON_H
#define __LISTE_MAILLON_H
/*
 * Creer une classe liste generique (avec template)
 * avec une implentation avec Maillon.
 *
 * Auteur : JC. Creput
 * 20/03/2020
 */
#include <iostream>
#include <locale>
#include <sstream>
#include <string>
#include <cstring>

using namespace std;

template <class T>
struct Maillon {
    T val;
    Maillon* suiv;
    Maillon* prec;//si double chainage

    ~Maillon(){
        delete suiv;
    }
};

// Classe Liste
template <class T>
class Liste {
    Maillon<T>* _tete;
    Maillon<T>* _fin;

public:
    // Forme canonique de Coplien (4 operations)
```

```
    // Cons. par defaut
    Liste() : _tete(NULL), _fin(NULL) {} //liste vide
    // Cons copie
    Liste(const Liste& l){
        Maillon<T>* it_this;
        Maillon<T>* it_l;

        //Création de la tête de liste
        _tete = new Maillon<T>();
        _tete->val = l._tete->val;
        _tete->prec = NULL;

        it_this = _tete;
        it_l = l._tete->suiv;

        while(it_l != l._fin) {
            it_this->suiv = new Maillon<T>();
            it_this->suiv->val = it_l->val;
            it_this->suiv->prec = it_this;
            it_this = it_this->suiv;
            it_l = it_l->suiv;
        }
        //Création de la fin de liste
        it_this->suiv = new Maillon<T>;
        it_this->suiv->val = l._fin->val;
        _fin = it_this->suiv;
        _fin->prec = it_this;
        _fin->suiv = NULL;

    }
    // Destructeur
    ~Liste(){
        delete _tete;
    }
    // Affectation
    Liste& operator=(const Liste& l) {
        if(this != &l) {
            delete this->_tete;
            Liste l_copy(l);
            this->_tete = l_copy._tete;
            this->_fin = l_copy._fin;
            l_copy._tete = NULL;
        }
        return *this;
    }

    Liste(const string& s) : _tete(NULL),_fin(NULL) {
        stringstream s_str(s);
        string token;
        int toAdd;

        while(getline(s_str, token, '-')){
            toAdd = stoi(token);
            ajouter(toAdd, longueur());
```

```
        }
    }

    // Modificateur
    T& operator[](int pos) {
        Maillon<T>* it = _tete;
        int index = 0;
        while(index != pos) {
            it = it->suiv;
            index++;
        }
        return it->val;
    }

    // Longueur
    int longueur() {
        int l = 0;
        Maillon<T>* it_this = _tete;
        while(it_this != NULL) {
            ++l;
            it_this = it_this->suiv;
        }
        return l;
    }

    // Ajout
    void ajouter(T& val, int pos) {
        int i = 0;
        Maillon<T>* it_this = _tete;
        Maillon<T>* nMaillon = new Maillon<T>;
        nMaillon->val = val;
        while(it_this != NULL && it_this->suiv != NULL && i != pos) {
            it_this = it_this->suiv;
            i++;
        }
        if(it_this == NULL) {
            // Case where the list is empty, we must initialize it with one
element
            this->_tete = nMaillon;
            this->_tete->prec = NULL;
            this->_tete->suiv = NULL;
            this->_fin = this->_tete;
        } else if(pos == 0) {
            // Case where we insert the element at the first position.
            nMaillon->suiv = it_this;
            nMaillon->prec = NULL;
            it_this->prec = nMaillon;
            this->_tete = nMaillon;
        } else if(it_this->suiv == NULL) {
            // Case where we reach the end of the list.
            it_this->suiv = nMaillon;
            it_this->suiv->prec = it_this;
            it_this->suiv->suiv = NULL;
            this->_fin = it_this->suiv;
```

```
        } else {
            // Case where we insert the element at the 'pos' position.
            nMaillon->suiv = it_this;
            nMaillon->prec = it_this->prec;
            it_this->prec->suiv = nMaillon;
            it_this->prec = nMaillon;
        }
    }

    // Suppression
    void supprimer(int pos) {
        int i = 0;
        Maillon<T>* it_this = _tete;
        while(it_this != NULL && it_this->suiv != NULL && i != pos) {
            it_this = it_this->suiv;
            i++;
        }
        if(pos == 0) {
            // Case where we insert the element at the first position.
            it_this->suiv->prec = NULL;
            _tete = it_this->suiv;
            it_this->suiv = NULL;
            delete it_this;
        } else if(it_this->suiv == NULL) {
            // Case where we reach the end of the list.
            it_this->prec->suiv = NULL;
            delete it_this;
        } else if(i == pos) {
            // Case where we insert the element at the 'pos' position.
            it_this->prec->suiv = it_this->suiv;
            it_this->suiv->prec = it_this->prec;
            it_this->suiv = NULL;
            it_this->prec = NULL;
            delete it_this;
        }
    }

    // Concatenation
    Liste concat(Liste& l) {
        Liste this_copy(*this);
        return this_copy.autoConcat(l);
    }

    // Auto-concatenation
    Liste& autoConcat(Liste l) {
        Liste l_copy(l);
        _fin->suiv = l_copy._tete;
        l_copy._tete->prec = _fin;
        _fin = l_copy._fin;
        l_copy._tete = NULL;
        return *this;
    }

    // Entrees-sorties
```

```
    friend ostream& operator<<(ostream &o, Liste &l) {
        Maillon<T>* it = l._tete;
        o << "( ";
        while(it != NULL) {
            o << it->val << " ";
            it = it->suiv;
        }
        return o << ")";
    }

    class ListeIterateur {
        Liste* liste;
        Maillon<T>* it;
    public:
        ListeIterateur(Liste& l): liste(&l),it(liste->_tete){}
        void reset() {
            it = liste->_tete;
        }
        bool hasNext() {
            return it != NULL;
        }
        T& get(){
            T& val = it->val;
            it = it->suiv;
            return val;
        }
    };
};

#endif //__LISTE_MAILLON_H
```

Client file

These two implementations use the same client file. They also have the same behaviour.

```
#include <iostream>

// Choisir une implementation (l'une ou l'autre)
//#include "ListeMaillon.h"
#include "ListeMaillon.h"

using namespace std;

int main()
{
    cout << "Test de la classe liste" << endl;

    Liste<int> l1;// liste de integer
    Liste<float> lf;// liste de float

    cout << l1 << endl;
```

```
    cout << lf << endl;
    // Liste<Camion*> lc;// si classe Camion existe

    // Operations sur la liste

    // Ajouter des elements a une position donnee
    int i = 3;
    l1.ajouter(i, 0);
    cout << "liste l1 " << l1 << endl;
    i = 4;
    l1.ajouter(i, 0);
    cout << "liste l1 " << l1 << endl;
    i = 5;
    l1.ajouter(i, 0);
    cout << "liste l1 " << l1 << endl;
    i = 6;
    l1.ajouter(i, 0);

    Liste<int> l4("3-4-5-6");
    cout << "liste l4 " << l4 << endl;
    cout << "liste l1 " << l1 << endl;
    // Suppression d'element
    l1.supprimer(2); // suppression à une position donnee

    // Longeur
    int l = l1.longueur();
    cout << "longeur de l1 " << l << endl;
    cout << "liste l1 " << l1 << endl;

    // Cons par copie
    cout << "l1 : " << l1 << endl;
    Liste<int> l2 = l1;
    cout << "copie de l1 (l2) : " << l2 << endl;
    cout << "address of the head (l1) : " << &l1 << endl;
    cout << "address of the head (l2) : " << &l2 << endl;
    Liste<int> l3;
    // Affectation
    l3 = l2;

    cout << "l2 : " << l2 << endl;
    cout << "copie de l2 (l3): " << l3 << endl;
    cout << "address of the head (l2) : " << &l2 << endl;
    cout << "address of the head (l3) : " << &l3 << endl;

    // modifier un element a une position donnee
    l3[2] = 8;

    // Lire un element a une position donnee
    int elt = l3[2];

    cout << "element " << elt << endl;

    cout << "l1 : " << l1 << endl;
    cout << "l2 : " << l2 << endl;
```

```
    cout << "l3 : " << l3 << endl;

    l1.autoConcat(l2);
    cout << "l1.autoConcat(l2) : " << l1 << endl;


    l3 = l1.concat(l2);
    cout << "l3 = l1.concat(l2) : " << l3 << endl;
    cout << "l1 : " << l1 << endl;

    l3.supprimer(3);
    cout << "l3 suppr pos=3 : " << l3 << endl;

    // Test des itérateurs
    Liste<int>::ListeIterateur iter(l3);
    int r = 0;
    iter.reset();
    cout << "l3 jusqu'à l'élément 3: ";
    while(iter.hasNext() && r < 4) {
        cout << iter.get() << " ";
        r++;
    }
    cout << endl;
    iter.reset();

    cout << "l3 : ";
    while(iter.hasNext()) {
        cout << iter.get() << " ";
    }
    cout << endl;

    cout << "FIN DE TEST" << endl;
    return 0;
}
```