Université Paris Cité **Langages à objets avancés**
M1 **C++**

TP n° 9 : Templates

Durant ce TP nous allons aborder deux exemples d'implémentations qui encapsulent un pointeur.

[**Pointeurs intelligents unique**]

En introduction, et pour justifier d'un besoin possible, notez que lorsqu'on utilise habituellement un pointeur vers un objet, sa destruction n'est pas assurée, comme vous pouvez vous vous en rendre compte sur cet exemple :

```
class A {
        public:
        A() { cout << "construction d'un A" << endl;}
        virtual ~A(){ cout << "destruction d'un A" << endl;      }
        void f() {}
}

int main(){
        A * a = new A;
        return 0;
}  // l'adresse de l'objet A n'a pas été restituée
```

On souhaite utiliser un pattern `Mon_ptr_u<T>` qui se charge de la durée de vie de l'objet de type `T` dont il encapsule l'adresse. Plus particulièrement on assigne à ces objets le rôle de se comporter en tout point comme des pointeurs habituels, mais de se spécialiser au cas où ils sont censés être seuls à s'en occuper.

Ainsi le code réécrit sous la forme suivante s'assurerait d'une destruction complète :

```
int main(){
        Mon_ptr_u<A> p {new A};
        return 0;
}  // ici la destruction de A devra avoir eu lieu
```

et, pour que les choses soient bien faites, on devra encore pouvoir écrire `p->f()` ou `(*p).f()` c.a.d utiliser `p` naturellement comme un pointeur.

Nous allons faire les choses progressivement dans la suite de cet exercice.

1. Déclarez une classe générique `Mon_ptr_u` qui encapsule simplement un pointeur vers un objet de classe `T`. Assurez vous de choisir une visibilité adéquate pour ce champ, et codez le destructeur.
2. Si on cherche à garantir qu'il n'y aura pas d'autres pointeurs intelligents vers le même objet, il ne faut donc pas que nous même dans cette classe nous permettions d'en faire une copie. Il nous faut penser à désactiver le constructeur de copie ainsi que l'opérateur d'affectation par copie.
3. Proposez une méthode `release()` qui libère notre objet de son rôle vis à vis de l'objet encapsulé. Elle retournera le pointeur stocké, le remplacera par **nullptr**.

4. Finalement, l'opérateur d'affectation peut tout de même être utilisé mais dans une sémantique différente (dite du "Move-assignement") : le sens de l'affectation `x = y` sera de transférer à `x` le rôle de représenter le pointeur encapsulé par `y`.
   Bien sûr des questions se posent : quelle est la nouvelle valeur de `y` ? que devient celle de `x` ? Votre code se comporte t'il correctement dans le cas limite `x = x` ?
5. Ecrivez une méthode `void echange(Mon_ptr_u&)` qui permute les éléments pointés.
6. Pour un pointeur `p` classique, on peut par exemple écrire `while(p){...}` ou `if (p){...}`, c'est-à-dire qu'on peut convertir un pointeur vers une valeur de vérité. Redéfinissez l'opérateur `operator bool()const` pour que l'on puisse faire de même avec `Mon_ptr_u`.
7. Redéfinissez les opérateur `*` et `->` pour `Mon_ptr_u` de sorte que `*m_p` et `m_p->xxx` aient le sens attendu. (Il s'agit de redéfinir `operator*()const` et de `operator->()const`)
8. Si l'on veut réellement assurer l'unicité, on ne devrait pas pouvoir construire deux fois un élément à partir du même pointeur. Ajoutez la gestion d'une liste statique des adresses en cours d'encapsulation. Levez une exception si l'unicité n'est pas assurée à la construction.
9. Testez votre travail

```
#include <iostream>
#include "MonPtrU.hpp"
using namespace std;

class A{
        public :
                int v;
                A(int x):v(x){}
};

template <class T> void f(Mon_ptr_u<T> x) {}
template <class T> void g(Mon_ptr_u<T> &x) {}

int main(){
        // Tests 1
        Mon_ptr_u<A> p1 { new A {1} };

        // Tests 2
        // Mon_ptr_u<A> p1bis { p1 }; // Ne doit pas marcher
        // f(p1);       // Ne doit pas marcher
        g(p1);          // Est OK
        Mon_ptr_u<A> p1bis { nullptr };
        // p1bis = p1;      // Ne doit pas marcher

        // Tests 3
        cout << "----------" << endl;
        cout << (p1.release())->v << endl;
        //cout << (p1.release())->v << endl;  // ne marche plus
        cout << p1.release() << endl; // affiche le 0 correspondant à
            nullptr
```

```
    // Tests 4
    cout << "----------" << endl;
    Mon_ptr_u<A> p2 { new A {2} };
    p1 = p2;
    //cout << (p2.release())->v << endl;  // ne doit pas marcher
    cout << p2.release() << endl;        // affiche le 0
        correspondant à nullptr
    cout << (p1.release())->v << endl;  // affiche 2
    cout << p1.release() << endl;        // affiche le 0
        correspondant à nullptr
    Mon_ptr_u<A> p3 { new A {3} };
    p3=p3;
    p3=p3;
    cout << p3.release()->v << endl;     // affiche 3
    cout << p3.release() << endl;        // affiche le 0
        correspondant à nullptr
    p3=p3;
    cout << p3.release() << endl;        // affiche le 0
        correspondant à nullptr
    p3=p3;

    // test 5
    cout << "----------" << endl;
    Mon_ptr_u<A> p4 {new A{4} }, p5 { new A{5} };
    p4.echange(p5);
    cout << p4.release()->v << endl; // affiche 5
    cout << p5.release()->v << endl; // affiche 4

    // Tests 6
    cout << "----------" << endl;
    Mon_ptr_u<A> p6 { new A{6} };
    if (p6) cout << "p6 pointe vers la valeur " << p6.release()->v
        << endl;
    else cout << "p6 pointe vers nullptr" << endl;
    if (p6) cout << "p6 pointe vers la valeur " << p6.release()->v
        << endl;
    else cout << "p6 pointe vers nullptr" << endl;

    // Tests 7
    cout << "----------" << endl;
    Mon_ptr_u<A> p7 { new A{7} };
    cout << "p7 contient " << p7->v << endl;
    cout << "p7 contient " << (*p7).v << endl;

    // Tests 8
    cout << "----------" << endl;
    A* a = new A {7};
    Mon_ptr_u<A> p8 {a};
    //Mon_ptr_u<A> p9 {a}; // Doit lever une exception
    Mon_ptr_u<A> p9 { new A{4} };
    p9 = p8;;
    //Mon_ptr_u<A> p10 {a}; // Doit lever une exception
    p9.release();
    Mon_ptr_u<A> p10 {a};

    return EXIT_SUCCESS;
}
```