Université Paris M1 — **Langages à objets avancés Année 2021-2022**

TP n° 9 : Templates et Patterns

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

Exercice 1 [Pointeurs intelligents unique]

En introduction, et pour justifier d'une 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 0;
}
```

Exercice 2 Cet exercice ressemble beaucoup au précédent, mais les objectifs, et donc les choix d'implémentation diffèrent.

On s'intéresse aux cas de figure où le résultat d'une opération `f()` n'est pas toujours présent. D'ordinaire on se contente d'utiliser un type pointeur pour le

retour de `f()` ce qui lui laisse la possibilité de renvoyer **nullptr** pour signifier l'absence de résultat. Mais on peut objecter parfois que le type déclaré n'exprime pas suffisamment clairement ce risque, et que le programmeur peut s'oublier à écrire `x=f()` puis `x->methode()` sans avoir pris suffisamment de précautions. Pour ces raisons, on peut souhaiter mettre à disposition du programmeur une classe `optional<T>`. Il préférera récupérer un `optional<T>` lors de l'appel de `f();`, et les méthodes suivantes manipulant cet optional lui permettrons d'envisager les cas (selon que le résultat existe ou pas). Voici la description attendue pour un utilisateur de la classe optional [1] :

1. elle ne propose pas de constructeur directs, mais voudrait que vous passiez par des fabriques :
   — `of(T &x)` qui retourne un optional encapsulant `x`
   — `empty()` qui retourne une référence vers un optional vide. (Comme on demande une référence il faut qu'elle ait un nom, pour cela déclarez une constante dans la classe)
   — `ofNullable(T *x)` qui retourne un optional encapsulant `x` s'il est non nul, et empty sinon.
2. `isEmpty()` et `isPresent()` répondent à la question de savoir si l'optional est vide ou non
3. `orElseThrow()` retourne la valeur contenu ou lance une exception s'il n'y en a pas.
4. `orElse(T & other)` retourne la valeur contenue si elle existe, ou l'alternative donnée en argument dans le cas contraire.
5. On souhaite par principe qu'un optional se définisse à l'aide de ses fabriques, c'est à dire que la construction directe soit remplacée par l'appel aux fabriques mentionnées précédemment. Cela nous conduit à réfléchir à ce qu'on peut vouloir souhaiter pour les constructions indirectes (celles par copie et par affectation) Prenez une décision, justifiez là par écrit (en commentaire) et illustrez là.
6. Bien que l'utilisation de pointeurs sur des optional risque de nous ramener à ce qu'on voulait éviter initialement, on décide de ne pas les interdire quand même. Quelles concessions pouvez vous faire (ou pas) à l'utilisation des opérateurs `*, ->, &` ? (Argumentez votre réflexion dans vos commentaires)
7. quand un objet `optional<T>` est contextuellement convertit en booléen, le résultat est vrai ssi il est différent de `empty()`.
8. En c++ vous pouvez passer en argument des fonctions. Vous pouvez donc écrire une méthode `map` qui prend en argument une fonction de `T` vers `U*`,

1. la description choisie ici n'est pas standard : c'est un mélange de ce qui est fait en c++17 et en java. Ainsi vous ne devriez pas trouver de réponses toutes faites qq part sur le web

et qui retourne un `optional<U>` selon le résultat de cette fonction sur le contenu de l'optional. Pour la tester écrivez une classe A et une classe B contenant respectivement un attribut x et un attribut y, et une fonction qui prend un élément de A et retourne une nouvelle adresse vers un B si la valeur contenue dans le A est positive, et **nullptr** sinon.

9. de la même façon, écrivez une méthode `filter` dont l'argument est un prédicat, c'est à dire une fonction qui effectue un test sur le contenu. Le résultat retourné sera l'optional courant ssi il respecte le prédicat, et l'optional vide dans le cas contraire.

Pour vos fichiers, utilisez la trame suivante qui facilitera la correction :

Pour le programme principal :

```
int main(int argc, char** argv) {
        /* description 1
         sequences d'illustration de l'utilisation des fabriques
         et vérification des cas d'impossibilité ou d'erreur
         mettez en commentaire les séquences qui produisent des échecs */

         /* description 2
         tests de isEmpty() et isPresent() */

         /* description 3 et 4
         test des deux méthodes orElse */

         /* description 5
          illustration/explication des choix */

         /* description 6
         illustration/explication des choix */

         /* description 7
         tests de la conversion vers un boolean */

         /* description 8
         testez ici la situation demandée pour map */

         /* description 9
         testez ici la situation demandée pour filter */
        return 0;
}
```