# TP Cooref - Rapport

Dorine Audebert - Vincent Tourenne

# 1. Compréhension du sujet & problématiques soulevées

Ce projet consiste à implémenter deux types de *smartpointer* en C++ avec l'aide de patrons de classe. L'objectif est de gérer efficacement la mémoire dynamique tout en évitant des erreurs de fuite de mémoire. Les contraintes suivantes ont été respectées :

- Utiliser le moins de pointeurs possibles,
- Éviter l'utilisation de *std::shared_ptr* et *std::unique_ptr*,
- Éviter l'utilisation d'un gestionnaire de mémoire,
- Simplifier l'utilisation de nos classes pour l'utilisateur.

# 2. Conception & structure des classes

## a. coorefcounter

**Description**

*coorefcounter* est une classe représentant le compteur de références incluse dans un *coopointer<T>*.

**Attributs**

- *unsigned int cmpt* : Compteur incrémenté ou décrémenté.

Le type unsigned int a été choisi car la valeur minimum du compteur est 0. Un compteur de références négatif ne fait pas de sens dans notre conception.

**Constructeurs & Opérateurs**

- Deux constructeurs : le constructeur par défaut sans argument (initialisant un compteur à 0), et le constructeur usuel prenant un *const int&* en paramètre (initialisant le compteur à celui-ci).
- Les opérateurs préfixes d'incrémentation et de décrémentation (++ et --), modifiant la valeur du compteur.
- L'opérateur d'affectation *(=)* affectant la valeur d'un compteur à un autre.
- L'opérateur de comparaison ==, comparant un compteur à un nombre entier.

## b. coopointer<T>

**Description**

*coopointer<T>* est une classe qui permet de gérer un pointeur vers un objet de type T générique. Son comportement est analogue à un *std::shared_ptr*, permettant l'affectation de plusieurs *coopointer<T>* vers le même espace mémoire.

**Attributs**

- *coorefcounter* cmpt* : Pointeur vers un compteur de références partagé,
- *T* ptr* : Pointeur vers l'objet alloué dynamiquement.

**Constructeurs & Opérateurs**

- Le constructeur par défaut sans argument, initialisant un *coopointer<T>* vers un *nullptr*. On inscrit le compteur à 1 pour éviter des fuites de mémoire dans le destructeur.
- Le constructeur usuel prenant en argument un pointeur sur un objet alloué dynamiquement.
- Le constructeur par copie, partageant le même objet entre les pointeurs tout en augmentant leur compteur commun de référence.
- Le destructeur, libérant la zone mémoire allouée lorsque le compteur de références atteint 0.
- Les opérateurs de déréférencement (*), d'accès par pointeur (->) et d'affectation.

### c. coounique<T>

**Description**

*coounique<T>* est une classe qui permet de gérer un pointeur vers un objet de type T générique. Son comportement est analogue à un *std::unique_ptr*, permettant l'affectation d'un seul *coounique<T>* vers le même espace mémoire.

**Attributs**

- *T* ptr* : Pointeur vers l'objet alloué dynamiquement.

**Constructeurs & Opérateurs**

- Le constructeur par défaut sans argument, défini en privé et ainsi interdit d'utilisation.
- Le constructeur usuel, semblable à celui défini sur *coounique<T>*.
- Le constructeur par copie, transférant la propriété de la mémoire d'un objet *coounique<T>* vers celui construit.
- Le destructeur.

- Les opérateurs de déréférencement (*), d'accès par pointeur (->)
- L'opérateur d'affectation, transférant la propriété de la mémoire d'un objet *coounique<T>* vers un autre.

# 3. Choix techniques

### *coopointer<T>* : pourquoi un attribut coorefcounter* ?

Un attribut *coorefcounter* est associé à un *coopointer<T>* pour compter le nombre de références vers une même zone mémoire. Lorsque plusieurs *coopointer<T>* partagent la même zone mémoire, ils doivent également partager le même compteur. Pour ce faire, on utilise un pointeur.

### *coopointer<T>*, constructeur usuel : comment gérer la mémoire et pourquoi faire des copies en surface ?

Le constructeur usuel de *coopointer<T>* prend toujours en argument un objet déjà alloué dynamiquement. C'est obligatoire pour qu'il soit géré dans le tas, l'automatisation de la gestion de mémoire de la pile pouvant poser problème.

Dans ce constructeur ainsi que tous les autres, on fait une copie en **surface** des pointeurs : l'objectif est que l'on partage la même adresse mémoire, et non les objets pointés.

```
coopointer(T* t) : cmpt(new coorefcounter(1)), ptr(t) {}
```

*Code du constructeur usuel de coopointer<T>*

### *coounique<T>* : pourquoi interdire le constructeur par défaut ?

Le constructeur par défaut sans arguments est interdit afin d'empêcher l'initialisation d'un *coounique<T>* vers une zone mémoire non valide.

# 4. Points techniques intéressants

### Destructeur et libération de mémoire de *coopointer<T>*

```
~coopointer(){
	--(*cmpt);

	if (*cmpt == 0)
	{
		delete cmpt;
		delete ptr;
	}
}
```

*Code du destructeur de coopointer<T>*

Une contrainte de *coopointer<T>* est qu'il faut s'assurer d'empêcher les fuites de mémoire. Ainsi, il est important de libérer une zone mémoire lorsque celle-ci n'est plus utilisée. En d'autre termes, lorsque le compteur *cmpt* sur celle-ci tombe à 0, mais pas avant.

Il faut s'assurer qu'on ne détruise pas trop tôt l'objet si d'autres pointeurs y sont affectés. Ainsi, notre destructeur doit faire deux choses :

- Décrémenter le compteur de références
- S'il est à 0, alors libérer la mémoire du compteur et du pointeur.

**Opérateur d'affectation dans *coopointer<T>***

```
coopointer<T>& operator=(const coopointer<T>& other) {
	if (ptr != other.ptr)
	{
		--(*cmpt);
		// Cas spécial : l'élément qui prend
		// l'affectation était le dernier alloué à sa zone mémoire
		if (*cmpt == 0)
		{
			delete cmpt;
			delete ptr;
		}
		// Sinon, on décrémente simplement son compteur

		// Affectation en surface
		cmpt = other.cmpt;
		ptr = other.ptr;
		++(*cmpt);
	}

	return *this;
}
```

*Code de l'opérateur d'affectation de coopointer<T>*

L'opérateur d'affectation fait une copie en surface des attributs d'un *coopointer<T>* vers un autre.

Il est important de gérer plusieurs cas spéciaux :

- Si les deux *coopointer<T>* sont déjà affectés à la même zone mémoire
  - Alors, rien ne doit se passer.
- Ou si le *coopointer<T>* que l'on affecte était le dernier à pointer à sa zone mémoire
  - Alors il faut la libérer pour éviter des problèmes de fuite de mémoire.

En plus de cela, il faut gérer les compteurs : les décrémenter et augmenter proprement.

**Gestion de la mémoire : utilisation de *valgrind***

Pour vérifier l'absence de fuites mémoire, nous avons utilisé Valgrind, un outil de détection des erreurs liées à la gestion de la mémoire. Le rapport obtenu est le suivant :

```
==8863==
==8863== HEAP SUMMARY:
==8863==     in use at exit: 0 bytes in 0 blocks
==8863==   total heap usage: 9 allocs, 9 frees, 76,828 bytes allocated
==8863==
==8863== All heap blocks were freed -- no leaks are possible
==8863==
==8863== For lists of detected and suppressed errors, rerun with: -s
==8863== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Comme vous pouvez le constater, notre gestion des fuites mémoire est efficace. Cette méthode nous a permis, tout au long de l'élaboration du projet, de détecter et corriger plusieurs erreurs.

# 5. Limitations de la solution

En raison des contraintes inhérentes au sujet, plusieurs limitations doivent être prises en compte pour cette solution. Lors de leur création, un *coopointer<T>* ou un *coounique<T>* ne peut pas vérifier si une zone mémoire a déjà été allouée. Les mécanismes de compteurs et d'unicité n'interviennent qu'au moment des interactions entre objets.
Ainsi, bien que le code suivant soit techniquement valide, il peut entraîner de graves problèmes de gestion de mémoire :

```
int* test = new int(2);

coo::coopointer<int> c1(test);
coo::coopointer<int> c2(test);

coo::coounique<int> cu1(test);
coo::coounique<int> cu2(test);
```

*Code valide mais faux*

Le code est valide mais incorrect. Les compteurs de *c1* et *c2* seront chacun à 1, et de même, *cu1* et *cu2* seront considérés comme valides. Cependant, lorsque c1 sera détruit, le pointeur test sera libéré, bien que *c2*, *cu1*, et *cu2* pointent toujours vers lui, causant ainsi des erreurs potentielles.