Chapitre 12 — Smart Pointers

Les smart pointers (pointeurs intelligents) font partie du **C++ moderne** depuis **C++11**. Ils permettent de gérer la mémoire automatiquement et d'éviter :

- Les memory leaks (fuites mémoire),
- Les double delete,
- Les dangling pointers (pointeurs pendants).

Ils remplacent progressivement les new / delete.

1. Pourquoi les Smart Pointers ?

Avant C++11 :

- Le programmeur devait gérer manuellement la mémoire.
- Erreurs fréquentes :
- int* p = new int(5);
- ...
- delete p; // oublié ou doublé -> crash

Avec les smart pointers :

- ils appellent automatiquement delete
- ils respectent RAII *(Resource Acquisition Is Initialization)*
- ils sont sûrs, efficaces et modernes

2. auto_ptr — Déprécié / Supprimé

auto_ptr existait avant C++11 mais :

- transférait la propriété lors d'une copie
- provoquait des comportements dangereux

Exemple (problème !) :

```
auto_ptr<int> p1(new int(5));
auto_ptr<int> p2 = p1;
```

Après affectation :

- p2 possède la ressource
- p1 devient nul → très difficile à maintenir

Résultat :

- Déprécié en C++11**,**
- Supprimé en C++17**.**

3. unique_ptr — Pointeur exclusif (C++11)

unique_ptr représente une ressource possédée par un seul pointeur**.**

✓ pas copiable

✓ déplaçable (*move semantics*)

✓ très léger et performant

✓ idéal pour RAII

Exemple:

```
#include <memory>
using namespace std;

int main() {
```

```
   unique_ptr<int> p = make_unique<int>(10);
   cout << *p;  // 10
}
```

Transfert de propriété (move) :

```
unique_ptr<int> p1 = make_unique<int>(20);
unique_ptr<int> p2 = move(p1);
```

Après move :

- p2 possède la ressource
- p1 est nul

Utilisation dans une fonction

```
void f(unique_ptr<int> ptr) {
   *ptr = 42;
}

int main() {
   auto p = make_unique<int>(5);
   f(move(p));   // obligatoire
}
```

Idéal pour :

- objets uniques
- ressources systèmes (fichiers, sockets, mutex…)
- structures fortement encapsulées

4. shared_ptr — Comptage de références (C++11)

shared_ptr permet une propriété partagée de la ressource.

✓ peut être copié

✓ détruit la ressource lorsque le compteur atteint 0

✓ pratique pour graphes, callbacks, structures complexes

Exemple :

```
shared_ptr<int> p1 = make_shared<int>(100);
shared_ptr<int> p2 = p1;

cout << p1.use_count(); // 2
```

Fonctionnement :

Pointeur	**Propriété**
shared_ptr	possède une ressource *avec compteur*
use_count()	affiche le nombre de références

Quand tous les shared_ptr meurent → delete automatique.

⚠ Attention : cycles de références

Exemple avec deux objets A et B qui se pointent mutuellement :

A -> B

B -> A

Si chacun utilise un shared_ptr, la mémoire ne sera jamais libérée**.**

Solution : utiliser weak_ptr pour briser les cycles.

5. weak_ptr — Pointeur observateur (C++11)

weak_ptr :

- ne possède pas la ressource
- ne modifie pas le compteur
- observe seulement un shared_ptr
- permet de vérifier si la ressource existe encore

Exemple :

```
shared_ptr<int> sp = make_shared<int>(50);
weak_ptr<int> wp = sp;

if (auto p = wp.lock()) {
   cout << *p;
}
```

🔻 lock() → retourne un shared_ptr temporaire (si ressource valide)
🔻 sinon → retourne nullptr

Quand utiliser quel smart pointer ?

Smart pointer	Propriété	Usage idéal
unique_ptr	exclusif	ressources, RAII, vitesse
shared_ptr	partagé	graphes, callbacks, ownership multiple
weak_ptr	observateur	casser cycles, références faibles
auto_ptr	✗ à éviter	déprécié et supprimé

6. Comparaison rapide

Caractéristique	unique_ptr	shared_ptr	weak_ptr
Possession	exclusive	partagée	aucune
Copiable	✗ non	✓ oui	✓ oui
Déplaçable	✓ oui	✓ oui	✓ oui
Compteur	✗ non	✓ oui	✓ non
Usage	ressources uniques	ressources partagées	éviter cycles

7. Smart Pointers et Orienté Objet

Les smart pointers s'intègrent parfaitement avec les classes :

```
class Compte {
public:
   void afficher() const { cout << "Compte actif\n"; }
};

int main() {
   unique_ptr<Compte> c = make_unique<Compte>();
   c->afficher();
}
```

Ils permettent même de gérer des hiérarchies polymorphiques :

```
unique_ptr<Base> p = make_unique<Derive>();
```

8. Bonnes pratiques C++ moderne

✓ utiliser make_unique() et make_shared()
✓ éviter new et delete

✓ passer des pointeurs par référence si on n'a pas besoin de posséder
✓ éviter les cycles avec shared_ptr