Isima

PARTIE VIII

Pointeurs «intelligents»

Bruno Bachelet

Loïc Yon

Risques des pointeurs

- Nécessaires pour une gestion dynamique de la mémoire
  - Allocation sur le tas
- Mais des risques liés à une gestion manuelle
  - Oubli de libération ⇒ fuite mémoire
  - Pointeur sur une zone libérée ⇒ comportement indéfini
- Même en faisant attention, risque d'erreur
- Exemple (testez avec Valgrind)

```
void f() {
 A * p = new A();

 if (…) throw std::exception(); // Fuite mémoire !

 delete p;
}
```

Pointeurs «intelligents» *(1/2)*

- *Smart pointers*
  - Reposent sur le *design pattern* «proxy»
  - Exploitent la technique RAII du C++

- Proxy = objet qui encapsule un objet
  - Il se fait passer pour l'objet ⇒ interface similaire
  - Il contrôle l'appel à ses méthodes

- RAII = *Resource Acquisition Is Initialization*
  - Acquisition d'une ressource liée à la durée de vie de l'objet
  - Construction objet ⇒ acquisition ressource
  - Tant que l'objet est disponible ⇒ utilisation ressource
  - Destruction de l'objet ⇒ libération ressource
  - Destruction garantie même en cas d'erreur (cf. mécanisme exceptions)

Pointeurs «intelligents» *(2/2)*

- *Smart pointer* = objet qui encapsule un pointeur
- Contrôle les opérations élémentaires
  - Construction, affectation, destruction
- Réduit éventuellement l'interface
  - Pour empêcher l'accès direct à la mémoire pointée (e.g. «`weak_ptr`»)
  - Pour empêcher la copie du pointeur (e.g. «`unique_ptr`»)
- Complète l'interface
  - Opérateurs de mouvement pour le transfert de propriété (e.g. «`unique_ptr`»)
  - Méthodes pour le partage de propriété (e.g. «`shared_ptr`»)
- Réduction des risques d'erreur
  - Destruction du *smart pointer* ⇒ libération de la mémoire liée au pointeur
  - Libération après avoir appelé le destructeur des données pointées

Smart pointers en C++

- Nouvelle proposition depuis C++11

- `unique_ptr` (propriété unique)
  - Garantit un pointeur unique sur une zone mémoire
  - Garantit la libération de la mémoire

- `shared_ptr` (propriété partagée)
  - Permet plusieurs pointeurs sur une même zone mémoire
  - Comptage des pointeurs
  - Garantit la libération de la mémoire quand plus aucun pointeur
    ≈ *garbage collection*

- `weak_ptr` (sans propriété)
  - Permet de s'assurer que le pointeur est toujours valide avant d'accéder à la mémoire associée

Pointeur *unique_ptr (1/3)*

- `unique_ptr` ⇒ un seul pointeur sur une zone mémoire
  - `std::unique_ptr<A> p(new A);`
  - `std::unique_ptr<A[]> p(new A[10]);`
- Destruction du *smart pointer* ⇒ libération de la mémoire
  - Evite toute fuite mémoire
- Déréférencement possible
  - Manipulation classique des opérateurs «`*`», «`->`» et «`[]`»
  - Répercussion sur le pointeur encapsulé

Pointeur *unique_ptr (2/3)*

- Propriété unique ⇒ impossible de le copier
  - Constructeur de copie impossible
  - Affectation par copie impossible
- Possibilité de mouvement ⇒ transfert de propriété
  - Constructeur et affectation par mouvement possibles
- Exemple de transfert de propriété
  - `p1 = std::move(p2);`
  - «`p1`» pointe où «`p2`» pointait
  - «`p2`» pointe sur «`nullptr`»
- Abandon de propriété
  - Méthode «`release`»

Pointeur *unique_ptr (3/3)*

- Exemple

```
void f() {
 std::unique_ptr<A> p1; // Pointeur vide

 {
  std::unique_ptr<A> p2(new A);
  std::unique_ptr<A[]> p3(new A[3]);

  p1 = std::move(p2); // Transfert de propriété
  // Destruction p3 ⇒ libération mémoire (3 objets)
  // Destruction p2 ⇒ rien ne se passe
 }

 // Destruction p1 ⇒ libération mémoire (1 objet)
}
```

Cas d'utilisations d'un *unique_ptr (1/2)*

- Garantir la destruction d'une zone mémoire renvoyée
  - ```
    std::unique_ptr<A> getA() {
     std::unique_ptr<A> p(new A);
     …
     return p; // Pas de copie (car optimisation)
                  ⇒ retour en rvalue
    }
    ```
  - ```
    std::unique_ptr<A> x = getA();
    // Opération de mouvement ⇒ transfert de propriété
    // (Destruction rvalue ⇒ rien ne se passe)
    …
    // Destruction x ⇒ libération mémoire
    ```

Cas d'utilisations d'un *unique_ptr (2/2)*

- Transmettre un pointeur unique en argument
  - ```
    void display(std::unique_ptr<A> x) {
     std::cout << *x << std::endl;
     // Destruction de x ⇒ libération de la mémoire
    }
    ```
  - ```
    {
     std::unique_ptr<A> p(new A);

     display(std::move(p));
     // Mouvement ⇒ transfert de propriété
     …
     // Destruction de p ⇒ rien ne se passe
    }
    ```

Pointeur *shared_ptr (1/2)*

- `shared_ptr` ⇒ plusieurs pointeurs sur une même zone
  - `std::shared_ptr<A> p1(new A);`
  - `std::shared_ptr<A> p2 = p1;`
- Propriété multiple ⇒ copie autorisée
- Les *smart pointers* sont comptés et partagent le compteur
  - Accès au compteur via la méthode «`use_count`»
- Destruction *smart pointer* ⇒ décrémentation compteur
- Changement de pointeur ⇒ décrémentation compteur
  - Via opérateur «`=`» ou méthode «`reset`»
- Compteur = 0 ⇒ libération de la mémoire

Pointeur *shared_ptr (2/2)*

- Exemple

```
void f() {
 std::shared_ptr<A> p1(new A);
 std::shared_ptr<A> p2; // Pointeur vide

 {
  std::shared_ptr<A> p3(new A);

  p2 = p3; // p2 et p3 pointent sur la même zone
  p1.reset(new A); // Destruction de l'objet pointé
                   // et pointage sur le nouvel objet

  std::cout << p3.use_count() << std::endl; // ⇒ 2

  // Destruction de p3 ⇒ compteur = 1 ⇒ rien ne se passe
 }

 // Destruction de p2 ⇒ compteur = 0 ⇒ libération mémoire
 // Destruction de p1 ⇒ compteur = 0 ⇒ libération mémoire
}
```

Pointeur *weak_ptr (1/2)*

- `weak_ptr` = pointeur sur mémoire gérée par «`shared_ptr`»
- N'acquiert pas la propriété
  - Pas d'impact sur le compteur
  - Pas d'impact sur la destruction
- Déréférencement impossible directement
  - Il faut obtenir un «`shared_ptr`»
  - Via la méthode «`lock`»
- Test de validité du pointeur
  - Appel méthode «`expired`»
- Permet un accès fiable à la donnée pointée

Pointeur *weak_ptr (2/2)*

- Exemple d'accès (invalide) sans *smart pointer*

```
A * p1 = new A;
A * p2 = p1;
…
delete p1;
…
std::cout << *p2 << std::endl;
```

- Exemple d'accès (sécurisé) avec *smart pointer*

```
std::shared_ptr<A> p1(new A);
std::weak_ptr<A> p2 = p1;
…
p1.reset(); // Libération mémoire, p1 pointe sur «nullptr»
…
auto p3 = p2.lock();
if (p3) std::cout << *p3 << std::endl;
```

Précautions *(1/2)*

- Allocation mémoire ⇒ pointeur immédiatement dans un *smart pointer*
  - Juste à la sortie du «new»
  - Ne pas utiliser le pointeur brut
  - Ne pas le détruire

- Exemple de problème

```
A * p = new A;
std::shared_ptr<A> p1(p); // Compteur = 1
std::shared_ptr<A> p2(p); // Compteur = 1
// Destruction ⇒ erreur à la seconde exécution
```

- Solution

```
std::shared_ptr<A> p1(new A); // compteur = 1
std::shared_ptr<A> p2 = p1; // compteur = 2
```

- Recommandation: utiliser «make_shared» et «make_unique»

Précautions *(2/2)*

- `make_shared<T>(a1,…,an)` / `make_unique<T>(a1,…,an)`
  - Effectuent l'allocation et retournent un *smart pointer*
  - Se chargent de l'appel au constructeur: `T(a1,…,an)`
- Exemple: `auto p = make_shared<A>();`
- Avantages
  - Aucun accès au pointeur ⇒ évite tout risque de fuite
  - Simplifie l'écriture: évite la répétition du type du *smart pointer* avec «`auto`»
  - Pour «`make_shared`»: plus efficace
    - Allocation groupée: compteurs (pointeurs *shared* et *weak*) + objet(s) pointé(s)
- Inconvénients
  - Fonctionne seulement avec les constructeurs publics de la donnée
  - Problème aussi avec les structures «agrégats» (i.e. construction avec «`{}`»)
  - Pour «`make_shared`»: libération de la mémoire possiblement différée
    - Allocation groupée ⇒ libération impossible si pointeur *weak* actif (besoin compteurs)