Gestion de la mémoire

Loïc Joly, Michel Simatic et Amina Guermouche

Télécom SudParis

2 avril 2025

Plan

1. Durée de vie des objets (RAII)
2. Pointeurs intelligents : unique_ptr
3. Pointeurs intelligents : shared_ptr (et weak_ptr)
4. Destructeurs et polymorphisme

Durée de vie des objets (RAII)

1. → Durée de vie des objets (RAII)
2. Pointeurs intelligents : unique_ptr
3. Pointeurs intelligents : shared_ptr (et weak_ptr)
4. Destructeurs et polymorphisme

Cas d'un objet créé sans `new`

- Lorsqu'un objet est créé dans une fonction, il est automatiquement détruit lorsqu'on quitte cette fonction.

```
class Shape {
  int x, y;
};

void f() {
  Shape s{0, 0};
  ...
  // s est automatiquement detruit lorsqu'on quitte f().
}
```

- Le destructeur des objets est appelé automatiquement à la sortie de la fonction.
- C'est le principe de RAII (*Resource Acquisition Is Initialization*).

Cas d'un objet créé avec `new`

- Lorsqu'un objet est créé en utilisant `new`, sa durée de vie va au delà de la fonction dans laquelle il est défini.

```
class Shape {
  int x, y;
};

void f() {
  Shape *s = new Shape(0, 0);
  ...
  // A la fin de f, s est detruit. Mais l'objet sur lequel pointait s n'est pas detruit a la fin de la fonction.
}
```

- $\Rightarrow$ Souci : Il faut se préoccuper du moment où cet objet doit être effectivement détruit.
  - Avant C++11, seule solution était de faire un appel explicite à `delete`.
  - Avec C++11 (C++ moderne), utilisation de *pointeurs intelligents* (pas de ramasse-miettes/*garbage collector* comme en Java).

Pointeurs intelligents : unique_ptr

1. Durée de vie des objets (RAII)
2. → Pointeurs intelligents : unique_ptr
3. Pointeurs intelligents : shared_ptr (et weak_ptr)
4. Destructeurs et polymorphisme

Principe des `unique_ptr`

- Un `unique_ptr` est un pointeur avec une sémantique particulière : il s'agit du seul possesseur d'un objet donné.
- Afin de construire un `unique_ptr` sur un objet, on utilise `make_unique(Classe)`.
- La zone mémoire pointée par un `unique_ptr` est détruite, quand le `unique_ptr` n'est plus possédé par une variable.

```
#include <memory> // Pour recuperer la declaration de std::unique_ptr et de std::make_unique
class Shape {
  int x, y;
};

void f() {
  unique_ptr<Circle> p = make_unique<Circle>(parametres du constructeur);
  // ou alors
  auto q =  make_unique<Circle>(parametres du constructeur);
}// quand on sort de f, p et q sont detruits. Les objets sur lesquels pointaient p et q sont egalement detruits.
```

Remarques sur les `unique_ptr` (1/3)

- Lorsqu'un `unique_ptr` est passé en paramètre à une fonction, cette fonction prend possession du pointeur et de l'objet pointé.
- Pour transférer la propriété d'un objet, il suffit d'utiliser `std::move`.

```
class Shape {
  int x, y;
};

void f() {
  vector <unique_ptr<Shape>> v;
  unique_ptr<Circle> p{new Circle(...)};
  v.push_back(std::move(p));// Ici, il n'est pas possible de faire push_back(p), car sinon duplication du unique_ptr.
  // a partir d'ici, p vaut null_ptr.
}
```

- Lorsqu'on utilise move, le pointeur initial est positionné à `null_ptr`.

Remarques sur les `unique_ptr` (2/3)

- Cependant, si l'objet est temporaire (pas affecté à une variable), le `move` n'est pas nécessaire.

```
void f() {
  vector <unique_ptr<Shape>> v;
  unique_ptr<Circle> p{new Circle(...)};
  v.push_back(make_unique<Circle>(...)); // Il n'est pas necessaire de faire un move du unique_ptr car c'est une variable temporaire.
  v.push_back(unique_ptr<Circle>(new Circle(...))); // Idem
}
```

- Il est possible de retourner un `unique_ptr` sans faire de `std::move` car le return provoque un transfert et non une copie.

```
unique_ptr<Circle> f() {
  auto p{make_unique<Circle>(...)};
  return p;
}
```

Remarques sur les `unique_ptr` (3/3)

- Un **observateur** de l'objet pointé par un `unique_ptr` peut récupérer le pointeur contenu dans un `unique_ptr` avec la fonction `get`.

```
void f() {
  auto p = make_unique<Circle>(...);
  Circle* observateur_du_circle = p.get();
  // Ne pas faire : delete observateur_du_circle
  // Ne plus utiliser observateur_du_circle si jamais p et donc sa valeur pointee sont detruits !
}
```

Pointeurs intelligents : shared_ptr (et weak_ptr)

1. Durée de vie des objets (RAII)
2. Pointeurs intelligents : unique_ptr
3. → Pointeurs intelligents : shared_ptr (et weak_ptr)
4. Destructeurs et polymorphisme

Principe des `shared_ptr`

- Il est parfois nécessaire que plusieurs pointeurs possèdent une même donnée.
- Pour éviter les problèmes de fuites mémoire, on utilisera des `shared_ptr`.
- Un comptage de référence pour chaque objet est utilisé. Lorsque ce compteur passe à 0, l'objet est détruit.
- Pour construire un `shared_ptr` d'un objet, utilisez `make_shared`.

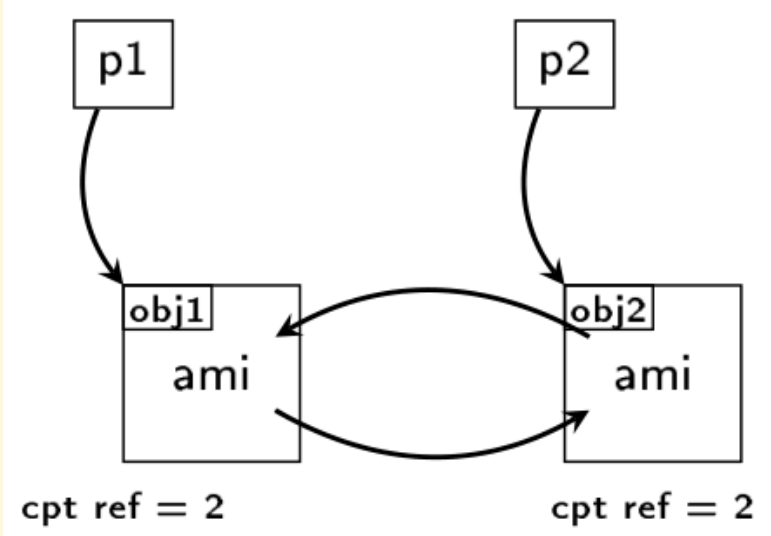
```
#include <memory> // Pour recuperer la declaration de std::shared_ptr et de std::make_shared
void f() {
  auto p = make_shared<Circle>(parametres du constructeur);
  auto q = p; // q pointe sur le même objet que p
  ...
} // A la sortie de f, p et qu sont detruits (RAII). Le Circle vers lequel ils pointaient est detruit aussi.
```

Problème des références circulaires avec les `shared_ptr` (1/3)

- L'objet pointé par des `shared_ptr` est détruit lorsque le compteur de référence passe à 0. En d'autres termes, tant qu'il y a une référence vers un objet, l'objet reste vivant.

```
class Person {
  shared_ptr<Person> ami;
  void set_ami(shared_ptr<Person> t){...}
}

int main()
{
  auto p1 = std::make_shared<Person>(...);
  auto p2 = std::make_shared<Person>(...);
  p1->set_ami(p2);
  p2->set_ami(p1);
}
```

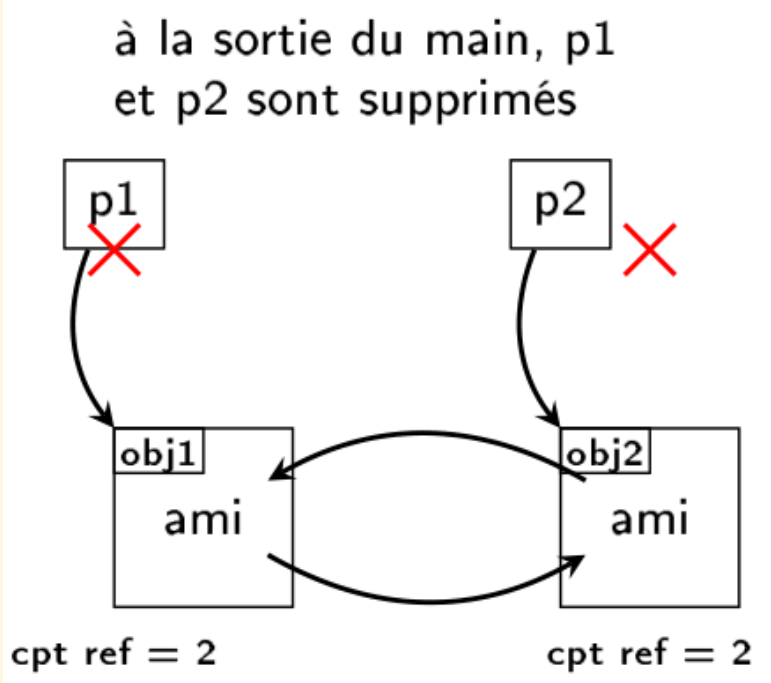


Problème des références circulaires avec les `shared_ptr` (2/3)

- L'objet pointé par des `shared_ptr` est détruit lorsque le compteur de référence passe à 0. En d'autres termes, tant qu'il y a une référence vers un objet, l'objet reste vivant.

```
class Person {
  shared_ptr<Person> ami;
  void set_ami(shared_ptr<Person> t){...}
}

int main()
{
  auto p1 = std::make_shared<Person>(...);
  auto p2 = std::make_shared<Person>(...);
  p1->set_ami(p2);
  p2->set_ami(p1);
}
```



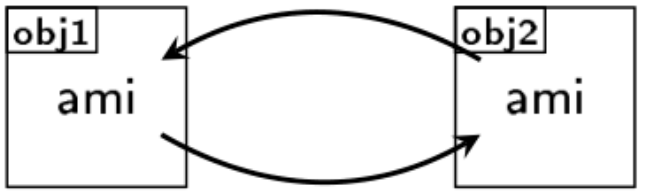
Problème des références circulaires avec les `shared_ptr` (3/3)

- L'objet pointé par des `shared_ptr` est détruit lorsque le compteur de référence passe à 0. En d'autres termes, tant qu'il y a une référence vers un objet, l'objet reste vivant.

```
class Person {
  shared_ptr<Person> ami;
  void set_ami(shared_ptr<Person> t){...}
}

int main()
{
  auto p1 = std::make_shared<Person>(...);
  auto p2 = std::make_shared<Person>(...);
  p1->set_ami(p2);
  p2->set_ami(p1);
}
```

les compteurs sont décrémentés de 1. obj1 fait toujours référence à obj2 et inversement (le compteur de référence vaut 1). Les deux objets ne peuvent donc pas être détruits



Résolution du problème des références circulaires : `weak_ptr`

- Un `weak_ptr` est un *observer* de `shared_ptr` : il observe et accède aux mêmes données qu'un `shared_ptr`, mais ne les possède pas.

```
class Person {
  weak_ptr<Person> ami;
  void set_ami(shared_ptr<Person> t){ami = t;}
}

int main() {
  auto p1 = std::make_shared<Person>(...);
  auto p2 = std::make_shared<Person>(...);
  p1->set_ami(p2);
  p2->set_ami(p1);
}
```

Remarques sur `weak_ptr`

- Un `shared_ptr` peut être assigné directement à un `weak_ptr` (avec l'opérateur `=`).
- Afin d'affecter un `weak_ptr` à un `shared_ptr`, il faut utiliser la fonction `lock`.

```
int main () {
  std::shared_ptr<int> sp1,sp2;
  std::weak_ptr<int> wp;
  sp1 = std::make_shared<int> (10);
  wp = sp1;
  sp2 = wp.lock();
```

Destructeurs et polymorphisme

1. Durée de vie des objets (RAII)
2. Pointeurs intelligents : unique_ptr
3. Pointeurs intelligents : shared_ptr (et weak_ptr)
4. → Destructeurs et polymorphisme

Le destructeur de la classe mère doit être `virtual`

```
void f() {
  vector<unique_ptr<Shape>> v;
  // Ajout de Circle et Group dans v.
}
```

- Dans cet exemple, lorsque v sera détruit, tous les `unique_ptr<Shape>` seront détruits.
- $\Rightarrow$ `delete s` (où `s` est un `Shape*`) sera appelé.
- `s` peut pointer vers un `Circle` ou vers un `Group`. Donc on veut que le bon destructeur soit appelé.
- On ne peut pas appeler le destructeur de la classe fille **sauf** si le destructeur de la classe mère est `virtual` $\Rightarrow$ `virtual ~Shape();`

Questions ?

1. Durée de vie des objets (RAII)
2. Pointeurs intelligents : unique_ptr
3. Pointeurs intelligents : shared_ptr (et weak_ptr)
4. Destructeurs et polymorphisme