

Gestion de la mémoire

Loïc Joly, Michel Simatic, Amina Guermouche

Télécom SudParis

22 Mai 2018

Durée de vie des objets (1/2)

- 1 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 détruit lorsqu'on
       quitte f()*/
}
```

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

Durée de vie des objets (2/2)

- 1 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 détruit mais l'
       objet sur lequel pointe s ne sera pas
       détruit a la fin de la fonction*/
}
```

Pointeurs intelligents : `unique_ptr`

- 1 Un `unique_ptr` est un pointeur avec une sémantique particulière : il s'agit du seul possesseur d'un objet donné.
- 2 Afin de construire un `unique_ptr` d'un objet, on utilisera `make_unique`

```
class Shape{
    int x, y;
};
void f()
{
    unique_ptr<Circle> p = make_unique<Circle>(
        paramtres du constructeur);
    /* ou alors */
    auto q = make_unique<Circle>(paramtres du
        constructeur);
}/* quand on sort de f, p et q seront
détruits. Les objets sur lesquels
pointent p et q vont également être
détruits */
```

Remarques sur les `unique_ptr` (1/3)

- 1 Lorsqu'un `unique_ptr` est passé en paramètre à une fonction, cette fonction prend possession du pointeur et de l'objet pointé
- 2 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) */
    /* a partir d'ici, p vaut nullptr */
}
```

- 3 Lorsqu'on utilise `move`, le pointeur initial est positionné à `nullptr`

Remarques sur les `unique_ptr` (2/3)

- 2 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(unique_ptr<Circle>(new Circle
        (...))); /* il n'est pas nécessaire de
        faire un move du unique_ptr car c'est
        une variable temporaire*/
}
```

Remarques sur les `unique_ptr` (3/3)

- ③ Il est possible de retourner un `unique_ptr` sans faire de `std::move` car le `return` provoquera un transfert et non une copie

```
unique_ptr f()  
{  
    unique_ptr<Shape> p(new Circle(...));  
    return p;  
}
```

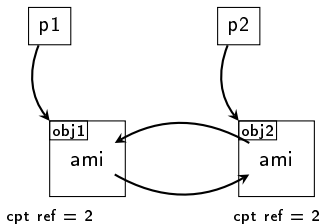
Pointeurs intelligents : `shared_ptr`

- Il est parfois nécessaire que plusieurs pointeurs possèdent une même données.
- 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.
- Afin de construire un `shared_ptr` d'un objet, on utilisera `make_shared`

Problème des références circulaires avec les `shared_ptr`

- 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.

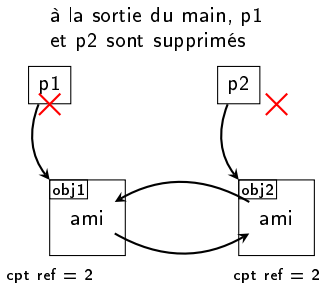
```
class Person
{
    shared_ptr<Person> ami;
    void set_ami(shared_ptr<
        Person> t) {...}
}
int main()
{
    auto p1 = std::make_shared<
        Person >( ... );
    auto p2 = std::make_shared<
        Person >( ... );
    p1->set_ami(p2);
    p2->set_ami(p1);
}
```



Problème des références circulaires avec les `shared_ptr`

- 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);
}
```

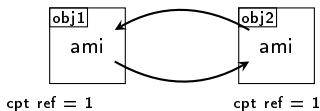


Problème des références circulaires avec les `shared_ptr`

- 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 dé-
crémentés de 1. obj1
fait toujours référence
à obj2 et inversement
(le compteur de réf-
érence vaut 1). Les deux
objets ne peuvent donc
pas être détruits



Problème des références circulaires avec les `shared_ptr`

- Résolution du problème des références circulaires -> utilisation de `weak_ptr`.
- Un `weak_ptr` est un *observer* : il observe et accède aux mêmes données qu'un `shared_ptr` mais ne les possède pas.
- Un `weak_ptr` fonctionne en duo avec un `shared_ptr`.

```
class Person
{
    weak_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);
}
```

Quelques remarques sur les `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

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

- 1 Dans l'exemple, lorsque v sera détruit, tous les unique_ptr<Shape> seront détruits.
- 2 delete s (ou s est un Shape*) sera appelé.
- 3 s peut pointer vers un Circle ou vers un Group. Donc on veut que le bon destructeur soit appelé.
- 4 On ne peut pas appeler le destructeur de la classe fille sauf si le destructeur de la classe mère est virtual : virtual ~Shape()