Programmation orientée objet

Pointeurs intelligents

Motivation

- L'utilisation de pointeurs est une source majeure de problèmes. Par exemple:
  - La perte de tout pointeur vers une ressource allouée dynamiquement entraine une fuite de mémoire
  - L'utilisation d'un pointeur vers une ressource déjà libérée fait « planter » le programme
  - Lorsqu'une fonction retourne un pointeur vers une ressource, qui possède (et doit donc détruire) cette ressource ?

std::unique_ptr - Caractéristiques

- Fournie par la STL (entête **<memory>**)
- Classe générique imitant les pointeurs
- Surcharge les opérateurs * et -> pour offrir les fonctionnalités des pointeurs
- **Détruit automatiquement la mémoire allouée** lorsque l'objet unique_ptr est détruit
- L'objet de type unique_ptr ne peut etre **copié.** La fonction std::move doit etre utilisé pour transférer la ressource à un nouvel objet unique_ptr.

std::unique_ptr - Exemple

- Code original

```
void fonction()
{
     int* ptr = new int;
     ...
     delete ptr;
}
```

- Code avec unique_ptr

```
void fonction()
{
     unique_ptr<int> ptr(new int);
     ...
}
```

Il faut d'abord allouer un pointeur normal et le passer en paramètre au constructeur unique_ptr.

std::unique_ptr - Exemple

- Le code original utilise un pointeur normal, donc la mémoire doit être libérée manuellement (sinon fuite)
- Avec unique_ptr, la mémoire est automatiquement libérée quand l'objet de type unique_ptr est détruit (à la fin de la fonction dans l'exemple)

std::unique_ptr - Exemple

```
class MaClasse
{
    ...
}

void fonction(unique_ptr<MaClasse> pointeur)
{
    ...
}

int main()
{
    unique_ptr<MaClasse> p1(new MaClasse());
    ...
    fonction(std::move(p1));
    ...
    cout << p1->getAttribut();
}
```

À la sortie de la fonction, unique_ptr sera détruit (et la mémoire désallouée).

Aussitôt que p1 aura été envoyé à la fonction, il sera invalidé.

Cette instruction causera donc une erreur à l'exécution.

std::unique_ptr - Exemple

```
unique_ptr<MaClasse> fonction(unique_ptr<MaClasse>
   pointeur)
{
    pointeur->setAttribut(5);
    return std::move(pointeur);
}


int main()
{
    unique_ptr<MaClasse> p1(new MaClasse());
    unique_ptr<MaClasse> p2;
    ...
    p2 = fonction(std::move(p1));
    ...
    cout << p2->getAttribut();
}
```

Lorsque la valeur de retour de la fonction est copiée dans p2, c'est celui-ci qui prend le contrôle de la mémoire allouée.

std::unique_ptr vs pointeur normal

unique_ptr	Pointeur normal
Libération automatique de la ressource quand l'objet unique_ptr est détruit	Libération manuelle de la ressource (généralement opérateur delete)
Quand une exception est lancée, toute ressource pointée par un unique_ptr est détruite	Quand une exception est lancée, il faut s'assurer de tout libérer manuellement, avec un try/ catch(…)
Lorsqu'il est copié, il devient invalide (pointe vers 0)	Lorsqu'il est copié, les deux pointeurs pointent vers le même espace
Ne peut pas pointer vers un tableau	Peut pointer vers un tableau (opérateur [])

std::unique_ptr - Limites

- Même s'ils sont plus avantageux que les pointeurs normaux, les unique_ptr présentent une limitation importante
- Ils ne permettent pas de partager la mémoire (pas copié ou passé en valeur)
- Il n'est donc **pas possible, par exemple, d'utiliser les unique_ptr comme éléments dans un container STL** (car, il serait ensuite impossible de les manipuler par des fonctions ou foncteurs)

shared_ptr

- Un objet de type shared_ptr permet de simuler un pointeur possédant une ressource partagée
- Cet objet peut donc être copié et toute copie va elle-même pointer vers la ressource
- Lorsque le **dernier objet pointant vers une ressource est détruit, il libère la mémoire**

shared_ptr

- Afin de pouvoir fonctionner correctement, la classe shared_ptr implémente un **compteur de référence** (« reference counting »)
- Tous les objets pointant vers la même ressource partagent donc un compteur
- **Chaque objet créé incrémente ce compteur**
- Chaque objet détruit le décrémente
- Lorsque le dernier objet est détruit, donc quand le compteur est 0, cet objet va libérer la ressource

shared_ptr

- En plus de la surcharge des opérateurs, shared_ptr offre deux méthodes intéressantes :
  - use_count(), qui retourne le nombre d'éléments partageant la ressource
  - unique(), qui retourne un booléen indiquant si l'objet est l'unique propriétaire de la ressource (donc si la ressource n'est pas partagée)
- Si on exclut le comportement lors de la copie, un shared_ptr se comporte de manière similaire à un unique_ptr,

shared_ptr

- Code :

```
using namespace std;
int main()
{
  shared_ptr<int> ptr(new int);
  cout <<  "Nb : " << ptr.use_count() << endl;
  shared_ptr<int> ptr2 = ptr;
  cout <<  "Nb : " << ptr.use_count() << endl;
}
```

- Affichage :

```
Nb : 1
Nb : 2
```

shared_ptr - Exemple

```
int main()
{
   Company poly("Polytechnique", "Michele");

   // Add some employees
   shared_ptr<Employee> e1(new Employee("John", 15000));

   poly.addEmployee(e1);
}

void Company::addEmployee(shared_ptr<Employee> employee)
{
   // Insert new employee
   employees_.push_back(employee);
}
```