Programmation par Objets en C++

Clément PERNET

L3-MI, UFR IM^{2}AG, Université Grenoble Alpes

Plan du cours

Compléments de C++11 (10)

Plan

Compléments de C++11 (10)

Les pointeurs intelligents

Plan

Compléments de C++11 (10)

Les pointeurs intelligents

Les pointeurs intelligents

Problème :

- Chaque objet alloué avec un `new` demandait d'être détruit explicitement avec `delete`.
- un oubli de `delete` ne se voit pas, mais consomme de la mémoire (fuite)

Les pointeurs intelligents

Problème :

- Chaque objet alloué avec un **new** demandait d'être détruit explicitement avec **delete**.
- un oubli de **delete** ne se voit pas, mais consomme de la mémoire (fuite)

```
class A{int m,n;};
void f(){
  A a; // allocation sur la pile
  // a sera détruit à la sortie du code de f
}
void g(){
  A* a = new A(); // allocation sur le tas
  // l'objet persiste après la fin de g
  delete a; // à moins qu'on le supprime explicitement
}
```

Les pointeurs intelligents

Problème :

- Chaque objet alloué avec un **new** demandait d'être détruit explicitement avec **delete**.
- un oubli de **delete** ne se voit pas, mais consomme de la mémoire (fuite)

```
class A{int m,n;};
void f(){
  A a; // allocation sur la pile
  // a sera détruit à la sortie du code de f
}
void g(){
  A* a = new A(); // allocation sur le tas
  // l'objet persiste après la fin de g
  delete a; // à moins qu'on le supprime explicitement
}
```

But : simplifier la désallocation sur le tas, pour qu'elle soit automatique (comme celle des objets alloués sur la pile)

Pointeurs intelligents

Trois pointeurs intelligents :

`unique_ptr<T>` : pointage unique

- garantit qu'il est le seul à pointer vers l'objet alloué
- en fin de vie (fin du bloc) il lance la destruction de l'objet pointé

Pointeurs intelligents

Trois pointeurs intelligents :

`unique_ptr<T>` : pointage unique

- garantit qu'il est le seul à pointer vers l'objet alloué
- en fin de vie (fin du bloc) il lance la destruction de l'objet pointé

`shared_ptr<T>` pointage partagé

- permet d'avoir plusieurs pointeurs sur le même objet
- système de comptage de référence (combien de `shared_ptr` pointent sur l'objet)
- A la disparition du dernier, l'objet est désalloué

Pointeurs intelligents

Trois pointeurs intelligents :

`unique_ptr<T>` : pointage unique

- garantit qu'il est le seul à pointer vers l'objet alloué
- en fin de vie (fin du bloc) il lance la destruction de l'objet pointé

`shared_ptr<T>` pointage partagé

- permet d'avoir plusieurs pointeurs sur le même objet
- système de comptage de référence (combien de `shared_ptr` pointent sur l'objet)
- A la disparition du dernier, l'objet est désalloué

`weak_ptr<T>` pointage faible

- n'agit pas sur le décompte des références
- à sa fin de vie, n'implique rien sur celle de l'objet pointé

Pointeurs intelligents : exemples

Pointeurs uniques :

```
#include <memory>
struct A{
  int _m,_n;
  A (int m, int n): _m(m),_n(n){}
};
int main(){
  std::unique_ptr<A> upa (new A(1,2));
  std::unique_ptr<A> upa2  (upa); // erreur de compil
  std::unique_ptr<A> upa3;
  upa3 = upa; // erreur de compil
  A* pa = new A (2,3);
  std::unique_ptr<A> upa4 (pa); // OK
  std::unique_ptr<A> upa5 (pa); // pas d'erreur de compil
  // mais erreur de double désallocation en sortie
}   // ici les objets a et pa sont désalloués automatiquement
```

Pointeurs intelligents : exemples

Pointeurs partagés et faibles :

```
#include <iostream>
struct A{
  int _m,_n;
  A (int m, int n): _m(m),_n(n){}
};

int main(){
  std::shared_ptr<A> spa1 (new A(1,2)); // compteur = 1
  std::shared_ptr<A> spa2  (spa1);      // compteur = 2
  std::shared_ptr<A> spa3;
  spa3 = spa1;                          // compteur = 3
  std::cout<<"Compteur = " << spa3.use_count() << std::endl;
  // affiche "Compteur = 3"

  sp2.reset(); // ne pointe plus vers l'objet compteur =2

  std::weak_ptr<A> wpa = spa1; // compteur = 2 inchangé
}   // ici les objets a et pa sont désalloués automatiquement
```

Pointeurs intelligents : exemples

```
#include <memory>
struct Truc{int n;};
class Machin{
  std::shared_ptr<Truc> ptr;
public:
  Machin(){ptr = std::shared_ptr<Truc>(new Truc());}
  std::shared_ptr<Truc> getTruc(){return ptr;}
};
int main(){
  Machin* m = new Machin(); // alloue un Truc sur le Tas, cpt=1

  std::shared_ptr<Truc> pt  = m->getTruc(); // cpt = 2

  delete m;                                 // cpt = 1

  // ici l'instance de Truc existe encore
  int k = pt.n;

  return 0; // fin de vie de pt -> cpt = 0 -> désallocation de l
      'instance de Truc
}
```