Introduction aux pointeurs intelligents

Patrice Roy

Patrice.Roy@USherbrooke.ca

CeFTI, Université de Sherbrooke

Problème simple

- Objectif : déterminer la responsabilité quant à la durée de vie d'un pointé
  - Plus précisément, encoder la responsabilité à même le type
  - Simplifier la programmation
  - Réduire les risques de fuite de ressources
  - À coût zéro, ou à moindre coût, selon le cas

Exemple

```
class X { /* … */ };
X* f();
void g(X*);
int main()
{
    X *p = f(); // ou auto p = f();
    g(p);
    delete p;
}
```

Exemple

```
class X { /* … */ };
X* f();
void g(X*);
int main()
{
    X *p = f(); // d'où vient p?
    g(p);
    delete p;
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p); // g() détruira-t-elle *p?
    delete p;
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p); // g() lèvera-t-elle une
          // exception? Si oui, qui
          // libérera *p?
    delete p;
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p);
    delete p; // est-ce le rôle
              // de main()?
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p);
    delete p; // est-ce que p a été
              // alloué avec new?
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p);
    delete p; // est-ce que p a été
              // alloué avec new[]?
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p);
    delete p; // est-ce que p a été
              // alloué avec un tout
              // autre mécanisme?
}
```

Exemple

```
class X { /* … */ };
X * f();
void g(X*);
int main()
{
    X *p = f();
    g(p);
    delete p; // est-ce que p a été
              // alloué dynamiquement?
}
```

~~Le problème~~ Les problèmes

- Un pointeur brut a une sémantique indéterminée
- Deux exemples simples:

```
X* f() { return new X; }
X* f()
{
    static X singleton;
    return &singleton;
}
```

- Que devrait faire le code client avec le pointeur retourné?

~~Le problème~~ Les problèmes

- Un pointeur brut n'a pas de responsable clair
- Par exemple:

```
int main()
{
    X *p = f();
    g(p);
    delete p;
}
```

- Qu'adviendra-t-il de `*p` si `g()` lève une exception?
  - Fuite de mémoire (relativement mineur)
  - Non-finalisation de `*p` (peut-être sérieux!)

~~Le problème~~ Les problèmes

- Un pointeur brut n'a pas de responsable clair
- Par exemple:

```
int main()
{
    X *p = new X;
    thread t0{ [p]() { /* ... */ }},
           t1{ [p]() { /* ... */ }};
    t0.detach(); t1.detach();
}
```

- Qui sera responsable de détruire `*p`?

~~Le problème~~ Les problèmes

- Problème de fond

```
int f();
class Y
{
    string *s0;
    int val;
public:
    Y(const string &s)
        : s0{ new string {s}}, val { f() }
    {
    }
    ~Y() { delete s0; }
    // ...
};
```

- Si `f()` lève une exception, qu'adviendra-t-il de `*s0`?

Avec C++03...

- Le standard offrait un seul outil : `auto_ptr<T>` (de `<memory>`)

```
int main()
{
    auto_ptr<X> p(f());
    p->m(); // Ok si X::m() existe
    g(p); // Ok si g(auto_ptr<X>&) existe
}
```

- Un `auto_ptr<T>` offre les services d'un `T*`
  - `operator*()`, `operator->()`
- Le destructeur d'un `auto_ptr<T>` applique `delete` sur son pointé
- Si `g()` lève une exception, la fonction se complétera et `p` sera détruit, amenant `*p` avec lui

Avec C++03...

- Problème de fond... résolu...?

```
int f();
class Y
{
    auto_ptr<string> s0;
    int val;
public:
    Y(const string &s)
        : s0{ new string {s}}, val { f() }
    {
    }
// ...
};
```

- Si `f()` lève une exception, `*s0` sera libéré par le destructeur de `s0`

Avec C++03

- Le type `auto_ptr<T>` est imparfait
  - Ne gère pas les tableaux
  - Ne libère le pointé qu'avec `delete`
  - Sémantique de copie destructrice:

```
int main()
{
    auto_ptr<X> p(f());
    auto_ptr<X> q = p; // transfert de
                       // p vers q
    // ici, p est nul, ce qui
    // peut surprendre
}
```

Avec C++03...

- Pour cette raison, `auto_ptr` ne peut être utilisé dans un conteneur standard de C++03
  - On en viendrait à des conteneurs remplis de pointeurs nuls!

```
vector<auto_ptr<X>> v;
v.push_back(auto_ptr<X>{new X});
v.resize(10);
assert(v.front()); // oups!
```

Depuis C++11

- Le type `auto_ptr` est déprécié
  - Mauvaise solution à un vrai problème
- Deux (en fait, trois) pointeurs intelligents standards sont offerts
  - `unique_ptr`
  - `shared_ptr`
  - (`weak_ptr`, pour accompagner `shared_ptr`)

unique_ptr

```
#include <memory>
int main()
{
    using std::unique_ptr;
    unique_ptr<X> p{f()};
    p->m(); // Ok si X::m() existe
    g(p); // Ok si g(unique_ptr<X>&)
          // existe
}
```

unique_ptr

- Caractéristiques clés
  - Détermine un responsabilité unique sur le pointé
    - Incopiable
    - Déplaçable

```
int main()
{
    unique_ptr<X> p{ new X };
    // unique_ptr<X> q = p; // illégal
    unique_ptr<X> q = std::move(p); // Ok
    // ici, p est nul, mais sans surprises
}
```

  - `vector<unique_ptr<X>>` est Ok
    - Fera des mouvements plutôt que des copies

unique_ptr

- Caractéristiques clés
  - `sizeof(unique_ptr<T>)==sizeof(T*)`
  - Aucun coût en espace!
- Clarifie la sémantique
  - Responsabilité sur le pointé, encodée à même le type
- Réduit les risques
- Aucun coût en espace
- Aucun coût en temps pour `*`, `->`, `==`, `!=`
  - Des nuances, évidemment (p.ex.: il est plus long de copier un `vector<unique_ptr<T>>` qu'un `vector<T*>` car `std::move(unique_ptr<T>)` est $O(3)$

unique_ptr

- Type `unique_ptr` et fonctions, quelques cas types

```
// créer un X dynamiquement, le
// retourner sans risque de fuite
unique_ptr<X> fabrique(args);
// passer par mouvement (car incopiable)
unique_ptr<X> emprunt(unique_ptr<X>);
// X* serait un meilleur choix
void possible_mutation(unique_ptr<X>&);
// const X* serait un meilleur choix
void consultation(const unique_ptr<X>&);
// sink() consommera le paramètre
void sink(unique_ptr<X> &&);
```

unique_ptr

```
int main()
{
    auto p = fabriquer(...);
    p = emprunt(std::move(p));
    possible_mutation(p);
    consultation(p);
    sink(std::move(p));
}
```

unique_ptr

```
void possible_mutation(X*);
void consultation(const X*);
int main()
{
    auto p = fabriquer(...);
    p = emprunt(std::move(p));
    possible_mutation(p.get());
    consultation(p.get());
    sink(std::move(p));
}
```

- Un pointeur brut doit être vu comme une indirection non-propriétaire du pointé (*Non-Owning Pointer*)

unique_ptr

- Gère les tableaux

```
int main()
{
    unique_ptr<int[]> p {
        new int[1000]
    };
    // ...
} // appellera delete [] pour &*p
```

unique_ptr

- Gère des stratégies de destruction alternatives (avec léger coût en espace)

```
class X
{
   ...
private:
   ~X() = default;
   friend struct Meurs;
   void destroy() { delete this; }
};
struct Meurs
{
   void operator()(X *p) { if(p) p->destroy(); }
};
int main()
{
   unique_ptr<X, Meurs> p { new X };
   // ...
} // appellera Meurs::operator() sur &*p, donc p->destroy()
```

unique_ptr

- Gère des stratégies de destruction alternatives (avec léger coût en espace)

```
class X
{
private:
    ~X() = default;
    friend void meurs(X*);
    void destroy() { delete this; }
};
void meurs(X *p) { if (p) p->destroy(); }
int main()
{
    unique_ptr<X, void(X*)> p{new X, meurs};
    // ...
}
```

shared_ptr

```
int main()
{
    X *p{new X};
    thread t0{
        [p]() { /* utiliser p */ }
    };
    thread t1{
        [p]() { /* utiliser p */ }
    };
    t0.detach();
    t1.detach();
}
```

- Sans savoir dans quel ordre se termineront `t0` et `t1`, qui libérera `*p`?

shared_ptr

```
#include <memory>
using std::shared_ptr;
int main()
{
    shared_ptr<X> p{new X};
    thread t0{
        [p]() { /* utiliser p */ }
    };
    thread t1{
        [p]() { /* utiliser p */ }
    };
    t0.detach();
    t1.detach();
}
```

- Le dernier client de `p` appliquera `delete` sur le pointeur qu'il encapsule!

shared_ptr

- Implémente une sémantique de partage

```
shared_ptr<int> p { new int { 3 } };
// q et p partagent leur pointé
shared_ptr<int> q = p;
*q = 4;
cout << *p << endl; // 4
// déconnecter q de p et faire
// pointer q ailleurs
q = shared_ptr<int>{ new int {5} };
```

Considérations d'implémentation

- Implémentation d'un `unique_ptr`
  - Coût en taille
  - Coût en espace
- Implémentation d'un `shared_ptr`
  - Coût en taille
  - Coût en espace
- Esquisse des enjeux
  - Faire un choix éclairé

Esquisse d'un unique_ptr simpliste

```
#include <algorithm>

template <class T>
   class unique_ptr
   {
      T *p;
   public:
      unique_ptr() noexcept
         : p{}
      {
      }
      unique_ptr(T *p) noexcept
         : p{p}
      {
      }
      unique_ptr(const unique_ptr&) = delete;
      unique_ptr& operator=(const unique_ptr&) = delete;
      // ...
```

Esquisse d'un unique_ptr simpliste

```
// ...
unique_ptr(unique_ptr &&autre) noexcept
   : p{ autre.p }
{
   autre.p = {};
}
void swap(unique_ptr &autre) noexcept
{
   using std::swap;
   swap(p, autre.p);
}
unique_ptr& operator=(unique_ptr &&autre) noexcept
{
   swap(autre);
   return *this;
}
~unique_ptr()
   { delete p; }
// ...
```

Esquisse d'un unique_ptr simpliste

```
    // ...
    bool operator==(const unique_ptr &autre) const
    {
       return p == autre.p; // suspect
    }
    bool operator!=(const unique_ptr &autre) const
    {
       return !(*this == autre);
    }
    T& operator*() noexcept { return *p; }
    const T& operator*() const noexcept { return *p; }
    T* operator->() noexcept { return p; }
    const T* operator->() const noexcept { return p; }
};
```

Esquisse d'un shared_ptr simpliste

- Voir http://h-deb.clg.qc.ca/Sujets/AuSecours/ptr_partage.html
  - On parle d'un sujet d'une toute autre complexité!
- Je vous en prie: même si le mien (pédagogique, académique) fonctionne en pratique, et a été utilisé dans de vrais projets, préférez `std::shared_ptr`!

Résumé

- Déterminez la responsabilité sur le pointé à même le type
  - Pointeur brut: non-responsable du pointé (sémantique de référence)
    - Utile lors de passage de paramètre à une fonction
    - Entente tacite: la fonction n'est pas responsable du pointeur
  - `unique_ptr`: seul responsable du pointé (sémantique de responsabilité unique)
    - Léger, rapide
    - Fabriques
    - Évite les fuites
    - Convertible en `shared_ptr` si nécessaire

Résumé

- Déterminez la responsabilité sur le pointé à même le type
  - `shared_ptr`: responsabilité collective sur le pointé (sémantique de partage)
    - Plus lent
    - Plus lourd, plus volumineux
  - Par défaut, exploite mal la *Cache* (deux pointeurs distincts)
    - … mais voir plus bas!
  - Utile en situation de co-responsabilité
    - … mais seulement si l'ordre de destruction des co-responsables est inconnu *a priori*

Fonctions de fabrication

- Fonctions `make_shared` et `make_unique`
  - Rôle de `make_shared`
  - Raison de l'absence de `make_unique` dans C++11
  - Raison de l'ajout de `make_unique` dès C++14!