Programmation orientée objet

Pointeurs intelligents

Définition

- Un pointeur brut = pointeur dont le type est directement T* (référence brut = T&)
- Un pointeur intelligent est une classe qui encapsule la notion de pointeur et règle l'appartenance de la mémoire dynamique.

Motivation

- Pointeur brut (T*) : source majeure de problèmes.
  - Qui doit désallouer la mémoire dynamique?
    - Lors d'un passage en paramètre
    - Lorsque retourné par une fonction
    - Dans un simple new (i.e. items dans Qt)
  - Oublie de désallouer → fuite de mémoire.
  - Utilisation après la désallocation → « undefined behavior ».
  - Comment bien gérer les exceptions?
  - Quand désallouer la mémoire d'une possession de mémoire partagée?

Bonnes pratiques en C++

- Pour ces raisons: (C++ core guidelines)
  - Jamais transférer la possession de la mémoire à l'aide d'un pointeur ou référence brut (I.11)
  - Un pointeur ou référence brut ne devrait jamais posséder une ressource mémoire (R.3, R.4)

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Pointeurs intelligents (C++11)

- But:
  - Désallocation automatique de la mémoire quand aucun pointeur intelligent n'y pointe.
    - Incluant si son propriétaire détruit par une exception
  - Utilisation comme un pointeur
- Propriétaire unique: unique_ptr
  - L'unique pointeur intelligent décide de la durée de vie de l'espace mémoire
- Plusieurs propriétaires: shared_ptr
  - Plusieurs pointeurs intelligents possèdent la mémoire dynamique
- #include <memory>

unique_ptr et shared_ptr

- (« … » pour « unique » ou « shared »)
- Classes génériques
  - …_ptr<T> un élément de type T
  - …_ptr<T[]> tableau d'éléments
- Surcharge pour utilisation comme un pointeur
  - * -> s'il pointe vers un élément
  - [] s'il pointe vers un tableau
  - = dont = nullptr
  - == != < <= > >= comparaison de pointeurs
- Généralement préférer make_... à new pour allouer la mémoire dynamiquement
  - make_...<T>(arguments, au, constructeur)
  - make_...<T[]>(taille_du_tableau)
  - (make_unique est C++14; make_shared<T[]> est C++20)

unique_ptr

- Ne peut pas être copié
  - Erreur de compilation lors de copie
  - Un passage par valeur sans « move » est une copie donc une erreur de compilation
- Peut transférer la possession de la mémoire dynamique
  - Utiliser std::move

```
vector<unique_ptr<Item>> items;
auto unItem = make_unique<Item>();
items.push_back(move(unItem));
```

Pourquoi?

Allocation dynamique.

Transfert possession au vector.

unItem devient nullptr.

unique_ptr – Exemple

- Avant

```
class MaClasse {
public:  void posseder(Item* item) {
            delete item_;
            item_ = item;
         }
         ~MaClasse() { delete item_; }
private: Item* item_ = nullptr;
};
```

Pas évident que l'appelant donne sa possession de la mémoire.

Doit libérer manuellement, sinon fuite

Doit initialiser sinon « undefined behavior » lors du premier delete.

- Avec unique_ptr

```
class MaClasse {
public:  void posseder(unique_ptr<Item> item)
         { item_ = move(item); }
private: unique_ptr<Item> item_;
};
```

Appelant doit donner sa possession

Désallocation automatique de l'ancien Item_, et item_ conserve l'espace mémoire du paramètre item

Désallocation automatique à la destruction

std::unique_ptr – Exemple (suite)

- Avant

```
MaClasse a;
Item* item = new Item(1, "sans1");
cout << item->nom << endl;
a.posseder(item);
a.posseder(new Item(2, "sans2"));
```

Manque-t-il un delete après?

- Avec unique_ptr

```
MaClasse a;
auto item = make_unique<Item>(1, "avec1");
cout << item->nom << endl;
a.posseder(move(item));
a.posseder(make_unique<Item>(2, "avec2"));
```

Comme un pointeur

Transfert explicite; item est nullptr après

move non nécessaire car objet temporaire

shared_ptr

- Possession de la mémoire est partagée entre différents pointeurs intelligents
  - Compteur de pointeurs intelligents qui possèdent la mémoire dynamique
    - .use_count() pour obtenir ce compte
  - Chaque copie compte +1, chaque destruction -1
  - **Mémoire dynamique libérée lorsque le dernier pointeur intelligent est détruit** (compte à 0)
- Peut transférer la possession (optimisation)
  - À un autre shared_ptr (**pas à un unique_ptr**)

```
vector<shared_ptr<Item>> items;
auto unItem = make_shared<Item>();
items.push_back(unItem);
items.push_back(move(unItem));
```

Partage la possession (+1)

Transfert la possession → **rien à compter**; **unItem devient nullptr**

shared_ptr

- Code :

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

- Affichage :

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

Bonnes pratiques en C++ (suite)

- On devrait: (C++ core guidelines)
  - Utiliser unique_ptr et shared_ptr pour représenter la possession mémoire. (R.20)
  - Préférer unique_ptr à shared_ptr sauf si besoin de partager la possession. (R.21)
  - Prendre en paramètre à une fonction un pointeur intelligent si et seulement si pour changer la durée de vie de l'espace mémoire. (R.30)
    - Utiliser une simple référence (T&) ou pointeur (T*) si la fonction n'a pas à influencer la durée de vie.
    - Utiliser T* si peut être nullptr; sinon préférer T&.

Exemple shared_ptr

```
class Chanson {
public:
    Chanson(string chanteur): chanteur_(chanteur)
    { cout << chanteur_<< endl; }
    ~Chanson () { cout << "detruit "
                       << chanteur_ << endl; }
    string getChanteur() { return chanteur_; }
private:
    string chanteur_;
};
```

Exemple shared_ptr

```
int main()
{    vector<shared_ptr<Chanson>> v {
       make_shared<Chanson>("Bob Dylan"),
       make_shared<Chanson>("Aretha Franklin"),
       make_shared<Chanson>("Celine Dion")
     };

   vector<shared_ptr<Chanson>> v2;
   remove_copy_if(v.begin(), v.end(), back_inserter(v2),
                  [](const shared_ptr<Chanson>& s)
                  { return s->getChanteur() == "Bob Dylan"; });
```

Les chansons sont partagées entre les v et v2

Exemple shared_ptr

```
cout << "V" << endl;
for (const auto& s : v)
    cout << s->getChanteur() << endl;

cout << "V2" << endl;
for (const auto& s : v2)
    cout << s->getChanteur() << endl;
```

Bonnes pratiques – Exemple

```
void parametreParReference(Item& item)
{
    item.nom += "A";
}
void parametreParPointeur(Item* item)
{
    if (item != nullptr)
        item->nom += "B";
}
int main()
{
    vector<unique_ptr<Item>> items;
    //...
    parametreParReference(*items[1]);
    parametreParPointeur(items[2].get());
}
```

unique_ptr ou shared_ptr ne change rien au reste du code de cet exemple

.get() pour obtenir le pointeur

Pointeurs intelligents et itération sur conteneurs STL

- Doit utiliser une référence au pointeur

```
vector<unique_ptr<Item>> items;
```

Référence au pointeur intelligent; pas de copie

```
for (const auto& item : items)
   cout << item->nom << "; ";
```

Se comporte comme un pointeur (`->` et non `.`)

Pointeurs intelligents, conteneurs STL et lambdas

- Paramètre est une référence au pointeur
  - (exception à la règle)

```
sort(items.begin(), items.end(),
   [](const unique_ptr<Item>& a,
      const unique_ptr<Item>& b)
   { return a->nom < b->nom; }
);
sort(items.begin(), items.end(),
   [](const auto& a, const auto& b)
   { return a->nom < b->nom; }
);
```

Référence au pointeur; pas de copie

Peut utiliser « auto » en C++14

Résumé

- Un pointeur intelligent est une classe générique.
- La mémoire dynamique allouée est détruite quand le pointeur intelligent qui possède cette mémoire n'y pointe plus (détruit ou réaffecté).
- Cette mémoire dynamique peut ou ne pas être partagée.
- move() pour le transfert de possession de mémoire à un autre pointeur intelligent.