Programmation orientée objet

Pointeurs intelligents

Motivation

- Un pointeur brut est un pointeur dont le type est directement T*, où T peut être un int, un string, un Employee, ...
- Les pointeurs bruts sont une source majeure de problèmes.
- Les problèmes se trouvent majoritairement au niveau de:
  - la possession de mémoire
    - Qui doit désallouer la mémoire dynamique?
    - Quand désallouer la mémoire d'un pointeur partagé?
  - la désallocation des pointeurs
    - Risque de fuite de mémoire
    - Risque d'erreur d'exécution(« undefined behavior »)

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

Qu'est-ce qu'un pointeur intelligent?

- Un pointeur intelligent est une classe de la librairie standard qui encapsule la notion de pointeur:
  - S'occupe d'allouer et désallouer la mémoire dynamique
  - Les opérateurs de déréférencement (*, ->), d'accès ([]) et de comparaison sont accessibles par un pointeur intelligent de la même manière qu'un pointeur brut
- Pour pouvoir utiliser les pointeurs intelligents, on doit inclure leur librairie:

```
#include <memory>
```

Classes de pointeurs intelligents

- Propriétaire unique: unique_ptr
  - Un objet de type unique_ptr est le seul à pointer vers l'espace mémoire qu'il a créé
  - Un objet de type unique_ptr décide de la durée de vie de l'espace mémoire vers lequel il pointe
- Plusieurs propriétaires: shared_ptr
  - Un objet de type shared_ptr peut partagée la mémoire qu'il a crée avec d'autres objets du même type
  - Lorsqu'il n'y a plus d'objets de type shared_ptr qui pointe vers l'espace mémoire, l'espace est automatiquement désallouée

cppreference – shared_ptr et unique_ptr

Création de pointeurs intelligents

```
int main() {
   unique_ptr<Employee> emp_ptr_unique =
                        make_unique<Employee>();
   shared_ptr<Employee> emp_ptr_shared =
                         make_shared<Employee>(“Bob”);
   unique_ptr<Employee[]> emp_tableau =
                       make_unique<Employee[]>(1);
}
```

À la sortie de la fonction, les variables locales sont détruites. L’espace mémoire allouée dynamiquement par les pointeurs intelligents est désallouée par ceux-ci lors de leur destruction.

Manipulation des pointeurs intelligents

```
int main() {
    unique_ptr<Employee> emp_ptr_unique =
            make_unique<Employee>("Bob");
    emp_ptr_unique->setName("Bobette");
    cout << emp_ptr_unique->getName() << endl; //"Bobette"
    Employee emp;
    *emp_ptr_unique = emp;
    cout << emp_ptr_unique->getName() << endl; //"unknown"
    unique_ptr<Employee[]> emp_tableau =
            make_unique<Employee[]>(1);
    cout << emp_tableau[0].getName() << endl; //"unknown"
}
```

Accès au pointeur brut

Important

```
int main() {
    unique_ptr<Employee> emp_ptr_unique =
            make_unique<Employee>("Bob");
    Employee* emp_ptr = emp_ptr_unique.get();
}
```

unique_ptr

- Une erreur de compilation se produit lorsqu'on tente de copier un pointeur intelligent unique.

Seulement 1 ptr peut pointer vers l'espace

Pourquoi?

- Par contre, on peut transférer la possession de la mémoire dynamique en utilisant la fonction std::move

```
int main() {
    unique_ptr<Employee> emp_ptr_1 =
            make_unique<Employee>("Bob");
    unique_ptr<Employee> emp_ptr_2 = move(emp_ptr_1);
}
```

Transfert de possession de emp_ptr_1 à emp_ptr_2.

emp_ptr_1 devient nullptr.

Exemple de possession de mémoire

- Avec les pointeurs bruts

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

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

Doit libérer manuellement, sinon fuite

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

Manque-t-il un delete après?

```
int main() {
    MaClasse a;
    Item* item = new Item;
    a.posseder(item);
    a.posseder(new Item);
}
```

Exemple de possession de mémoire

- Avec pointeurs intelligents uniques

```
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

Transfert explicite; item est nullptr après

move non nécessaire car objet temporaire

```
int main() {
    MaClasse a;
    unique_ptr<Item> item = make_unique<Item>();
    a.posseder(move(item));
    a.posseder(make_unique<Item>());
}
```

shared_ptr

- Possession de la mémoire est partagée entre différents pointeurs intelligents
  - Compteur de pointeurs intelligents qui possèdent la mémoire dynamique qui est obtenu à l'aide de la fonction use_count()
  - **Mémoire dynamique libérée lorsque le dernier pointeur intelligent est détruit** (compte à 0)
- Un transfert de possession de mémoire est possible en utilisant la fonction std::move() entre deux shared_ptr

```
int main() {
    shared_ptr<Item> item_1 = make_shared<Item>();
    shared_ptr<Item> item_2 = item_1;
    shared_ptr<Item> item_3 = move(item_1);
}
```

Partage la possession (+1)

Transfert la possession → **rien à compter**; **item_1 devient nullptr**

shared_ptr – Partage de la possesion

```
int main() {
    shared_ptr<Item> item_1 = make_shared<Item>();
    cout << item_1.use_count() << endl; //1
    shared_ptr<Item> item_2 = item_1;
    cout << item_2.use_count() << endl; //2
    shared_ptr<Item> item_3 = move(item_1);
    cout << item_3.use_count() << endl; //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&.

Bonnes pratiques – Exemple

```
void parametreParReference(Item& item) {
    item.ajouterCaractere("A");
}

void parametreParPointeur(Item* item) {
    if (item != nullptr)
        item->ajouterCaractere("B");
}

int main() {
    unique_ptr<Item> item = make_unique<Item>();
    parametreParReference(*item);
    parametreParPointeur(item.get());
}
```

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

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.