Chapitre 5

Pointeurs intelligents

Allocation de ressources

Pointeur intelligent unique

Pointeur intelligent partagé

Divers

Allocation de ressources

Valeurs, Références, Pointeurs

En C++, les objets peuvent être manipulés par valeur, référence, pointeur.

```
personne p1(...);
personne & p2(p1);
personne * p3(nullptr);
personne * p4(&p1);
personne * p5(new personne(...));
```

Une valeur et une référence identifient obligatoirement un objet.

Un pointeur peut être **`nullptr`** : on ne doit pas appeler de méthode à partir de ce pointeur.

Syntaxiquement `p4` et `p5` sont de même type mais `p5` doit être libéré (**`delete`** `p5;`) et surtout pas `p4`.

Valeurs, Références, Pointeurs

Quand on peut éviter d'utiliser des pointeurs, on les évite :

- Cause de fuites de mémoire (si on oublie `delete p5;`)
- Cause de plantages (`delete p4;`)

Même chose pour le passage de paramètres :

```
bool estplusjeuneque(personne const * p2) const // ou
bool estplusjeuneque(personne const & p2) const
```

La deuxième solution est préférable, car un pointeur peut être `nullptr`.

Même chose pour le type des attributs d'une classe.

```
class personne{
...
  private:
    std::string * _nom;
    unsigned int * _age;
};
```

Faire ceci est une mauvaise idée : il faut allouer dynamiquement `std::string` et `unsigned int` dans le constructeur, et les libérer dans le destructeur.

Pointeurs

Quels sont les avantages des pointeurs, alors ?

- Ils peuvent être **`nullptr`** : Donnée optionnelle
- Ils permettent de repérer des objets de différentes classes (polymorphisme)
- Ils sont facilement copiables (partage d'objets)

Le partage d'objets est aussi une difficulté : qui doit libérer l'objet pointé par le pointeur si plusieurs classes/fonctions ont accès au même objet par pointeur ? quand le faire ?

Pointeurs

Dès qu'on utilise des pointeurs, se pose la question de la propriété de l'objet pointé.

Bonne pratique (RAII). L'objet pointé par un pointeur attribut est créé dans le constructeur de la classe propriétaire, détruit dans le destructeur de la même classe. Une classe robuste ne devrait jamais donner accès à ce pointeur.

Problème. Comment respecter la *règle des 0* quand un attribut d'une classe est un pointeur.

Pointeurs - Information optionnelle

```
class fichier {
...
  private:
  fichiernom _nom;
  fichiertaille _taille;
  infofichier * _infosuppl;
};
```

`_infosuppl` est une information optionnelle : certaines instances de `fichier` auront une valeur **`nullptr`** , d'autres une valeur non nulle (pointant sur un objet alloué dynamiquement dans le constructeur, libéré dans le destructeur).

`fichier` respecte la bonne pratique RAII mais requiert d'écrire un destructeur. Comment faire pour s'en passer ?

Pointeurs - Conteneur et polymorphisme

Dans la classe `stock`, les objets doivent être manipulés par pointeur.

- Pointeurs sur des objets créés dynamiquement dans `stock`
- Objets libérés dans `stock`
- Les pointeurs sur ces objets ne doivent être visibles nulle par ailleurs.

$\Rightarrow$ Un constructeur virtuel a été défini, ce qui est mémorisé dans `stock` lors d'un ajout est une copie de `produit` dont le `stock` est propriétaire.

`stock` respecte la bonne pratique RAII mais requiert l'écriture d'un destructeur. Comment faire pour s'en passer ?

Pointeur intelligent unique

Pointeurs

`stock` est robuste car respecte le principe RAII et la règle des 3 mais...

Requiert de gérer la mémoire, avec un constructeur par recopie particulier, un destructeur particulier, un opérateur d'affectation particulier, si on en oublie un, code incorrect.

Un `produit` d'un `stock` a comme (unique) propriétaire le `stock` et n'est accessible qu'à ce `stock`.

$\rightarrow$ Pour un `produit` il existe un unique pointeur pointant sur cette instance, ce pointeur étant stocké dans une instance de `stock`.

unique_ptr

Un pointeur intelligent unique est un pointeur qui représente une relation de propriété unique d'un objet pointé.

- Unique point d'accès à l'objet propriété du pointeur.
- Peut être vide (n'est propriétaire d'aucun objet).
- Sa valeur ne peut être copiée (cela créerait un deuxième accès à l'objet pointé). Vérifié à la compilation.
- Libère automatiquement l'objet propriété quand le pointeur cesse d'exister.
- Coût en mémoire ou en temps d'exécution : 0.

unique_ptr

- `#include <memory>`
- Type : `std::unique_ptr<type>`
- Constructeur par défaut : Pas d'objet propriété
- Allocation dynamique d'un objet propriété par `std::make_unique<type>(parametrescontructeur)`
- Conversion en **`bool`** : **`true`** si possède un objet, **`false`** sinon
- Le constructeur par recopie et l'opérateur d'affectation sont `=`**`delete`**
- `std::move` permet de transférer la propriété de l'objet pointé
- `reset` permet libérer l'objet propriété

unique_ptr - déclaration

stock : déclaration de l'attribut _prod (ressources/stock.hh)

```
std::vector<std::unique_ptr<produit>> _prod;
```

stock : destructeur (ressources/stock.hh)

```
~stock() =default;
```

`std::unique_ptr` s'utilise comme un pointeur : utilisation de `->` pour accéder à un membre, opérateur `*` pour accéder à l'objet pointé.

unique_ptr - accès

stock : parcours - Erreur (ressources/stock.cc)

```
void stock::afficher() const {
  for (auto p : _prod)
    p->afficher();
}
```

Erreur de compilation : `p` est résolu comme un `std::unique_ptr<produit>`, qui a chaque itération reçoit une copie d'un `std::unique_ptr<produit>` de `_prod`

```
error: use of deleted function
```

stock : parcours - Correct (ressources/stock.cc)

```
void stock::afficher() const {
  for (auto const & p : _prod)
    p->afficher();
}
```

unique_ptr - dynamic_cast

Un **dynamic_cast** ne peut être fait directement sur un `std::unique_ptr`, mais uniquement sur un pointeur (brut).

La méthode `get` de `std::unique_ptr` permet d'accéder au pointeur (brut) sur l'objet.
À utiliser avec précaution. Normalement, uniquement pour faire une conversion telle que **dynamic_cast**.

stock : dynamic_cast (ressources/stock.cc)

```
void stock::afficherperemptions() const {
  for (auto const & p : _prod) {
    auto pp = dynamic_cast<produitperissable const*>(p.get());
    if (pp != nullptr)
      std::cout << pp->peremption();
  }
}
```

`unique_ptr` - passage en paramètre

stock : ajout - Erreur (ressources/stock.cc)

```
void stock::ajouterproduit(std::unique_ptr<produit> p) {
  _prod.push_back(p);
}
```

Erreur de compilation : ajoute à `_prod` une copie de `p`. Il faut transférer la propriété de l'objet au `std::unique_ptr` qui sera dans `_prod` en utilisant `std::move`.

stock : ajout (ressources/stock.cc)

```
void stock::ajouterproduit(std::unique_ptr<produit> p) {
  _prod.push_back(std::move(p));
}
```

Après un appel à `std::move`, le `std::unique_ptr p` ne pointe plus sur un objet, sa conversion en `bool` est `false`.

`unique_ptr` - passage en paramètre

stock : ajout (ressources/stockmain.cc)

```
stock s;
auto pp = std::make_unique<produitperissable>("pp1", 100, "10/10")
    ;
/* Erreur, copie impossible.
s.ajouterproduit(pp);
*/
s.ajouterproduit(std::move(pp));
```

Il n'a pas été nécessaire de définir un constructeur virtuel pour écrire une classe robuste.

La classe `stock` est plus simple que précédemment : pas de gestion d'allocations dynamiques (aucun **`new`**, aucun **`delete`**), pas de destructeur.

`unique_ptr` - constructeur par recopie

Toute classe dispose d'un constructeur par recopie implicite qui appelle les constructeurs par recopie de ses attributs.

$\rightarrow$ Le constructeur par recopie de `stock` appelle le constructeur par recopie de `std::vector<std::unique_ptr<produit>>`
qui appelle le constructeur par recopie de
`std::unique_ptr<produit>`
qui ne peut être appelé.

`unique_ptr` - constructeur par recopie

Mais alors... pourquoi pas d'erreur de compilation ?
Parce que le constructeur par recopie implicite n'est « généré » que s'il est appelé. Il suffit de l'appeler pour obtenir une erreur...un peu incomprehensible car elle est dans un code qu'on n'a pas écrit.

Il est donc conseillé d'interdire explicitement la copie (constructeur par recopie, opérateur d'affectation) de `stock` (idem **`operator=`**).

stock : interdire la copie (ressources/stock.hh)

```
stock(stock const & s) =delete;
```

Ainsi l'appel au constructeur par recopie provoquera l'erreur `error: use of deleted function 'stock::stock(const stock&)'`.

`unique_ptr` - constructeur par recopie

Mais si on veut copier des `stock` ? Rien n'empêche de définir le constructeur par recopie (et l'opérateur d'affectation) en utilisant le constructeur virtuel de `produit`.

produit : constructeur virtuel

(ressources/produit.hh)

```
virtual std::unique_ptr<produit> clone() const =0;
```

produit périssable : constructeur virtuel

(ressources/produitperissable.hh)

```
std::unique_ptr<produit> clone() const override {
return std::make_unique<produitperissable>(*this);
}
```

stock : constructeur par recopie

(ressources/stock.hh)

```
stock(stock const & s) {
for (auto const & p : s._prod)
  _prod.push_back(p->clone());
}
```

Pointeur intelligent partagé

`unique_ptr` et propriété partagée

Dans certains cas, `std::unique_ptr` n'est pas adapté : Quand la propriété d'un objet doit être partagée dans plusieurs instances.

Exemple. On veut rajouter à `produit`, `stock`... la possibilité de représenter des commandes composées de produits, et on veut stocker à l'intérieur d'une commande des pointeurs vers les produits du `stock` (et non des copies des produits).

Premier problème. Donner accès aux produits d'une instance de `stock`.

Propriété partagée

recherche d'un produit dans `stock` - Erreur (ressources/partage/stock.hh)

```
produit rechercher1(reference r) const {
  for (auto & p : _prod)
    if (p->ref() == r)
      return *p;
}
```

Erreur de compilation : la méthode retourne une instance de `produit` (construite par le constructeur par recopie de `produit`), or `produit` est abstraite.

Mais même si elle n'était pas abstraite, ce serait faux : Problème de *slicing*.

Propriété partagée

recherche d'un produit dans `stock` - Fragile (ressources/partage/stock.hh)

```
produit const * rechercher2(reference r) const {
  for (auto & p : _prod)
    if (p->ref() == r)
      return p.get();
  return nullptr;
}
```

Fonctionne, mais dangereux : `stock` expose un pointeur sur un objet dont il devrait être le seul propriétaire.

Démonstration de la fragilité (ressources/partage/stockmain.cc)

```
auto prod = s.rechercher2(ref);
std::cout << *prod << "\n";
delete prod;
```

Erreur de segmentation dans le destructeur de `stock`.

recherche d'un produit dans `stock` - Moyen (ressources/partage/stock.hh)

```
produit const & rechercher3(reference r) const {
  for (auto & p : _prod)
    if (p->ref() == r)
      return *p;
}
```

Fonctionne, mais génère un avertissement : la méthode peut se terminer sans rien retourner.
(Peut être géré par une levée d'exception).

Propriété partagée

Attention à l'utilisation :

Retour de référence - Incorrect (ressources/partage/stockmain.cc)

```
produit prod1 = s.rechercher3(ref);
auto prod2 = s.rechercher3(ref);
```

Ces deux lignes provoquent une erreur de compilation : Ce sont des appels au constructeur par recopie de `produit` (initialisation d'un `produit` à partir d'une référence de `produit`) : instanciation d'une classe abstraite.

Retour de référence - Correct (ressources/partage/stockmain.cc)

```
auto const & prod = s.rechercher3(ref);
std::cout << prod << "\n";
```

Propriété partagée

La classe commande - fragile

(ressources/partage/commandev1.hh)

```
class commande {
  public:
    commande(std::string const & client)
      :_client(client), _produits()
    {}
    void ajouter(produit const * p) {
      _produits.push_back(p);
    }
  private:
    std::string _client;
    std::vector<produit const *> _produits;
};
```

Utilisation

(ressources/partage/stockmain.cc)

```
  auto const & prod = s.rechercher3(ref);
  std::cout << prod << "\n";
  commande c1("nom");
  c1.ajouter(&prod);
```

Propriété partagée

Ça marche, mais...

- Les produits sont manipulés par référence et par pointeurs (bruts).
- Des pointeurs bruts sur les produits de `stock` sont utilisés ailleurs que dans `stock`.
- Ne pas libérer les pointeurs de `_produits`.
- Aucune garantie dans `commande` que les pointeurs mémorisés pointent sur des objets qui existent encore → Fragilité aux évolutions de `stock`.

shared_ptr

Un pointeur intelligent partagé est un pointeur qui représente une relation de propriété d'un objet pointé. La propriété peut être partagée à partir de plusieurs pointeurs.

- Peut être vide (n'est propriétaire d'aucun objet).
- Sa valeur peut être copiée : Cela partage la propriété avec un nouveau pointeur partagé.
- Libère automatiquement l'objet propriété quand le « dernier » pointeur partagé est détruit ou est modifié pour pointer vers un autre objet.
- Coût en mémoire (compteur de référence) et en temps d'exécution (incrémentation / décrémentation du compteur).

shared_ptr - Déclaration

Stock (ressources/partage/stockv2.hh)

```
class stock {
  public:
  stock( ) =default;
  stock(stock const & s) =delete;
  ~stock( ) =default;
  stock & operator=(stock const & s) =delete;
  void afficher( ) const;
  float tvamoyenne( ) const;
  void afficherperemptions( ) const;
  void ajouterproduit(std::shared_ptr<produit> p);
  std::shared_ptr<const produit> rechercher(reference r) const;
  private:
  std::vector<std::shared_ptr<produit>> _prod;
```

shared_ptr - Utlisation

Stock (ressources/partage/stockv2.cc)

```
void stock::ajouterproduit(std::shared_ptr<produit> p) {
  _prod.push_back(std::move(p));
}
std::shared_ptr<const produit> stock::rechercher(reference r) const{
  for (auto const & p : _prod)
    if (p->ref() == r)
      return p;
  return nullptr;
}
```

- L'appel à `std::move` dans `ajouterproduit` est optionnel.
- `rechercher` permet de partager la propriété d'un produit.
- `nullptr` peut être utilisé pour construire un `std::shared_ptr`.
- Préférer le passage par valeur des `std::shared_ptr`, sauf si on sait ce qu'on fait.

shared_ptr - Utilisation

Commande

(ressources/partage/commandev2.hh)

```
class commande {
  public:
    commande(std::string const & client)
      :_client(client), _produits()
    {}
    void ajouter(std::shared_ptr<const produit> p) {
      _produits.push_back(std::move(p));
    }
  private:
    std::string _client;
    std::vector<std::shared_ptr<const produit>> _produits;
};
```

Garantie. Les pointeurs mémorisés dans `commande` pointent sur des objets qui ne peuvent être détruits ailleurs (même si le `stock` est modifié / supprimé).

shared_ptr - Exemple

main (ressources/partage/stockmainv2.cc)

```
int main() {
  stock s;
  auto pp = std::make_shared<produitperissable>("pp1", 100, "10/10")
      ;
  auto ref = pp->ref();
  s.ajouterproduit(std::move(pp));

  auto prod = s.rechercher(ref);
  commande c1("nom");
  c1.ajouter(std::move(prod));
  return 0;
}
```

`shared_ptr` - Opérateurs de conversion

La méthode `get` retourne le pointeur brut d'un `std::shared_ptr` mais doit être évitée.
Il est inutile de l'appeler pour faire appel à un opérateur de conversion : `std::static_pointer_cast`, `std::const_pointer_cast` et `std::dynamic_pointer_cast`.

`std::dynamic_pointer_cast` (ressources/partage/stockv2.cc)

```
void stock::afficherperemptions() const {
  for (auto const & p : _prod) {
    auto pp = std::dynamic_pointer_cast<produitperissable>(p);
    if (pp)
      std::cout << pp->peremption();
  }
}
```

shared_ptr

- Utiliser systématiquement `std::make_shared<>`.
- Un constructeur de `std::shared_ptr` prend comme paramètre un `std::unique_ptr`.
- `std::shared_ptr<>::use_count()` permet d'accéder au compteur de références.
- `std::shared_ptr` libère l'objet pointé quand le compteur de références atteint 0. Totalement déterministe.
- Attention aux références cycliques. Résolues par `std::weak_ptr`.

Pointeurs intelligents - Résumé

- Éviter autant que possible les pointeurs bruts.
  Causes de fuites de mémoire, Cause d'erreurs d'exécution si mal utilisés, Complique le code : Requiert destructeur, constructeur par recopie, opérateur d'affectation (règle des 3).
- Préférer les pointeurs intelligents.
  Permettent souvent d'appliquer la règle des 0.
- Préférer les `std::unique_ptr` aux `std::shared_ptr`.
  Gestion « automatique » de la mémoire, sans aucun coût (mémoire, cpu).
- Réserver les `std::shared_ptr` à la propriété partagée de ressources entre différentes instances pour lesquelles on ne peut isoler un propriétaire qui aura une durée de vie supérieure aux autres et qui contiendra toutes les ressources.

Divers

Déclaration avancée

Habituellement, une classe est déclarée dans un fichier `.hh` et définie dans un fichier `.cc` de même nom.

- Pour hériter d'une classe C, pour déclarer un attribut de type C, pour utiliser C comme paramètre (passé par valeur) d'une déclaration de méthode ou valeur de retour (par valeur) d'une déclaration de méthode, pour appeler une méthode de C...
  Le compilateur a besoin de la déclaration de la classe ⇒ Inclure le fichier `.hh`
- Pour déclarer un pointeur (ou une référence) sur un C...
  Le compilateur a besoin de savoir que C est une classe.
  ⇒ Il n'est pas nécessaire d'inclure le fichier `.hh`
  Une déclaration avancée suffit.

Déclaration avancée

Précédemment, nous avions fait un `#include "produit.hh"` dans `stock.hh` pour déclarer l'attribut `std::vector<produit *> _prod`.
(ou un `std::vector<std::unique_ptr<produit>>`)

En utilisant une déclaration avancée, on peut écrire :

Déclaration de `stock` (ressources/stockv2.hh)

```
#pragma once
#include <vector>
#include <iostream>
#include <memory>
class produit;

class stock {
```

Déclaration avancée

Par contre, pour **appeler** des méthodes de `produit`, le compilateur a besoin de connaître la déclaration complète de la classe.
Ici `stockv2.hh` ne contient pas de code de méthodes appelant des méthodes de `produit` donc une déclaration avancée est possible.
Les méthodes de `produit` sont appelées dans `stockv2.cc`, il faut donc une déclaration complète de la classe :

Définition de `stock` (ressources/stockv2.cc)

```
#include "stockv2.hh"
#include "produit.hh"
#include "produitperissable.hh"
#include <iostream>
void stock::afficher() const {
  for (auto const & p : _prod)
    p->afficher();
}
```

`friend`

Problème. Rendre visibles à une fonction `f` ou aux méthodes d'une classe `D` les membres privés d'une classe `C`... sans les montrer aux autres fonctions/classes de l'application.

Exemple. Montrer les attributs de `fichier` dans son opérateur de sortie.

Syntaxe `friend`

La déclaration dans une classe `D` de :

- `friend nomclasse;`
  rend les membres privés de `D` visibles par toutes les méthodes de `nomclasse`.
- `friend typeretour nomfonction(arguments);`
  rend les membres privés de `D` visibles par la fonction `nomfonction`.

friend

Opérateur de sortie de stock (ressources/stockv2.hh)

```
  void ajouterproduit(std::unique_ptr<produit> p);
  private:
  std::vector<std::unique_ptr<produit>> _prod;
  friend std::ostream & operator<<(std::ostream & os, stock const &
      s);
};
```

Opérateur de sortie de stock (ressources/stockv2.cc)

```
std::ostream & operator<<(std::ostream & os, stock const & s) {
  for (auto const & p : s._prod)
    p->sortie(os);
  return os;
}
```