Travaux Pratiques
Programmation Multi-Paradigme
Licence 3 Informatique

Julien BERNARD et Arthur HUGEAT

Table des matières

Consignes communes à tous les projets

Au cours de cette UE, vous avez **trois** projets à réaliser à raison d'un projet pour deux séances de trois heures de travaux pratiques encadrées. Les projets sont à faire et à rendre dans l'ordre du présent sujet.

Pour chaque projet, vous devrez implémenter une interface donnée dans un fichier d'en-tête, ainsi qu'un ensemble de tests unitaires pour cette interface. Les tests serviront à montrer que votre implémentation est correcte et complète.

Il est attendu que vos codes sources et les commentaires soient rédigés en anglais et uniquement en anglais.

Projet n°1 : *Smart Pointers*

En C++, lorsqu'on alloue dynamiquement un objet, l'utilisateur doit le libérer quand il n'est plus utile. Il ne faut pas le libérer trop tôt afin d'éviter des accès invalides (souvent traduit par des *segfaults*). Il ne faut pas non plus trop attendre sous peine de saturer la mémoire ; dans le pire des cas, l'objet n'est jamais libéré ce qui entraîne des fuites mémoires.

Avec l'arrivée du C++ moderne, on a mis au point des pointeurs intelligents *(Smart Pointers)*. Ce concept se base le notion de propriété *(ownership)* d'un objet dynamique. Un pointeur intelligent possède un objet dynamique lorsqu'il est responsable de l'accès et de la libération de l'objet.

Le but de ce premier projet est d'implémenter trois types de pointeur intelligent :

1. `Unique`
2. `Shared`
3. `Weak`

Étape 1 : *Unique pointer*

Le premier type de pointeur intelligent que nous allons implémenter est le pointeur `Unique`. Lorsqu'un pointeur `Unique` possède un objet dynamique, aucun autre `Unique` pointeur ne peut le posséder. Cela signifie qu'à tout instant du programme, un objet dynamique ne peut être possédé que par un seul pointeur `Unique`.

Par conséquent, on ne peut pas copier un pointeur `Unique` car cela signifierait que deux pointeurs `Unique` possèdent le même objet. Toutefois, il est possible de déplacer un pointeur `Unique` vers un autre. Dans ce cas, la propriété de l'objet est transférée entre les pointeurs `Unique`. Lorsqu'un pointeur `Unique` est détruit, l'objet qu'il possédait est libéré.

Pour plus de détails sur le fonctionnement d'un pointeur `Unique`, vous pouvez aller vois la classe équivalente de la bibliothèque standard : `std::unique_ptr`.

Étape 2 : *Shared pointer*

Le deuxième type de pointeur que nous allons voir est le pointeur `Shared`. Plusieurs pointeurs `Shared` peuvent posséder un même objet dynamique. Chacun des ces pointeurs peut accéder et modifier l'objet dynamique.

Il est donc tout à fait possible de copier un pointeur `Shared` créant ainsi un pointeur `Shared` gérant le même objet dynamique. On peut également déplacer un pointeur `Shared` et de ce cas là, le pointeur source transfert la possession de l'objet au nouveau pointeur. L'objet dynamique n'est libéré que lorsque le dernier pointeur `Shared` le possédant est détruit. Tant qu'il reste au moins un pointeur `Shared` qui possède l'objet, il est gardé en mémoire.

Pour cela, il est nécessaire d'avoir un compteur du nombre de pointeurs `Shared` qui pointent vers le même objet. Le compteur accompagne le pointeur alloué. Quand ce compteur tombe à zéro, on peut libérer l'objet.

Pour plus de détails sur le fonctionnement d'un pointeur `Shared`, vous pouvez aller voir la classe équivalente de la bibliothèque standard : `std::shared_ptr`.

Étape 3 : *Weak pointer*

Le dernier type de pointeur que nous allons ajouter sont les pointeurs `Weak`. Ce type de pointeur fonctionne de pair avec les pointeurs `Shared`. En effet, un pointeur `Weak` ne possède pas réellement l'objet alloué, il doit s'assurer que l'objet existe encore avant de pouvoir y accéder. S'il existe encore, il générera un nouveau pointeur `Shared` possédant l'objet dynamique.

L'intérêt des pointeurs `Weak` vient du fait qu'un cycle de pointeurs `Shared` ne peut pas être libéré. L'introduction d'un pointeur `Weak` permet de casser le cycle et donc de permettre la libération de tous les pointeurs.

On peut copier et déplacer les pointeurs `Weak` comme on l'a fait pour les pointeurs `Shared`. En revanche, l'objet dynamique est libéré lorsque le dernier pointeur `Shared` qui le possède est détruit. Il est donc tout a fait possible d'avoir des pointeurs `Weak` qui font référence à un objet libéré. C'est pourquoi il est impératif de vérifier l'existence de l'objet avant d'y accéder. À l'inverse, il est possible de détruire tous les pointers `Weak` qui font référence à un objet dynamique sans que celui-ci ne soit libéré.

En pratique, il faut, en plus du compteur d'objets précédemment décrit, ajouter un compteur de pointeurs `Weak` qui servira à savoir quand libérer les compteurs associés à un objet.

Pour plus de détails sur le fonctionnement d'un pointeur `Weak`, vous pouvez aller vois la classe équivalente de la bibliothèque standard : `std::weak_ptr`.

Exemple d'utilisation

```cpp
#include <iostream>

#include "Shared.h"
#include "Unique.h"
#include "Weak.h"

int main() {
  auto unique = sp::makeUnique<int>(0);

  if (unique) {
    ++(*unique);
  }

  std::cout << *unique << std::endl; // 1

  unique.reset();
  bool exists = unique; // false

  auto shared = sp::makeShared<int>(42);
  if (shared.exists()) {
    std::cout << *shared << std::endl; // 42
  }

  sp::Weak<int> weak1(shared);
  {
```

```
    auto tmp = weak1.lock();
    bool b = tmp.exists(); // true
    (*tmp) /= 2;
    std::cout << *tmp << std::endl; // 21
  }

  shared.reset();
  exists = shared.exists(); // false

  shared = sp::makeShared<int>(1337);
  sp::Weak<int> weak2(shared);
  {
    auto tmp = weak1.lock();
    bool b = tmp.exists(); // false

    tmp = weak2.lock();
    if (tmp) {
      std::cout << *tmp << std::endl; // 1337
    }
  }

  return 0;
}
```

Projet n°2 : Vocabulary Type

Le but de ce projet est d'implémenter deux classes : `Any` et `Optional`. Toutes deux font parties de ce qu'on appelle les *vocabulary types*. `Any` est une classe qui peut contenir une donnée de n'importe quel type. Par exemple, elle peut être utilisée pour parser des fichiers JSON où on ne connaît pas le type des données à l'avance.

`Optional` est une classe qui peut contenir ou non une valeur du type donnée. Elle permet, par exemple, de pouvoir contrôler le retour d'une fonction sans devoir passer des variables supplémentaires, on récupère seulement un objet `Optional`.

Étape 1 : Any

La classe `Any` n'est pas une classe avec un template donc pour pouvoir contenir n'importe quel type de donnée, on va recourir à une suppression du type *(type erasure)* : on fait disparaître le type sous-jacent derrière une interface commune à tous les types.

Pour ce faire, on va créer une classe de base abstraite (virtuelle pure) qui servira à interagir avec l'objet en question (copier, déplacer...). Puis on écrit une classe templatée, qui héritera de la classe abstraite. Ainsi, dans la classe `Any`, on aura une variable membre du type le classe virtuelle et le polymorphisme appellera les bonnes méthodes pour interagir avec l'objet contenu.

Toutefois, il faudra permettre à l'utilisateur d'accéder à la variable stockée. Dans ce cas là, on ne pourra pas faire appelle à un `static_cast` ou autre puisque il n'existe aucun opérateur de conversion direct. Vous devrez donc implémenter une fonction `anyCast()` qui sera chargée de retourner une copie de l'objet contenu ou un pointeur vers l'objet contenu. Si jamais l'utilisateur essaie de transtyper l'objet `Any` vers un type différent de la valeur qu'il contient, la fonction `anyCast` lancera une exception du type `std::bad_cast`.

Un objet `Any` initialisé par le constructeur par défaut ne contiendra aucune valeur. Le méthode `clear()` permet de revenir à cet état par défaut. On pourra construire un objet `Any` directement avec une valeur. La variante de construction avec un paramètre de type `InPlaceTypeStruct` permet d'éviter de faire une copie et de construire l'objet en appellant directement le constructeur de la valeur.

Vous devrez également implémenter la fonction utilitaire `makeAny()`. Elle permet de retourner un objet `Any` contenant déjà un objet initialisé avec les arguments de la fonction.

Les types d'objet pouvant être contenu dans un `Any` doit être constructible par copie.

Étape 2 : Optional

La classe `Optional` sera quant à elle templatée. Le type indiquera le type de la valeur contenue s'il y en a une. Par défaut, la classe `Optional` ne contient aucune donnée. On peut lui transmettre une valeur à la construction ou par affectation ultérieurement. La classe dispose également d'une construction sur place, qui permet d'appeler directement le constructeur de la valeur.

Tout comme pour `Any`, on définit une fonction utilitaire `makeOptional()` qui permet elle aussi de créer un objet `Optional` directement avec une valeur.

L'accès à la variable passe par des fonctions membres de la classe `Optional` :

— `getValue()` : cette fonction retourne la valeur et lance une exception du type `std::runtime_error` si aucune variable n'est initialisée

— `getValueOr()` : cette fonction retourne la valeur contenu dans l'objet ou la valeur qui est passée en paramètre de la méthode

L'appel à la méthode `clear()` permet de réinitialiser l'objet et il sera alors considéré comme non initialisé.

Pour des raisons pratiques, on fournit les opérateur `*` et `->` pour nous permettre d'accéder directement aux attributs ou aux champs de la valeur. Toutefois, si la valeur n'a pas été initialisée, le comportement de ces opérateurs est indéfinis (en d'autres mots, vous n'avez pas à gérer ce cas).

Toujours pour des raisons pratiques, on peut comparer deux objets `Optional` entre eux mais on peut également comparer un objet `Optional` à une valeur.

Pour finir, un objet `Optional` doit pouvoir gérer un objet non copiable. Ceci dit, il s'agit d'une consigne relativement complexe donc il est préférable de la gérer quand tout le reste sera fait.

Exemple d'utilisation

```
#include <cmath>
#include <iostream>
#include <utility>

#include "Any.h"
#include "Optional.h"

namespace {
  voc::Optional<double> my_sqrt(long value) {
    if (value < 0) {
      return voc::Optional<double>();
    }

    return voc::Optional<double>(std::sqrt(value));
  }

  struct Point {
    Point(int x1, int y1) {
      x = x1;
      y = y1;
    }

    int x;
    int y;

    bool operator==(const Point& other) const {
      return x == other.x && y == other.y;
    }
  };

  voc::Optional<Point> couldCreatePoint(bool create) {
```

```
    if (create) {
      return voc::Optional<Point>(voc::InPlace, 42, 24);
    }

    return voc::Optional<Point>();
  }
}

int main() {
  voc::Any any;

  any = 42;
  std::cout << voc::anyCast<int>(any) << std::endl;
  // 42

  any = 3.14;
  std::cout << voc::anyCast<double>(any) << std::endl;
  // 3.14

  any = std::string("The cake is a lie!");
  std::cout << voc::anyCast<std::string>(any) << std::endl;
  // The cake is a lie!

  try {
    std::cout << voc::anyCast<bool>(any) << std::endl;
  } catch (const std::exception& e) {
    std::cout << e.what() << std::endl; // throw std::bad_cast, wrong type
  }

  voc::Any any_inplace(voc::InPlaceType<Point>, 42, 24);
  // Call the Point(int, int) constructor directly
  auto p = voc::anyCast<Point>(any_inplace);
  // {42, 24}

  auto any_cleared = voc::makeAny<Point>(42, 42);
  any_cleared.clear();
  std::cout << any_cleared.hasValue() << std::endl;
  // 0 (no data)

  try {
    auto point = voc::anyCast<Point>(any_cleared);
    std::cout << point.x << "x" << point.y << std::endl;
  } catch (const std::exception& e) {
    std::cout << e.what() << std::endl; // throw std::bad_cast, no initialized
  }

  voc::Optional<double> opt = my_sqrt(-1.0);
  std::cout << opt.getValueOr(-1.0) << std::endl;
  // -1.0

  opt = my_sqrt(9.0);
  std::cout << opt.getValueOr(-1.0) << std::endl;
```

```
  // 3.0

  voc::Optional<Point> optPoint = couldCreatePoint(true);
  p = optPoint.getValue();
  // {42, 24}

  auto opt2 = voc::makeOptional<Point>(42, 24);
  std::cout << (p == opt2.getValue()) << std::endl;
  // 1
  std::cout << (optPoint == opt2) << std::endl;
  // 1

  opt2.clear();
  std::cout << opt2.hasValue() << std::endl;
  // 0 (no data)

  return 0;
}
```