Programmation C++ avancée

26 novembre 2019

1 Nombres entiers et réels

1.1 Héritage et interface

La classe `number` ci-dessous définit une interface minimale pour des types de nombres. Supposons que `T` soit une classe qui hérite de `number`. La méthode `T::build` est censée créer un pointeur unique vers un objet de type `T` qui stockera la valeur numérique passée en argument. La méthode `T::display` affichera sur la sortie standard le type `T` et la valeur stockée.

```
class number {
 public:
  virtual std::unique_ptr<number> build(int) = 0;
  virtual void display() = 0;
  virtual ~number() = default;
};
```

Question : la méthode `number::build` aurait pu renvoyer un simple `number*` ; quel est l'intérêt de renvoyer un `unique_ptr<number>` ? (Dans la suite, ce type sera dénoté `number_ptr`.)

Définissez une classe `integer` qui hérite de `number` et implémente ses méthodes virtuelles. Un objet de cette classe stockera une valeur numérique à l'aide d'un champ de type `int`.

De la même façon, définissez une classe `real` qui utilisera un champ de type `double` pour stocker une valeur numérique.

Testez votre code avec une fonction du genre suivant :

```
int main() {
  number_ptr p = integer().build(1);
  real().build(2)->display();
  p->display();
}
```

Un affichage possible serait le suivant :

```
(real) 2
(integer) 1
```

1.2 Usine à objet

Une *factory* est un objet capable de créer sur demande des objets d'autres types. Il s'agit ici de créer à la demande des nombres ayant les types `integer` et `real` définis ci-dessus. La classe `number_factory` aura donc le squelette suivant :

```
class number_factory {
 public:
  number_ptr build(const std::string& t, int x);
};
```

La méthode `number_factory::build` prend en paramètre une chaîne de caractère et une valeur numérique. Si la chaîne vaut `"integer"`, un objet de type `integer` est créé à partir de la valeur numérique et est renvoyé sous forme de pointeur unique. Si la chaîne vaut `"real"`, le comportement est analogue.

Il serait possible de coder en dur tous les cas dans `number_factory::build` sous forme de *if-then-else*. L'objectif est ici d'avoir une approche plus extensible, c'est-à-dire que le code de la méthode ne connaît

pas *a priori* l'ensemble des types disponibles. On pourrait par exemple imaginer que de nouveaux types sont ajoutés dynamiquement.

Implémentez la méthode `number_factory::build` à l'aide d'un dictionnaire. Pour ce faire, on pourra utiliser `std::map<std::string,number_ptr>` ; un dictionnaire de ce type associe à une clé de type `string` une valeur de type `number_ptr`. La classe générique `map` s'utilise d'une façon similaire à un tableau ou à un vecteur. Par exemple, `dict[key] = value` crée une association entre la clé `key` et la valeur `value` dans le dictionnaire `dict`.

Ajoutez un constructeur à la classe `number_factory`. Il sera chargé d'initialiser le dictionnaire en créant des associations entre les chaînes `"integer"` et `"real"` et les pointeurs `number_ptr` respectifs.

Testez votre classe avec le code suivant :

```
int main() {
  number_factory fact;
  number_ptr q = fact.build("integer", 5);
  q->display();
  q = fact.build("real", 42);
  q->display();
}
```

Question : est-il nécessaire d'ajouter un destructeur à la classe `number_factory` pour libérer ce que le constructeur a alloué (les entrées du dictionnaire par exemple) ?

Question : pourquoi est-ce que le programme plante si l'on exécute `fact.build("toto", 17)` ?

Corrigez la méthode `number_factory::build` pour qu'elle renvoit un pointeur nulle si jamais le nom de type passé en argument est inconnu. On pourra utiliser la méthode `map::count` pour vérifier si le dictionnaire contient une entrée donnée.

Question : Comment modifier le code appelant pour tenir compte des erreurs ?

2 Tableaux à taille fixe

L'objectif de cet exercice est de définir une classe proposant des fonctionnalités proches de `std::array<T,N>` (la version C++ d'un tableau C de type `T[N]`). L'implantation sera cependant un peu plus subtile dans le cas où le tableau est grand pour éviter qu'il ne fasse déborder la pile et pour éviter que certaines opérations comme `move` ou `swap` soient trop coûteuses.

Les trois classes `template` définies ci-dessous auront la signature suivante :

```
template<typename T, std::size_t N>
class my_new_array {
  ...
};
```

2.1 Petits tableaux

Définissez une classe template `small_array<T,N>` contenant un champ privé ayant le type `T[N]`. Ajoutez les versions par défaut de toutes les méthodes spéciales : constructeur par défaut, constructeur par copie, constructeur par transfert, affectation par copie, affectation par transfert, destructeur.

Ajoutez deux opérateurs à la classe permettant d'accéder aux éléments comme si c'était un simple tableau :

```
T& small_array<T,N>::operator[](std::size_t i);
const T& small_array<T,N>::operator[](std::size_t i) const;
```

Question : pourquoi faut-il définir deux opérateurs crochets quasiment identiques ? (Voir le code de test ci-dessous pour un indice.)

Ajoutez à ces opérateurs des assertions pour empêcher le programme de continuer son exécution en cas d'accès hors des bornes du tableau.

Question : est-il possible de marquer ces opérateurs comme étant `noexcept` ?

Testez votre classe en utilisant le code ci-dessous :

```
int main() {
  small_array<int, 4> t;
  t[2] = 42;
  small_array<int, 4> const u = t;
  for (std::size_t i = 0; i < 4; ++i) {
    std::cout << '[' << i << "] = " << u[i] << '\n';
  }
  t[4] = 0; // assertion failed!
}
```

Question : est-ce que votre code affiche des valeurs surprenantes pour les cases autres que la deuxième ? Si oui, c'est normal (et sinon, c'est un coup de chance). Pourquoi ?

Ajoutez deux méthodes qui se comportent comme les opérateurs crochets, mais qui lèvent cette fois des exceptions quand les accès ont lieu hors des bornes :

```
T& small_array<T,N>::at(std::size_t i);
const T& small_array<T,N>::at(std::size_t i) const;
```

Testez vos nouvelles méthodes en modifiant le code de test ci-dessus.

2.2 Grands tableaux

Testez votre classe avec le code suivant :

```
int main() {
  small_array<int, 1000 * 1000 * 10> t;
  t[2] = 42;
}
```

Question : pourquoi le programme plante-t-il ?

Définissez une classe template `large_array<T,N>` dont le champ privé a maintenant le type suivant :

```
std::unique_ptr<small_array<T,N>>
```

Ajoutez des opérateurs crochets et des méthodes `at` permettant d'accéder aux éléments du tableau.

Question : pourquoi le constructeur par défaut fourni par le compilateur ne convient-il pas ?

Définissez un constructeur par défaut et testez votre classe avec le code suivant :

```
int main() {
  large_array<int, 1000 * 1000 * 10> t;
  t[2] = 42;
}
```

Les versions du constructeur par copie et de l'opérateur d'affectation par copie fournies par le compilateur ne conviennent pas non plus. Définissez des versions adaptées à `large_array`.

Complétez votre code de test en vous inspirant de celui utilisé pour les petits tableaux afin de vérifier que votre constructeur par copie fonctionne correctement.

Fournissez une méthode `swap` qui échange le contenu de deux tableaux larges en temps constant :

```
void large_array<T,N>::swap(large_array&);
```

Proposez une variante de l'opérateur d'affectation par copie qui fournisse une garantie plus forte concernant les exceptions : si une exception est levée lors de la copie, le tableau original est rendu inchangé plutôt qu'à moitié modifié. (Note : cette garantie n'est pas fournie par `small_array`.)

Question : quel est l'inconvénient de cette variante ?

Une fonction template incorrecte n'est généralement pas détectée par le compilateur tant qu'elle n'est pas utilisée par du code non-template. Modifiez le code de test afin que l'opérateur d'affectation par copie soit lui-aussi utilisé, de même pour la méthode `swap`.

2.3 Tableaux malins

Définissez un type template qui se résout vers `small_array<T,N>` s'il est suffisamment petit (inférieur à 16 octets par exemple) et vers `large_array<T,N>` sinon.

Testez votre type avec le code suivant en faisant varier la taille passée en paramètre. On pourra ajouter une assertion dans le constructeur de `large_array` pour s'assurer qu'il n'est pas appelé avec un petit `N`.

```
int main() {
  my_array<int, 1000 * 1000 * 10> t;
  t[2] = 42;
}
```

Note : la construction `typedef` n'accepte pas les paramètres template. On pourra utiliser la construction `template<typename T, std::size_t N> using my_array = ...` à la place.