

Programmation C++ Avancée

Joel Falcou

Guillaume Melquiond

12 décembre 2017

L'objectif de ce TP 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 {
    ...
};
```

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

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

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`.

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.