Christophe PICHAUD
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com |
www.windowscpp.net

NeOS-SDI makes IT work

Utiliser shared_ptr<T> en C++ pour la gestion des ressources

Le C++ n'a aucune leçon à recevoir des langages managés (C#, Java) pour la gestion automatique du cycle de vie des ressources car nous avons depuis TR1, les « smart pointers » ou pointeurs intelligents. Nous avons à notre disposition des templates comme std::shared_ptr<T> et std::unique_ptr<T>, et il est possible de nous passer de delete dans notre code, ceci sans sacrifier le contrôle et la performance. Le C++ est supérieur à tous ces langages soi-disant « safe and secure » qui possèdent une machinerie en arrière-plan qui compense les opérations que leurs développeurs n'ont pas à faire.

Ce mois-ci, je vous propose l'adaptation d'un paragraphe du livre de Scott Meyers « Effective Modern C++ », « 42 specific ways to improve your use of C++11 and C++14 ». Je vous propose donc de commenter l'élément 19 du livre qui se nomme Item 19 : utilisation du shared_ptr pour la gestion des ressources partagées.

Présentation de la mécanique

std::shared_ptr est le moyen en C++ 11 de combiner le meilleur de ces mondes. Un objet accédé via std::shared_ptr possède un cycle de vie géré par ses pointeurs au travers de de la propriété partagée. Aucun std::shared_ptr ne possède l'objet. A la place, tous les std::shared_ptrs qui pointent dessus s'entendent pour que sa destruction soit faite quand il n'est plus nécessaire.

Quand le dernier std::shared_ptr pointant sur un objet arrête de pointer dessus (parce que le std::shared_ptr est détruit ou pointe sur un objet différent), ce std::shared_ptr détruit l'objet sur lequel il pointe. Comme avec le garbage collector, les clients ne sont pas obligés de gérer eux-mêmes le cycle de vie des objets pointés, mais avec le mécanisme des destructeurs, la destruction de ces objets est déterministe.

Un std::shared_ptr peut dire que c'est le dernier à pointer sur une ressource en consultant le compteur de références de la ressource, une valeur associée avec la ressource qui garde trace de combien de std::shared_ptr pointent dessus. Les constructeurs de std::shared_ptr incrémentent ce compteur, les destructeurs std::shared_ptr décrémentent le compteur, et les opérateurs de copie font les deux.

(si sp1 et sp2 sont des std::shared_ptr sur différents objets, l'affectation "sp1 = sp2;" modifie sp1 de telle manière qu'il pointe sur l'objet pointé par sp2. L'effet de cette affectation et que le compteur de référence pour cet objet originalement pointé par sp1 est décrémenté, pendant que pour l'objet pointé par sp2 est incrémenté.) Si un std::shared_ptr voit son compteur de référence à zéro après une opération de décrémentation, plus aucun std::shared_ptrs ne pointe sur la ressource donc le std::shared_ptr le détruit.

Les détails

L'existence du compteur de référence a des implications de performance :

- la taille d'un **std::shared_ptr est double comparée à celle d'un pointeur raw,** parce qu'il contient un pointeur explicite vers la ressource et aussi vers le compteur de référence.
- **La mémoire pour le compteur de référence doit être allouée dynamiquement**. Conceptuellement,

Le compteur de référence est associé avec l'objet pointé mais les objets pointés n'en savent rien. Ils n'ont aucune place pour stocker un compteur de référence. (N'importe quel objet – même les types built-in – peuvent être gérés par std::shared_ptr.) La figure explique que le coût de l'allocation dynamique est évitée quand le std::shared_ptr est créé par std::make_shared, mais il est des cas où std::make_shared ne peut pas être utilisé. Dans ce cas, le compteur de référence est stocké en données allouées dynamiquement.

- **Les incrémentations et décrémentions du compteur de référence doivent être atomic,** parce qu'il peut y avoir simultanément des lecteurs et écrivains dans différents threads.

Par exemple, un std::shared_ptr qui pointe sur une ressource dans un thread peut exécuter son destructeur (décrémentation du compteur de référence pour la ressource qu'il pointe), pendant que dans un thread différent, un std::shared_ptr sur le même objet peut être copié (et donc incrémenter le même compteur de référence). Les opérations atomic sont typiquement plus lentes que les opérations non-atomic, mais vous devez savoir que même sur une telle opération, la lecture et l'écriture d'un compteur de référence a un prix coûteux.

Votre curiosité s'est-elle éveillée quand je vous écris qu'un constructeur de std::shared_ptr ne fait qu'incrémenter le compteur de référence pour l'objet sur lequel il pointe ?

Créer un std::shared_ptr qui pointe sur un objet fait plus que pointer sur un objet, donc pourquoi est-ce qu'on ne peut pas incrémenter le compteur de référence à tous les coups ?

La construction par déplacement (move construction), c'est la raison. Faire une construction de déplacement d'un std::shared_ptr vers un autre std::shared_ptr positionne la source std::shared_ptr à null et ça veut dire que le vieux std::shared_ptr arrête de pointer sur la ressource au moment où le nouveau std::shared_ptr démarre. En résultat, aucune manipulation de compteur de référence n'est requise. Le déplacement de std::shared_ptr est plus rapide que la copie car la copie requièrt l'incrémentation du compteur de référence, mais le déplacement non. C'est vrai pour l'affectation en construction, donc la construction par déplacement est plus rapide que la copie par construction et l'affectation de déplacement est plus rapide que l'affectation de copie. Comme std::unique_ptr, std::shared_ptr utilise delete comme son mécanisme de destruction de ressource par défaut, mais il supporte aussi les deleters spécifiques. Le design de cette fonctionnalité n'est cependant pas tout à fait standard pour un std::unique_ptr. .

Pour std::unique_ptr, le type du deleter fait partie du type de smart pointer. Pour std::shared_ptr, ce n'est pas :

```
auto loggingDel = [](Widget *pw) // custom deleter
{
makeLogEntry(pw);
delete pw;
};
std::unique_ptr<        // deleter type is
Widget, decltype(loggingDel)    // part of ptr type
> upw(new Widget, loggingDel);
std::shared_ptr<Widget>    // deleter type is not
spw(new Widget, loggingDel); // part of ptr type
```

Le design de std::shared_ptr est plus flexible. Considérons deux std::shared_ptr
<Widget>s, chacun avec un custom deleter d'un type différent (parce que les custom deleters sont spécifiés via des lambdas):

```
auto customDeleter1 = [](Widget *pw) { … };  // custom deleters,
auto customDeleter2 = [](Widget *pw) { … }; // each with a
// different type
std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

Parce que pw1 et pw2 ont le même type, ils peuvent être placés dans un container d'objets de ce type :

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

Ils peuvent aussi être assignés ou passés à une fonction qui prend un paramètre de type std::shared_ptr<Widget>. Aucune de ces possibilités ne peuvent être effectuées avec des std::unique_ptrs qui diffèrent dans le type de leur custom deleters parce que le type du custom deleter modifie le type du std::unique_ptr.

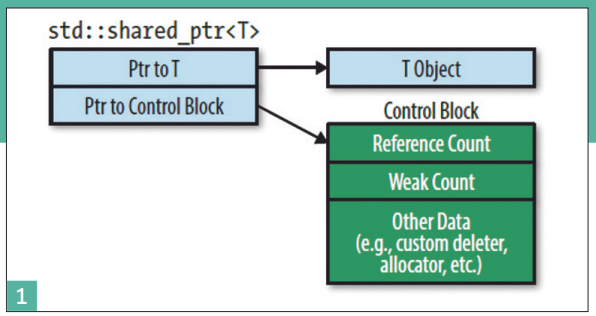
Une autre différence pour std::unique_ptr, spécifiant un custom deleter ne change pas la taille d'un objet std::shared_ptr. Quel que soit le deleter, un objet std::shared_ptr vaut 2 pointeurs en taille. C'est super mais bon… Les custom deleters peuvent être des objets fonctions et peuvent contenir n'importe quelle quantité de data. Ça veut dire qu'ils peuvent être larges. Comment un std::shared_ptr qui contient un custom deleter de taille arbitraire fait-il sans utiliser plus de mémoire ? Il ne peut pas. Il doit utiliser plus de mémoire. Cependant, cette mémoire ne fait pas partie de l'objet std::shared_ptr. C'est sur le cas où, si le créateur du std::shared_ptr prend avantage du support pour les custom allocators, c'est là que la mémoire gérée par l'allocateur est localisée. On remarque qu'un objet std::shared_ptr contient un pointeur vers le compteur de référence pour l'objet sur lequel il pointe. C'est vrai mais plus largement, le compteur de référence fait partie d'une structure de données plus large qui est le « control block ». Il y a un control block pour chaque objet géré par un std::shared_ptr.

Si spécifié, le control block contient en plus du compteur de référence, une copie du custom deleter si spécifié, un custom allocator si spécifié et aussi d'autres informations comme un compteur de référence. Voici à peu près à quoi cela ressemble : 1

Un objet control block est créé par la fonction qui crée le premier std::shared_ptr vers un objet. Mais ce n'est pas évident, donc les règles suivantes pour la création du control block sont :

- **std::make_shared en crée toujours un.** C'est le meilleur endroit car on démarre ;
- **un control block est créé quand std::shared_ptr est construit à partir d'un unique-ownership pointer (std::unique_ptr ou std::auto_ptr).**

Les pointeurs unique-ownership n'utilisent pas de control blocks. De plus,



1

lors de sa construction, le std::shared_ptr prend la possession de l'objet et le uniqueownership pointeur est mis à null.

- **Quand un constructeur de std::shared_ptr est appelé avec un raw pointeur, il crée le control block.**

Si vous voulez créer un std::shared_ptr à partir d'un objet qui a déjà un control block, vous allez probablement passer un std::shared_ptr ou un std::weak_ptr comme argument au constructeur et non un raw pointeur.

Les constructeurs de std::shared_ptr qui prennent un std::shared_ptr ou un std::weak_ptr en argument de constructeur ne créent pas de nouveau control block car ils s'appuient sur les smart pointers passés en argument pour pointer sur un control block.

La conséquence de ces règles est que construire plus d'un std::shared_ptr depuis un simple raw pointer vous amène droit dans une partie au comportement indéfini car il y aura plusieurs control blocks. Cela impliquerait la destruction multiple de l'objet et c'est mauvais ! Exemple de mauvais code :

```
auto pw = new Widget; // pw is raw ptr
…
std::shared_ptr<Widget> spw1(pw, loggingDel); // create control
// block for *pw
…
std::shared_ptr<Widget> spw2(pw, loggingDel); // create 2nd
// control block
// for *pw!
```

La création d'un pointeur raw vers un objet alloué dynamiquement est mauvaise car cela va à l'encontre des conseils précédemment évoquées : préférer les smart pointers aux raw pointeurs.

La ligne qui crée pw est foireuse mais au moins, elle ne crée pas de comportement indéfini. Maintenant que le constructeur de spw1 est appelé avec un raw pointeur, il crée un control block (et aussi un compteur de référence). Dans ce cas, c'est *pw. Le problème c'est que spw2 est appelé avec le même raw pointeur et donc il y a création d'un control block pour *pw. *pw possède maintenant 2 compteurs de référence susceptibles de passer à zéro et d'essayer de détruire *pw deux fois ! La seconde destruction est responsable d'un comportement indéfini qui sera certainement un plantage en règle. Il y a deux leçons au regard de std::shared_ptr utilisées ici. Premièrement, essayer de passer un raw pointer au constructeur d'un std::shared_ptr. L'alternative habituelle est d'utiliser std::make_shared mais dans notre exemple, nous utilisons un custom deleter et ce n'est pas possible avec std::shared_pr. Deuxièmement, si vous devez passer un raw pointer au constructeur de std::shared_ptr, passez le résultat du new directement au lieu de créer une variable raw pointer.

```
std::shared_ptr<Widget> spw1(new Widget, // direct use of new
loggingDel);
```

Créons-nous un second std::shared_ptr à partir du même raw pointer ? Non, à la place nous allons créer spw2 en fournissant spw1 comme initialisation dans le constructeur par copie et cela ne pose aucun problème :

```
std::shared_ptr<Widget> spw2(spw1);    // spw2 uses same
// control block as spw1
```

Dans ce cas, on va avoir plusieurs control blocks qui protègent le même raw pointer. Prenons un exemple pour que le programme gère des Widgets à traiter :

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Voici la classe Widget:

```
class Widget {
public:
…
void process();
…
};
```

Voici la méthode Process :

```
void Widget::process()
{
…                              // process the Widget
processedWidgets.emplace_back(this);          // add it to list of
}                          // processed Widgets;
// this is wrong!
```

C'est une erreur de passer le this, pas d'utiliser emplace_back. Pour gérer le this, il faut dériver de std::
enable_shared_from_this.

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
…
void process();
…
};
```

std::enable_shared_from_this définit une fonction membre qui crée un std::shared_ptr pour l'objet courant sans dupliquer les control blocks. Voici une implémentation de process :

```
void Widget::process()
{
// as before, process the Widget
…
// add std::shared_ptr to current object to processedWidgets
processedWidgets.emplace_back(shared_from_this());
}
```

En interne, shared_from_this regarde le control block pour l'objet courant et il crée un nouveau std::shared_ptr qui pointe sur ce control block. Le design est fonction de l'objet courant qui possède un control block. Pour ce cas-là, il faut qu'il existe un std::shared_ptr qui pointe sur l'objet courant.
Si aucun std::shared_ptr n'existe, le comportement est indéfini. En gros, c'est le plantage assuré.
Pour éviter qu'une fonction membre appelle une fonction qui appelle shared_from_this
Avant que un std::shared_ptr pointe sur l'objet, les classes qui héritent de std::enable_shared_from_this déclarent leurs constructeurs private et ont des clients qui créent des objets en appelant une fonction factory qui retourne un std::shared_ptrs. Widget, par exemple, pourrait ressembler à ça :

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
// factory function that perfect-forwards args
// to a private ctor
template<typename... Ts>
static std::shared_ptr<Widget> create(Ts&&... params);
…
void process(); // as before
…
private:
… // ctors
};
```

Maintenant, nous savons quand il faut limiter le nombre de block controls pour un std::shared_ptr.
Un control block vaut en taille quelques octets, bien que les custom deleters et allocators peuvent les rendre plus larges. L'implémentation est en réalité plus complexe que l'on ne pense. Il y a de l'héritage et même une fonction virtuelle qui est utilisée pour le pointeur sur objet soit proprement libéré. On est maintenant conscient que std::shared_ptr apporte le coût d'une machinerie complexe.
Après avoir pris connaissance des allocations dynamiques des controls blocks, des deleters et allocators, de la machinerie de la fonction virtuelle, des manipulations de compteurs de référence, vous vous dites qu'il y a du monde… Ne prenez pas peur.
Ce n'est pas la meilleure solution pour chaque problème de gestion de ressources mais pour la fonctionnalité qu'ils fournissent, les std::shared_ptrs offrent un coût raisonnable.
Dans des conditions ou le deleter par défaut et l'allocateur par défaut sont utilisés et que le std::shared_ptr est créé par std::make_shared, le control block n'est que de 3 words en taille.
Déréférencer un std::shared_ptr n'est pas plus coûteux que de déréférencer un raw pointer.
Réaliser une opération requiert une manipulation du compteur de référence (comme l'utilisation de constructeur de copie ou l'affectation par copie ou la destruction) via une ou deux opérations atomic. Ce sont des instructions simples peu coûteuses. La machinerie de fonction virtuelles dans le control block est utilisée une fois par objet géré par std::shared_ptrs quand l'objet est détruit.
La plupart du temps, utiliser std::shared_ptr est préférable pour gérer le cycle de vie d'un objet partagé à la main. Si le partage n'est pas nécessaire, vous pouvez utiliser std::unique_ptr. Il est possible de passer de l'un à l'autre mais uniquement dans le sens std::unique_ptr vers std::shared_ptr.

Note

En échange de ces coûts modestes, vous avez un management de cycle de vie automatique des ressources allouées dynamiquement.

A retenir

- Les std::shared_ptr offrent un système qui se rapproche du garbage collector pour le management du cycle de vie des ressources partagées.
- Comparé à std::unique_ptr, les objets std::shared_ptr sont deux fois plus gros car il y a le control block et les manipulations atomic du compteur de références.
- La destruction de ressource par défaut est via delete mais les custom deleters existent. Le type de deleter n'a aucun effet sur le type std::shared_ptr.
- Eviter de créer des std::shared_ptr à partir de variables de type raw pointers.

Conclusion

En tant qu'utilisateur du std::shared_ptr<T>, tout est simple. En interne, c'est plus compliqué et remarquablement bien fait. Pouvoir se passer du new dans un code est une fonctionnalité géniale. Tout est automatique et pourtant on est en C++. Une fois de plus : « Power and Performance ». •