Smart Pointer

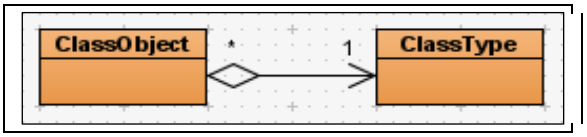
Christophe [Groove] Riccio
www.g-truc.net
Création : 07/04/2006
Mise à jour : 08/04/2006

Introduction

Le smart pointer est une classe a qui l'on confit la responsabilité d'une allocation dynamique. Il existe un grand nombre d'implémentation, possible celle traitée ici est parfaitement robuste et permet au programmeur de se passer des opérateurs delete, delete[] s'il le souhaite. Plus intéressant encore, le smart pointer permet de résoudre des situations complexes de manières très simple.

1. Association pénible à résoudre

Admettons que nous ayons le cas suivant à résoudre :



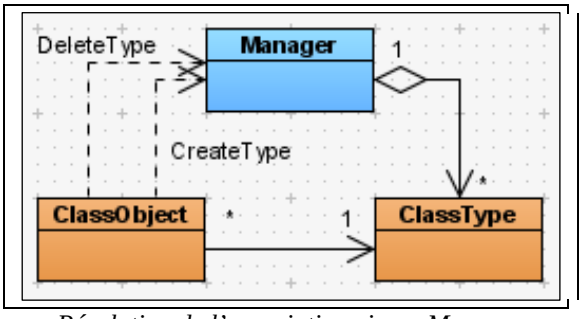
Association particulièrement difficile à résoudre en C++

Pour ceux qu'il nous pas fait UML en 2ème langue, voici une traduction française:
- Une instance de ClassType est associée à plusieurs instances de ClassObject
- Une instance de ClassObject est associée à une instance de ClassObject
- Une instance de ClassObject peut accéder aux données d'une instance de ClassType.
- Une instance de ClassType ne peut pas accéder aux données d'une instance de ClassObject.
- Les instances de la classe ClassType sont gérés par la classe ClassObject via une agrégation, c'est-à-dire usuellement en C++ via une allocation dynamique, donc ClassObject est responsable du delete.

Le problème de ce cas provient du fait que toutes les instances de la classe ClassObject ne pourront pas détruire une instance de ClassType sans provoquer une erreur du type « Objet déjà détruit ».

2. Intervention d'un manager

Une première approche pour résoudre ce problème est d'utiliser un manager, c'est-à-dire une tierce classe qui va référencer les instances de ClassType



Résolution de l'association via un Manager

Les flèches en pointillés se lisent:
- ClassObject dépend de Manager pour CreateType
- ClassObject dépend de Manager pour DeleteType

CreateType et DeleteType représentent les actions de créer et détruire des instances de ClassType.

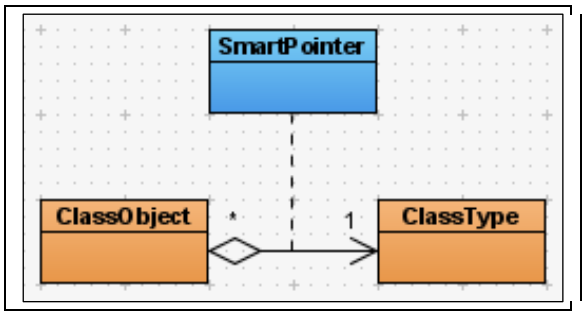
Typiquement ClassObject perd la responsabilité de la mémoire de ClassType, cette responsabilité est transférée au manager, il faut donc passer par ce manager pour créer une instance de ClassType. Il faut aussi passer par ce manager pour libérer les instances de ClassType.

Lorsque de la création d'une instance de ClassType, le manager vérifie si l'instance demandée n'est pas déjà en mémoire. Si oui, un compteur de références est incrémenté, si non, l'instance est créée et le compteur de références initialisé à 1.

Pour la destruction des instances de ClassType, nous pouvons soit attendre la destruction du manager soit demander à ClassObject de prévenir le manager quand elle n'a plus besoin de son instance de ClassType. Le compteur de références est ainsi décrémenté et la mémoire libéré quand ce compteur revient à 0.

3. Intervention d'un smart pointer

Une seconde approche pour résoudre ce problème consiste à utiliser un smart pointer.



Résolution de l'association via un SmartPointer

La ligne en pointillé se lit:
- L'association entre ClassType et ClassObject est résolue au moyen de SmartPointer.

Nous voici bien avancé ! Il n'y a-t-il pas un diagramme plus explicité ? Qu'elle est le lien du Smart Pointer avec les autres classes ? Comment fonctionne ce SmartPointer ?

Clairement, il n'y a pas de diagramme plus explicité à mon sens. En effet, l'utilisation d'un smart pointer peut-être totalement transparent au delà de la création de l'instance. Tout ce passe comme si nous utilisions un pointeur classique, qui dans ce cas serait un pointeur de ClassType.

Prenons, un exemple en C++ d'utilisation de ce smart pointer:

```
01  class ClassObject
02  {
03  public:
04      ClassObject(const smart_ptr<ClassType>& Type) :
05              _Type(Type)
06      {}
07
08  private:
09      smart_ptr<ClassType> _Type;
10  };
11
12  int main()
13  {
14      smart_ptr<ClassType> Type = new ClassType;
15
16      ClassObject Object1(Type);
17      ClassObject Object2(Type);
18  }
```

Le code précédant est parfaitement saint, il n'y a pas de fuite de mémoire. Lors de la ligne 14 nous créons une instance de ClassType dont nous confions la responsabilité à un objet de type smart_ptr<ClassType>. Lors de la création de cet objet, un compteur de références est créé dynamiquement et initialisé à 1. Les instances ClassObject contiennent une instance de smart_ptr<ClassType>. Lors de la création du premier ClassObject la construction de _Type, provoque

un incrément du compteur de références de Type. Des pointeurs sur ce compteur et sur l'instance de ClassType sont conservés par chaque instance de smart_ptr<ClassType> de telle manière que si une instance vient à disparaître alors le compteur de références est décrémenté. Lorsque sa valeur arrive à zéro alors l'instance de ClassType est détruite ainsi que le compteur de références.

4. std::auto_ptr

La bibliothèque standard du C++ dispose d'un outil nommé auto_ptr qui est tout à fait à même de résoudre correctement l'exemple précédent. Il s'agit d'une sorte de smart pointer mais ses capacités sont particulièrement limité car il ne possède pas de compteur de références. Son fonctionnement est très simple. Lors ce que l'instance std::auto_ptr<ClassType> est détruite alors la mémoire dont il est responsable est libérée.

Voici un exemple d'utilisation d'auto_ptr :

```
01  int main()
02  {
03      std::auto_ptr<int> Ptr1 = new int(76);
04      std::auto_ptr<int> Ptr2(Ptr1);
05      int* Ptr3 = Ptr2.release();
06      delete Ptr3;
07  }
```

Lors de la ligne 3, une instance de int est créée et la responsabilité de sa mémoire est confiée à Ptr1. La ligne 4 est à comprendre comme un transfère de responsabilité, c'est maintenant Ptr2 qui gère la mémoire de notre entier. Lors de la ligne 5, nous choisissons de retirer la responsabilité confiée à Ptr2. Nous devons alors libérer nous même la mémoire allouée en ligne 3.

L'auto_ptr est donc en quelques sortes un smart pointer à responsabilité unique de la mémoire. La responsabilité partagée du smart pointer traité ici nous permet de passer outre des cas telle que le suivant:

```
01  void f(const std::auto_ptr<int>& Ptr1)
02  {
03      // La mémoire de Ptr1 est confié à Ptr2
04      std::auto_ptr<int> Ptr2(Ptr1);
05      ++*Ptr2;
06      // Fin de la fonction f, Ptr2 est détruit
07      // La mémoire dont il est responsable est libérée
08  }
09
10  int main()
11  {
12      std::auto_ptr<int> Ptr1 = new int(0);
13      f(Ptr1);
14      printf("%d", *Ptr1); // Erreur!!!
15  }
```

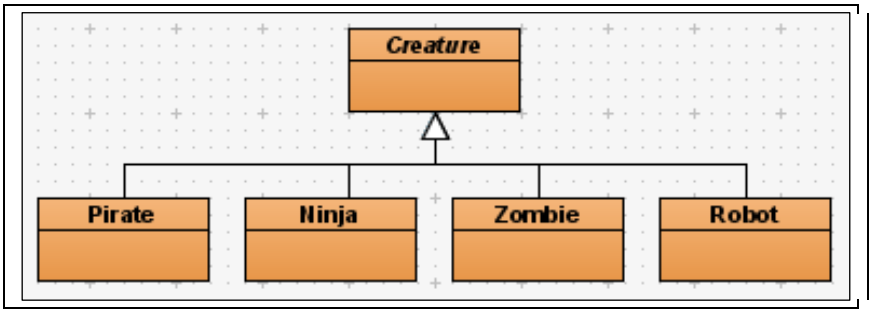
5. Utilités du smart pointer

Dans le cadre de la programmation d'un jeu, les associations à responsabilité multiple apparaissent principalement pour la gestion des ressources. Charger deux fois une même texture en mémoire est totalement inutile, nous préférons maintenir plusieurs références sur une seule texture.

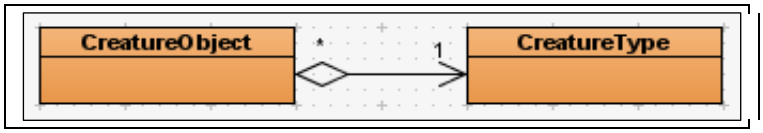
Cependant, le cas des textures n'est pas forcément le plus pertinent. En effet, l'idée de l'utilisation d'un smart pointer implique, l'envie et la possibilité de se passer d'une zone où seront références les instances en mémoire (Un manager). L'utilisation du smart pointer est plus fertile pour les instances qui sont créés exclusivement lors du chargement du jeu et qui n'auront plus besoin d'être créer pendant l'exécution du jeu.

Prenons le scénario suivant par exemple. Nous souhaitons créer un jeu, où le monde est peuplé de créatures de quatre types. Notre ambition est de créer un jeu data-driven, c'est-à-dire que la description des caractéristiques des créatures sera faites hors du code, par exemple dans un fichier XML. C'est une pratique qui devient courante dans l'industrie, comme avec Age of Mythology et Age of Empire 3.

D’autres jeux préfèrent créer leur propre format de données comme Ground Control 2 mais le principe est identique.



Solution 1 : basée sur l’héritage, dites hard-coded



Solution 2 : basée sur une description externe et générique, dites data-driven

La première solution est devenue relativement obsolète du fait de la diversité du nombre d’entité dans un jeu qui conduit rapidement à une explosion hiérarchique. De plus dans l’industrie les besoins évoluent facilement au gré des éditeurs ce qui peu impliquer un remaniement important de l’arbre d’héritage. Cependant pour des projets amateurs avec un nombre de type d’entités faible, cette solution peut parfaitement convenir. L’exemple de Dune 2 legacy entièrement basé sur un arbre d’héritage est révélateur de la robustesse de la solution.

La deuxième solution est plus aventureuse mais vraiment plus puissante. Il devient possible de créer de nouvelles unités sans toucher au code. CreatureType charge depuis un fichier les caractéristiques d’une créature, celles-ci sont ainsi utilisées par CreatureObject pour définir le comportement de la créature.

Nous trouvons dans ce deuxième cas le schéma d’utilisation du smart pointer. Nous pouvons raisonnablement penser que dans de nombreux jeux les entités sont créées seulement à la création de la partie, il n’est pas nécessaire de conserver la référence au type d’objet dans une zone autre que l’objet lui-même, ce qui nous évitera un impondérable singleton.

```
01  void Game::LoadEntity()
02  {
03      smart_ptr<CreatureType> Type[Typenames.size()];
04
05      for(int i = 0; i < Typenames.size(); ++i)
06              Type[i] = new CreatureType(Typenames[i]);
07      Typenames.clear();
08
09      for(int i = 0; i < ObjectIndex.size(); ++i)
10              Objects[i] = new CreatureObject(Type[ObjectIndex[i]]);
11  }
```

Il existe bien sur une multitude d’utilisations possibles tel que pour résoudre le problème de gestion de la mémoire d’un graphe, pour les fonctions qui retourne un pointer du un bloc que mémoire qu’elle a alloué ou simple pour tout problème qui rend la gestion de la mémoire très difficile.

6. Implémentation du smart pointer à partir des templates

Pour implémenter notre smart pointer nous allons nous appuyer sur les templates du langage C++.

```
01  template <typename T, bool Array = false>
02  class smart_ptr
03  {};
```

Le type T est le type d'instance dont le smart pointer a la responsabilité. Array nous permet d'appeler le bon opérateur delete lors de la libération de la mémoire de l'instance. Exemple:

```
01  int main()
02  {
03      smart_ptr<ClassType, true> Type1 = new ClassType[10];
04      smart_ptr<ClassType, false> Type2 = new ClassType;
05      smart_ptr<ClassType> Type3 = new ClassType;
06
07      ...
08  }
```

7. Implémentation du comportement d'un pointer

Nous souhaitons que l'utilisation du smart pointer soit la plus transparente possible, c'est-à-dire que la classe smart_ptr s'utilise comme un pointer. Pour cela nous surchargeons les opérateurs *, ->, == et !=. Ceci nous permet d'accéder à la valeur de l'instance comme si une instance de smart_ptr était vraiment un pointer. De plus nous pouvons effectuer les tests de comparaisons des adresses du pointer directement par l'objet smart_ptr.

```
template <typename T, bool Array = false>
class smart_ptr
{
public:
...
        T& operator*();
        T* operator->();
        const T& operator*() const;
        const T* operator->() const;
        bool operator==(const smart_ptr& SmartPtr) const;
        bool operator!=(const smart_ptr& SmartPtr) const;
...
private:
        T* _Pointer;
...
};

template <typename T, bool Array>
T& smart_ptr<T, Array>::operator*()
{
        return *_Pointer;
}

template <typename T, bool Array>
T* smart_ptr<T, Array>::operator->()
{
        return _Pointer;
}

template <typename T, bool Array>
const T& smart_ptr<T, Array>::operator*() const
{
        return *_Pointer;
}

template <typename T, bool Array>
const T* smart_ptr<T, Array>::operator->() const
{
        return _Pointer;
}

template <typename T, bool Array>
bool smart_ptr<T, Array>::operator==(const smart_ptr<T, Array>& SmartPtr) const
{
        return _Pointer == SmartPtr._Pointer;
}

template <typename T, bool Array>
bool smart_ptr<T, Array>::operator!=(const smart_ptr<T, Array>& SmartPtr) const
{
        return _Pointer != SmartPtr._Pointer;
}
```

8. Les différentes méthodes pour passer une valeur à une classe

Pour garantir que notre classe smart pointer ne comporte pas de fuite mémoire, nous devons comprendre comment une classe se construit et peut changer de valeur. Prenons l'exemple suivant en supposant que le code compile, quels opérateurs et constructeurs la classe Class utilise t'elle ?

```
01      int main()
02      {
03              Class Class1(76);
04              Class Class2(Class1);
05              Class Class3 = Class1;
06              Class1 = Class2;
07              Class2 = 76;
08      }
```

Ligne 3:
Class::Class(int i){...}

Ligne 4:
Class::Class(const Class& c){...}

Ligne 5:
Class::Class(const Class& c){...}
Non, ce n'est pas l'opérateur =. Cette syntaxe est l'héritage du langage C.

Ligne 6:
Class& Class::operator=(const Class& c);
L'opérateur = est appelé pour la modification d'une instance déjà créé.

Ligne 7:
Class& Class::operator=(int i);

Cette liste d'opérations représente la liste complète des opérations qui vont affecté la valeur du pointeur du smart pointer, nous devons donc toutes les définir en substituant le type int par notre pointeur sur un type T.

9. Implémentation du compteur de références

Voici le reste de la déclaration de notre smart pointer :

```
template <typename T, bool Array = false>
class smart_ptr
{
public:
        smart_ptr();
        smart_ptr(const smart_ptr& SmartPtr);
        smart_ptr(T* Pointer);
        ~smart_ptr();

        smart_ptr& operator=(const smart_ptr& SmartPtr);
        smart_ptr& operator=(T* Pointer);

        ...
private:
        void _Release();

        int* _RefCounter;
        T* _Pointer;
};
```

Le constructeur par défaut initialise le pointeur et le compteur de références à la valeur nulle. Cette opération nous permettra de vérifier si un pointer a été affecté au smart pointer.

```
template <typename T, bool Array>
```

```
smart_ptr<T, Array>::smart_ptr() :
	_RefCounter(0),
	_Pointer(0)
{}
```

Lorsque le smart pointer est créé avec un pointer en paramètre alors nous stockons ce pointer et créons le compteur de références avec la valeur 1. Ce pointeur est référencé une seule fois pour le moment.

```
template <typename T, bool Array>
smart_ptr<T, Array>::smart_ptr(T* Pointer) :
	_Pointer(Pointer)
{
	_RefCounter = new int(1);
}
```

Nous utilisons ici le constructeur par copie du smart pointer. Ce présente alors deux cas. Si le smart pointer passé en paramètre a été initialisé avec un pointer alors nous procédons à l'incrémentation du compteur de références qui accompagne le smart pointer. Sinon le smart pointer est créé avec des valeurs nulle provenant du paramètre.

```
template <typename T, bool Array>
smart_ptr<T, Array>::smart_ptr(const smart_ptr<T, Array>& SmartPtr) :
	_RefCounter(SmartPtr._RefCounter),
	_Pointer(SmartPtr._Pointer)
{
	if(_RefCounter != 0)
		++(*RefCounter);
}
```

Les choses se complexifient légèrement avec la surcharge de l'opérateur égale. En effet, il est possible que le smart pointer contiennent déjà une valeur, il faut donc libéré la valeur actuelle pour la remplacer par la valeur du paramètre. La libération est effectuée par la fonction _Release décrite plus bas.

```
template <typename T, bool Array>
smart_ptr<T, Array>& smart_ptr<T, Array>::operator=(T* Pointer)
{
	_Release();

	_Pointer = Pointer;
	_RefCounter = new int(1);

	return *this;
}
```

C'est la même chose avec l'opérateur suivant sauf qu'il faut tenir compte de la possible existence d'une valeur dans le smart pointer passé en paramètre.

```
template <typename T, bool Array>
smart_ptr<T, Array>& smart_ptr<T, Array>::operator=(const smart_ptr<T, Array>&
SmartPtr)
{
	_Release();

	_Pointer = SmartPtr._Pointer;
	_RefCounter = SmartPtr._RefCounter;
	if(_RefCounter != 0)
		++(*_RefCounter);

	return *this;
}
```

Bien entendu le destructeur doit aussi libérer l'instance pointée. Attention, pour cette implémentation, le destructeur n'est pas virtuel. Il est donc fortement déconseillé d'hériter de la classe smart_ptr.

```
template <typename T, bool Array>
smart_ptr<T, Array>::~smart_ptr()
{
	_Release();
```

```
}
```

Nous voici donc arrive à la dernière fonction de la classe. Elle vérifie si le smart pointer contient une valeur. Si oui, elle décrémente la valeur du compteur de références. Si la valeur du compteur de références est nulle alors ce smart pointer est le dernière à faire référence au pointeur. Le pointeur peut-être libéré.

```
template <typename T, bool Array>
void smart_ptr<T, Array>::_Release()
{
	if(_RefCounter != 0)
	{
		(*_RefCounter)--;
		if(*_RefCounter <= 0)
		{
			if(Array)
			{
				delete[] _RefCounter;
				if(_Pointer != 0)
					delete[] _Pointer;
			}
			else
			{
				delete _RefCounter;
				if(_Pointer != 0)
					delete _Pointer;
			}
		}
	}
}
```

Conclusion

Le smart pointer est un outil très puissant qui peut permettre de totalement mettre de coté l'aspect gestion de la mémoire et résout les problèmes les plus tordus. J'espère donc que ce nouveau jouet intègrera parfaitement votre trousse à outil C++.

Les seules contraintes sont un coup plus élevé pour la création et la destruction d'une instance, la possibilité d'instaurer un certain flou dans le niveau d'utilisation de la mémoire, la sensation d'abattre une fourmi avec un bazooka et un point clé, le fun de la gestion de la mémoire ^_^.

Enfin, je voudrais vous donner un conseil si vous souhaitez modifier cette implémentation : Jamais, j'ai bien dit jamais vous ne retournerez directement la valeur du pointeur stockée par le smart pointer. C'est le meilleur moyen d'avoir une erreur « segment fault », simplement parce que le smart pointer aura disparu et donc libéré l'instance dont il est responsable alors que vous utiliserez un pointeur fugitif dans un coin de votre code. Vous pouvez cependant étudier un système de « release » à la manière de std::auto_ptr bien que je le déconseille.

Pour tout commentaire, remarque et correction, vous pouvez me contacter par email qui est disponible à l'adresse www.g-truc.net/contact.html. La classe final est disponible à l'adresse : www.g-truc.net/articles/smart_ptr.zip