

# Gérer ses ressources de manière robuste en C++

par Aurélien Regat-Barrel ([Site personnel](#))

Date de publication : 09/07/2007

Dernière mise à jour :

En C++, il n'y a pas de destruction automatique des objets lorsque l'on perd leur trace dans le code source. Les objets ainsi perdus le sont définitivement, on parle alors de fuite. C'est donc au programmeur C++ qu'incombe l'entière responsabilité de gérer le cycle de vie des objets alloués. Il s'agit donc là d'une problématique centrale dans ce langage, qui doit être réfléchi et résolue de manière globale.

C'est ce que des experts ont fait, et des techniques spécifiques apportant une réponse globale au problème de la gestion des ressources (et non pas seulement au cas particulier de la mémoire) ont été développées. Ces pratiques sont à la fois robustes et élégantes, mais restent cependant peu connues et sous-utilisées. Le but de cet article est d'accroître leur notoriété au travers de leur mise en oeuvre dans le cas d'un problème classique de gestion de ressource limitée.

Introduction

Approche classique

Le problème de cette approche classique

L'approche C++ : le RAII

Mise en oeuvre : premier essai

    Présentation de shared\_ptr

    Premier essai

    Nouveaux problèmes soulevés

Nouvel essai : explorons les possibilités de shared\_ptr

Version finale

Programme de test

Liens, téléchargements, commentaires

## Introduction

Prenons le cas d'une ressource quelconque manipulée au moyen de la classe Resource, dont le but est de représenter cette ressource d'un point de vue du langage et de permettre sa manipulation en C++. Il pourrait s'agir d'une connexion réseau par exemple. Notre classe de ressource s'appellerait alors certainement Socket et non pas Resource.

Maintenant, imaginez que vous deviez contrôler le nombre d'instances de cette classe Resource. Dans notre exemple avec la classe Socket, il pourrait s'agir de limiter le nombre de connexions réseaux simultanément ouvertes. Vous vous retrouvez alors avec la contrainte de devoir comptabiliser le nombre d'instances de Resource qui sont toujours en cours d'utilisation. A priori, cela paraît enfantin. Mais comme toujours, et encore plus en C++, ce n'est pas parce qu'il y a de nombreuses façons de faire qu'elles se valent toutes. Penchons-nous donc vers un début de solution.

## Approche classique

Dans le cadre de cet article, j'ai choisi de centraliser les opérations de création et de contrôle du nombre d'instances de Resource au sein d'un gestionnaire spécialisé baptisé ResourceManager.

Cette classe ResourceManager s'apparente en fait à une factory, et possède une fonction membre New() chargée d'instancier ou non une nouvelle ressource, et d'en conserver la trace si jamais c'est le cas. Le but est de savoir à tout moment le nombre d'instances en cours d'utilisation, afin de limiter leur nombre. Ce nombre maximal d'instances est fourni comme paramètre au constructeur de ResourceManager.

### ResourceManager.h

```
class Resource;
typedef Resource* ResourcePtr;

class ResourceManager
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();
};
```

Si concevoir cette interface publique d'utilisation de ResourceManager est une chose, l'implémenter en est une autre. Car un problème se pose : comment connaître le nombre d'instances créées (facile) et toujours en cours d'utilisation (un peu moins facile) ? Autrement dit, comment être informé dans ResourceManager qu'un objet alloué via New() a été détruit ?

Une première solution, fréquente en programmation procédurale, consiste à fournir une fonction paire de New, chargée de la destruction des instances :

### ResourceManager.h

```
class Resource;
typedef Resource* ResourcePtr;

class ResourceManager
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();

    // Supprime une Resource allouée avec New()
    void Delete( ResourcePtr );

private:
    // Nombre d'instances en cours d'utilisation
    int NbInstances;
    // Nombre maximal d'instances autorisées
    const int MaxNbInstances;
};
```

### ResourceManager.cpp

```
#include "ResourceManager.h"
```

**ResourceManager.cpp**

```


#include "Resource.h"

ResourceManager::ResourceManager( int Max ) :
    NbInstances( 0 ),
    MaxNbInstances( Max )
{
}

ResourcePtr ResourceManager::New()
{
    if ( NbInstances < MaxNbInstances )
    {
        ResourcePtr r = new Resource();
        // On incrémente NbInstances _après_ avoir alloué
        // la ressource avec succès, afin de laisser notre
        // Manager dans un état cohérent si une exception
        // est levée lors de la création d'un objet Resource
        ++this->NbInstances;
        return r;
    }
    return 0; // NULL
}

void ResourceManager::Delete( ResourcePtr Res )
{
    if ( Res )
    {
        --this->NbInstances;
        delete Res;
    }
}
    
```

Tout comme `ResourceManager::New()` est chargé d'instancier des objets de type `Resource`, `ResourceManager::Delete()` est chargé de les détruire. On pourrait envisager de confier davantage de travail à ces fonctions, comme par exemple maintenir une liste des instances allouées, mais j'ai préféré garder les choses aussi simples que possibles.

 *Notez l'ordre des opérations au sein de `ResourceManager::New()` : l'objet `Resource` est d'abord alloué, puis le compteur `NbInstances` est mis à jour, et surtout pas l'inverse. Car si l'allocation d'un nouvel objet `Resource` venait à échouer (par manque de mémoire par exemple) et qu'une exception était levée (`std::bad_alloc`), l'objet `ResourceManager` se retrouverait alors dans un état incohérent : un objet supplémentaire `Resource` serait comptabilisé comme en cours d'utilisation alors qu'il n'existe pas.*

## Le problème de cette approche classique

Ce concept de fonctions jumelles repose sur un principe de programmation très saint : *C'est celui qui a alloué un objet qui devrait le détruire*. ResourceManager est ainsi responsable de la création mais aussi de la destruction des objets Resource au moyen de son couple de fonctions membres New() et Delete().

Si ce principe est très saint, sa mise en oeuvre dans cette première tentative l'est un peu moins, et n'est pas vraiment ce qui se fait de mieux en C++. En effet, en introduisant la fonction Delete() dans l'interface publique de ResourceManager, nous avons ajouté une contrainte de taille quant à l'utilisation de la fonction New() : le pointeur qu'elle retourne doit obligatoirement et systématiquement être libéré au moyen de Delete().

Si dans certains langages, comme en C, ce n'est déjà pas une contrainte évidente à satisfaire en n'importe quelle circonstance (un oubli est si vite arrivé), cela devient particulièrement complexe en C++ à cause du support des exceptions par le langage.

```
// Fonction quelconque qui utilise un objet Resource
void UseResource( ResourcePtr );

void Test()
{
    // on autorise 3 instances maximum
    ResourceManager mgr( 3 );

    ResourcePtr r = mgr.New();
    if ( r )
    {
        UseResource( r );
        mgr.Delete( r );
    }
}
```

Dans l'exemple ci-dessus, si une exception est levée par la fonction UseResource(), le flot d'exécution classique est interrompu et le pointeur r renvoyé par ResourceManager::New() n'est donc jamais libéré. Autrement dit, la ressource est définitivement perdue!

De nombreux langages objets (Java, C#, Python, PHP, ...) remédient à ce problème au moyen du mot-clé **finally**, mais C++ n'en fait pas partie. Mais il est possible d'approcher sa philosophie d'utilisation au moyen de la construction suivante par exemple :

```
// Fonction quelconque qui utilise un objet Resource
void UseResource( ResourcePtr );

void Test()
{
    // on autorise 3 instances maximum
    ResourceManager mgr( 3 );

    ResourcePtr r = mgr.New();
    if ( r )
    {
        try
        {
            UseResource( r );
        }
        catch ( ... ) // pseudo bloc finally
        {
            mgr.Delete( r );
            throw; // relancer l'exception
        }
    }
}
```

```
    mgr.Delete( r );  
  }  
}
```

Comme on peut le voir ci-dessus, gérer les ressources de cette manière est extrêmement lourd. De plus, comme le C++ ne collecte pas automatiquement la mémoire au contraire des langages qui proposent le mot-clé **finally**, une telle construction serait à utiliser bien plus souvent que dans ces autres langages, ce qui rendrait le code tout simplement illisible, et inciterait le programmeur à négliger ce genre de "détails". L'utilisation de la construction **try...finally** n'est donc pas une réponse adaptée à un langage comme C++.

Qui plus est, confier ainsi autant de travail au programmeur client pour s'assurer que notre gestion d'objets soit valide est le signe d'une bien piètre robustesse de la part de notre classe ResourceManager. En effet : une utilisation trop basique de cette classe suffit à mettre en péril sa fiabilité, et **c'est au programmeur client de s'assurer de maintenir la stabilité de notre système!** Or, le C++ a la réputation de permettre l'écriture de programmes robustes. Oui, mais comment ?

## L'approche C++ : le RAII

La réponse généralisée de C++ au problème de la gestion des ressources (et pas seulement au cas particulier de la gestion de la mémoire) s'appelle le RAII. Il s'agit de l'acronyme de *Resource Acquisition Is Initialization*, qui peut être traduit par *acquisition des ressources au moment de l'initialisation*. En réalité, il ne faut pas trop s'attacher au sens de cet acronyme, dans la mesure où il ne traduit pas parfaitement le concept qu'il qualifie. La **définition de la FAQ C++** dit : *Il s'agit d'un idiome de programmation consistant à manipuler une ressource quelconque (mémoire, fichier, mutex, connexion à une base de données, ...) au moyen d'une variable locale qui va acquérir cette ressource lors de son initialisation et la libérer lors de sa destruction.*

Cet idiome tire en fait profit d'une des particularités du langage C++ : la présence d'un destructeur de classe déterministe, autrement dit d'une fonction qui est *automatiquement* et *systématiquement* exécutée *dès qu'un objet est détruit*. Le principe du RAII est donc de se servir de cette parité entre le constructeur et le destructeur pour mettre en oeuvre, à la sauce C++ cette fois, le concept présenté plus haut : *C'est celui qui a alloué un objet qui devrait le détruire.*



*Rappel : le seul cas où le destructeur d'un objet n'est pas appelé alors que son constructeur l'a été est celui où ce dernier a levé une exception. En effet, si un constructeur lève une exception, l'objet n'est pas considéré comme construit, et donc son destructeur n'est pas exécuté. Soyez donc vigilant sur ce point avec les ressources que vous allouez dans le constructeur et libérez dans le destructeur : vous risquez une fuite si une exception est levée peu après leur allocation alors que l'on se trouve toujours dans le constructeur.*

Présenté autrement, le RAII consiste à matérialiser une ressource au moyen d'un objet qui acquiert cette ressource lors de son initialisation, et s'assure de sa bonne libération à sa destruction, même si le programmeur n'y a pas pensé. Le constructeur fait donc office de fonction Init() et le destructeur de fonction Free(). L'exemple qui suit met en oeuvre une classe Ressource gérée traditionnellement et une autre classe RessourceRAII dont le design respecte les concepts du RAII.

```
class Ressource
{
public:
    // Allocation des ressources : doit etre appelé
    // en premier, et une seule fois!
    bool Init();
    // Libération des ressources allouées : ne pas
    // oublier de l'appeler une fois terminé!
    void Free();

    void DoSomething();
};

class RessourceRAII
{
public:
    RessourceRAII();
    ~RessourceRAII();

    void DoSomething();
};

void TestRessource()
{
    Ressource r;
    if ( r.Init() )
    {
        try
        {
            r.DoSomething();
        }
    }
}
```

```
        catch ( ... ) // pseudo finally
        {
            r.Free();
            throw; // relancer
        }
        r.Free();
    }
    else
    {
        // heu... que faire, quel est le problème ???
    }
}

void TestRessourceRAII()
{
    RessourceRAII r; // exception levée si échec
    r.DoSomething();
}
```

Cet exemple illustre assez bien la très grande simplification d'utilisation et la robustesse qu'apporte le RAII. Car le principe du RAII veut aussi souvent que, *si l'objet a été créé avec succès, alors il est directement utilisable*. S'il ne parvient pas à s'initialiser, il peut lever une exception pour annuler sa construction. L'idée derrière ce comportement est : *A quoi me servirait un objet qui n'a pas pu se construire et qui n'est pas conséquent pas utilisable ?* Mais bien sûr, il existe des cas particuliers (comme `std::ifstream` par exemple).

Voyons maintenant comment solutionner notre problème initial de gestion des objets Resource en l'abordant sous ce nouvel angle.

## Mise en oeuvre : premier essai

### Présentation de shared\_ptr

La solution la plus simple et la plus économique pour mettre en oeuvre le RAII est certainement de recourir à des **pointeurs intelligents**. Les pointeurs intelligents sont l'exemple par excellence d'application concrète de cet idiome. Ils le marient avec le concept de généricité des templates, ce qui permet une mise en oeuvre du RAII rapide et à moindre frais. En fait, les pointeurs intelligents sont les compagnons indispensables de tout programmeur C++ sérieux.

Il en existe de nombreuses et diverses implémentations. J'ai choisi shared\_ptr, historiquement originaire de la bibliothèque **boost**, et maintenant membre du **TR1**. Cela signifie que shared\_ptr est en train d'être adopté au sein de la norme du langage et sera donc à terme disponible en standard avec la plupart des compilateurs.

En attendant ce jour futur, force est de constater que l'état actuel des choses (Juillet 2007) est différent. Seul GCC propose std::tr1::shared\_ptr en standard, et encore, pas sous toutes les plateformes. Mais les choses ne sont pas si négatives que cela non plus, vu qu'il existe la très mature implémentation boost::shared\_ptr, et qu'il existe aussi **Boost.TR1** qui fournit des fichiers d'en-tête redéfinissant ce type (ainsi que d'autres) au sein de l'espace référentiel std::tr1. En plus clair, il est parfaitement possible d'utiliser le type std::tr1::shared\_ptr tel qu'il est défini dans le *Technical Report 1* avec la plupart des compilateurs à condition d'installer *Boost.TR1*. A ce sujet, vous pouvez lire [Installer et utiliser Boost/Boost.TR1 avec Visual C++](#).

### Premier essai

Reprenons l'exemple initial et adaptons-le pour utiliser std::tr1::shared\_ptr:

#### ResourceManager.h

```
#include <tr1/memory>

class Resource;
typedef std::tr1::shared_ptr<Resource> ResourcePtr;

class ResourceManager
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();

private:
    // Nombre d'instances en cours d'utilisation
    int NbInstances;
    // Nombre maximal d'instances autorisées
    const int MaxNbInstances;
};
```

Vis à vis du programmeur client, notre ResourceManager est on ne peut plus simple et fiable d'utilisation : on alloue une nouvelle ressource via la fonction ResourceManager::New(), on l'utilise sans se poser de question, et une fois qu'on en a terminé avec elle, ResourceManager est automatiquement informé et mis à jour qu'une ressource a été libérée.

Ceci est séduisant, mais il reste encore à coder le *ResourceManager* est automatiquement informé et mis à jour.

La première solution qui vient à l'esprit est de modifier le comportement de Resource pour que la classe informe son gestionnaire qu'elle a été détruite.

#### Resource.h

```
class ResourceManager;

class Resource
{
public:
    // Pointeur NULL = pas rattaché à un ResourceManager
    Resource( ResourceManager* = 0 );

    // Avertit le ResourceManager attaché de sa destruction
    ~Resource();

private:
    // ResourceManager attaché, peut être NULL
    ResourceManager *Manager;
};
```

#### ResourceManager.h

```
#include <tr1/memory>

class Resource;
typedef std::tr1::shared_ptr<Resource> ResourcePtr;

class ResourceManager
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();

private:
    friend class Resource;
    // Appelé par les objets Resources lors de leur destruction
    void ResourceDestroyed( Resource* );

private:
    // Nombre d'instances en cours d'utilisation
    int NbInstances;
    // Nombre maximal d'instances autorisées
    const int MaxNbInstances;
};
```

#### Resource.cpp

```
#include "Resource.h"
#include "ResourceManager.h"

Resource::Resource( ResourceManager *Mgr ):
    Manager( Mgr )
{
}

Resource::~~Resource()
{
    if ( this->Manager )
    {
        this->Manager->ResourceDestroyed( this );
    }
}
```

## ResourceManager.cpp

```
#include "ResourceManager.h"
#include "Resource.h"

ResourceManager::ResourceManager( int Max ) :
    NbInstances( 0 ),
    MaxNbInstances( Max )
{
}

ResourcePtr ResourceManager::New()
{
    if ( NbInstances < MaxNbInstances )
    {
        ResourcePtr r( new Resource() );
        // On incrémente NbInstances _après_ avoir alloué
        // la ressource avec succès, afin de laisser notre
        // Manager dans un état cohérent si une exception
        // est levée lors de la création d'un objet Resource
        ++this->NbInstances;
        return r;
    }
    return ResourcePtr(); // NULL
}

void ResourceManager::ResourceDestroyed( Resource* )
{
    --NbInstances;
}
```

## Nouveaux problèmes soulevés

Cette solution fonctionne, mais pose de nouveaux problèmes :

- La modification de la classe Resource. Ceci n'est pas toujours possible. Il pourrait très bien s'agir d'une classe issue d'une bibliothèque tierce. Dans ce cas, il faudrait alors développer une classe proxy supplémentaire, ce qui est typiquement une contrainte dont on aime se passer.
- La référence croisée entre Resource et ResourceManager. Souvent signe d'une mauvaise conception, les références croisées augmentent le couplage entre les classes concernées, ce qui complique leur maintenance. Et dans ce cas précis, elle pose le problème supplémentaire que, conceptuellement, elle n'a pas lieu d'être. En effet, si l'on se place du point de vue de la classe Resource, on a que faire de savoir si on est géré via un ResourceManager ou pas. Et pourtant, cette information qui ne nous concerne pas vient parasiter notre design, alors qu'elle relève du détail d'implémentation de ResourceManager!
- L'utilisation du mot-clé **friend** est lui aussi souvent un indice d'une mauvaise conception. Le mot-clé en lui-même est une bonne chose puisqu'il permet de *renforcer* l'encapsulation de ResourceManager en rendant sa fonction membre ResourceDestroyed() **private**. Sans lui, nous aurions du la rendre **public**, ce qui aurait été pire. Mais il n'empêche que dans notre cas, il traduit le besoin qu'a la classe que *nous* gérons d'assurer *elle-même* sa propre gestion, alors que c'est l'unique raison d'être de ResourceManager.
- Cette solution est difficilement généralisable, sous forme d'une classe ResourceManager template par exemple.

## Nouvel essai : explorons les possibilités de `shared_ptr`

Notre nouvel objectif est donc de parvenir à appliquer le RAI sans avoir à modifier `Resource` et sans introduire de référence croisée vis à vis de `ResourceManager`. Comment faire ?

Si l'on réfléchit un instant, notre problématique est d'être informé de la destruction d'un objet `Resource` que l'on a nous même alloué un peu plus tôt. Demander à cet objet de nous prévenir quand cela se produit était une première possibilité, mais nous avons vu qu'elle comportait de nombreuses lacunes.

Quand on se retrouve confronté à une situation de dépendance circulaire, la solution pour s'en sortir consiste souvent à introduire un troisième acteur. Et justement, nous disposons d'un troisième acteur : `shared_ptr`. Son utilisation est presque passée inaperçue car discrète, mais il n'en demeure pas moins que nous avons introduit un proxy sur la classe `Resource`. Et l'avantage d'utiliser `shared_ptr` est que cette classe dispose de nombreuses fonctionnalités évoluées.

En particulier, `shared_ptr` dispose d'un constructeur très intéressant, acceptant en second paramètre un *deallocator* :

```
template<class Y, class D>
shared_ptr(Y * p, D d);
```

Le rôle premier d'un *deallocator* est de libérer la mémoire attribuée à l'instance de l'objet dont le pointeur est encapsulé. L'implémentation par défaut effectue un simple appel à `delete`. Mais comme le précise **la documentation de `shared_ptr`**, le concept de *deallocator* ouvre la voie à d'autres types d'utilisations :

*Custom deallocators allow a factory function returning a `shared_ptr` to insulate the user from its memory allocation strategy. Since the deallocator is not part of the type, changing the allocation strategy does not break source or binary compatibility, and does not require a client recompilation. For example, a "no-op" deallocator is useful when returning a `shared_ptr` to a statically allocated object, and other variations allow a `shared_ptr` to be used as a wrapper for another smart pointer, easing interoperability.*

Dans notre cas, nous allons aller un peu plus loin et utiliser le *deallocator* comme fonction callback exécutée au moment de la destruction d'un objet `Resource`, un peu comme un second destructeur en somme!

### ResourceManager.h

```
#include <tr1/memory>

class Resource;
typedef std::tr1::shared_ptr<Resource> ResourcePtr;

class ResourceManager
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();

private:
    class ResourceDeleter; // nested class
    friend ResourceDeleter;

private:
    // Nombre d'instances en cours d'utilisation
    int NbInstances;
    // Nombre maximal d'instances autorisées
```

## ResourceManager.h

```
const int MaxNbInstances;
};
```

## ResourceManager.cpp

```
#include "ResourceManager.h"
#include "Resource.h"

// Classe privée chargée de détruire les objets Resource et
// de mettre à jour le compteur NbInstances de ResourceManager
// (Utilisée comme deleter personnalisé de shared_ptr)
class ResourceManager::ResourceDeleter
{
public:
    ResourceDeleter( ResourceManager *Mgr ):
        Manager( Mgr )
    {
    }

    void operator()( Resource *Res )
    {
        --(this->Manager->NbInstances);
        delete Res;
    }

private:
    ResourceManager *Manager;
};

ResourceManager::ResourceManager( int Max ) :
    NbInstances( 0 ),
    MaxNbInstances( Max )
{
}

ResourcePtr ResourceManager::New()
{
    if ( NbInstances < MaxNbInstances )
    {
        ResourcePtr r( new Resource(), ResourceDeleter( this ) );
        // On incrémente NbInstances _après_ avoir alloué
        // la ressource avec succès, afin de laisser notre
        // Manager dans un état cohérent si une exception
        // est levée lors de la création d'un objet Resource
        ++this->NbInstances;
        return r;
    }
    return ResourcePtr(); // NULL
}
```

Élégant n'est-ce pas ? Il n'est plus nécessaire de modifier Resource (qui peut donc être une classe issue d'un code dont vous n'avez pas le contrôle), et la référence croisée a disparu. Certes, il reste l'utilisation du mot-clé **friend**, mais ce dernier porte cette fois sur une classe imbriquée privée de ResourceManager, et n'est donc pas choquante. ResourceDeleter fait en effet partie de l'implémentation de ResourceManager, et les deux classes évolueront donc ensemble de manière naturelle.

## Version finale

Cependant, il y a des personnes comme moi qui sont vraiment récalcitrantes à utiliser le mot clé **friend**. J'interprète en effet sa présence comme le signe d'un possible défaut dont on essaye de limiter l'étendue, au lieu de le corriger. Et dans ce cas, le défaut est de faire figurer dans l'interface de ResourceManager un détail d'implémentation, appelé ResourceDeleter. C'est un défaut discutable, dans la mesure où il s'agit de l'interface privée de la classe, et que le C++ impose de la rendre visible dans le fichier d'en-tête. Mais en ce qui me concerne, j'aime bien réduire cette interface privée exposée au minimum, dans la mesure du possible bien sûr. Car si l'utilisateur n'a pas le droit ni la possibilité d'utiliser ResourceDeleter, pourquoi l'informer de son existence ?

Et justement, dans ce cas, il est possible de supprimer cette information du fichier d'en-tête en cloisonnant ResourceManager dans un espace de nommage anonyme du fichier d'implémentation :

### ResourceManager.h

```
#include <tr1/memory>
#include <boost/utility.hpp>

class Resource;
typedef std::tr1::shared_ptr<Resource> ResourcePtr;

class ResourceManager : public boost::noncopyable
{
public:
    // Définit le nombre maximal de Resource utilisables simultanément
    ResourceManager( int MaxNbInstances );

    // Renvoie une nouvelle Resource s'il y en a moins que
    // MaxNbInstances en cours d'utilisation, NULL sinon.
    ResourcePtr New();

    // Renvoie le nombre d'objets Resource en cours d'utilisation
    int GetNbInstances() const;

private:
    // Nombre d'instances en cours d'utilisation
    int NbInstances;
    // Nombre maximal d'instances autorisées
    const int MaxNbInstances;
};
```

### ResourceManager.cpp

```
#include "ResourceManager.h"
#include "Resource.h"

namespace // anonyme
{
    // Classe chargée de détruire les objets Resource et
    // de mettre à jour le compteur NbInstances de ResourceManager
    // (Utilisée comme deleter personnalisé de shared_ptr)
    class ResourceDeleter
    {
    public:
        ResourceDeleter( int *MgrNbInstances ) :
            Manager_NbInstances( MgrNbInstances )
        {
        }

        void operator()( Resource *Res )
        {
            --(*this->Manager_NbInstances);
            delete Res;
        }
    };
}
```

## ResourceManager.cpp

```
private:
    // pointeur sur ResourceManager::NbInstances
    int *Manager_NbInstances;
};

ResourceManager::ResourceManager( int Max ) :
    NbInstances( 0 ),
    MaxNbInstances( Max )
{
}

ResourcePtr ResourceManager::New()
{
    if ( NbInstances < MaxNbInstances )
    {
        ResourcePtr r( new Resource(), ResourceDeleter( &this->NbInstances ) );
        // On incrémente NbInstances _après_ avoir alloué
        // la ressource avec succès, afin de laisser notre
        // Manager dans un état cohérent si une exception
        // est levée lors de la création d'un objet Resource
        ++this->NbInstances;
        return r;
    }
    return ResourcePtr(); // NULL
}

int ResourceManager::GetNbInstances() const
{
    return this->NbInstances;
}
```

Pour donner accès à une donnée membre privée de `ResourceManager` sans introduire **friend**, je suis obligé de recourir à un pointeur. Je ne suis en général pas très adepte de ce genre d'acrobatie visant en fait à contourner le contrôle d'accès du compilateur. Mais appliquée à ce cas précis, elle me semble acceptable.

Certains auraient peut-être utilisé une référence à la place d'un pointeur. J'ai choisi un pointeur car je trouve qu'il rend plus explicite le fait que la classe `ResourceDeleter` va modifier l'`int` reçu en paramètre. Mais libre à vous d'utiliser une référence si vous préférez.

Enfin, vous remarquerez l'ajout d'une fonction membre `GetNbInstances()` ainsi que l'inclusion de `boost/utility.hpp` dans le but de faire dériver `ResourceManager` de **`boost::noncopyable`**. Ceci permet de protéger notre gestionnaire d'une copie accidentelle, et d'indiquer de manière plus parlante que cette classe n'est pas faite pour copiée (le moyen habituel d'opérer est de déclarer privés, sans les implémenter, le constructeur par recopie ainsi que l'opérateur d'affectation). Cela permet aussi, avec Visual C++, de faire disparaître l'**avertissement 4512 (niveau 4)** : *'ResourceManager' : l'opérateur d'assignation n'a pas pu être généré.*

## Programme de test

Pour terminer, voici le code d'un programme testant le bon fonctionnement de notre gestionnaire d'objets. Il est à noter que je n'ai pas abordé dans cet article la nécessité de s'assurer que le gestionnaire ne soit pas détruit tant qu'il existe des instances des objets qu'il gère. En effet, si tel était le cas, ces instances tenteraient lors de leur destruction d'accéder à un gestionnaire qui n'existe plus, avec les conséquences fâcheuses que cela implique.

```
#include "ResourceManager.h"
#include "Resource.h"
#include <cassert>
#include <iostream>

int main()
{
    // on autorise 3 instances maximum
    ResourceManager mgr( 3 );

    ResourcePtr r1 = mgr.New();
    ResourcePtr r2 = mgr.New();
    ResourcePtr r3 = mgr.New();
    assert( r1 && r2 && r3 );
    assert( mgr.GetNbInstances() == 3 );

    // New() devrait echouer
    ResourcePtr r4 = mgr.New();
    assert( !r4 );
    assert( mgr.GetNbInstances() == 3 );

    // libérer une instance
    r2 = r1;
    assert( mgr.GetNbInstances() == 2 );

    // maintenant, New() devrait réussir
    {
        ResourcePtr r_local = mgr.New();
        assert( mgr.GetNbInstances() == 3 );
        assert( r_local );
    }
    assert( mgr.GetNbInstances() == 2 );

    // test d'exception safety
    Resource::ThrowExceptionDuringNextConstruction();
    try
    {
        mgr.New();
    }
    catch (...)
    {
    }
    assert( mgr.GetNbInstances() == 2 );

    std::cout << "Test ok!\n";
}
```

Une fois compilé et exécuté, il produit l'affichage suivant :

```
+ Construction de Resource()
+ Construction de Resource()
+ Construction de Resource()
- Destruction de Resource()
+ Construction de Resource()
- Destruction de Resource()
x Construction de Resource() : exception!
Test ok!
```

```
- Destruction de Resource()  
- Destruction de Resource()
```

## Liens, téléchargements, commentaires

L'ensemble du code source, ainsi que les fichiers projets pour le compiler au moyen de Visual C++ 2005, CodeBlocks, Xcode ou GNU Make sont disponibles sous forme d'une archive compressée :

- **ResourceRAII.zip**

Le tout a été compilé :

- Sous Windows, avec Boost 1.34, au moyen des compilateurs Visual C++ 2005, Visual C++ 9 Beta1 "Codename Orcas" et g++ 3.4 (MinGW).
- Sous Linux (Fedora Core 4), avec l'implémentation du TR1 de GCC au moyen de GNU Make et GCC 4.0
- Sous Mac OS X, avec l'implémentation du TR1 de GCC au moyen de Xcode 2.4.1 et GCC 4.0

A noter que, pour permettre une compilation de ce programme avec GCC  $\geq$  4.0 sans que Boost ne soit installé, je n'ai pas fait dériver ResourceManager de boost::noncopyable comme dans l'exemple précédent.

Si vous souhaitez poursuivre votre réflexion sur le sujet, je peux vous conseiller les lectures suivantes :

-  **Generic: Change the Way You Write Exception-Safe Code # Forever** (Andrei Alexandrescu, Petru Marginean)
-  **Smart Pointer Programming Techniques**
-  **Boost.SmartPtr : les pointeurs intelligents de Boost (Miles)**
-  **RAII !** (Emmanuel Deloget)

Si cet article vous a donné envie d'en savoir plus sur Boost, shared\_ptr et le TR1, vous pouvez partir à leur découverte en explorant la [FAQ C++](#) ainsi que mon autre article sur le sujet : [Installer et utiliser Boost/Boost.TR1 avec Visual C++](#).

N'hésitez pas à **laisser un commentaire** au sujet de cet article. Merci pour votre lecture.

