Boost.SmartPtr : les pointeurs intelligents de Boost

par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 01/06/2006

Dernière mise à jour : 31/08/2006

Les pointeurs intelligents ont la cote ces dernières années, principalement grâce à leur capacité à gérer la memoire dynamique de manière transparente. Pas de double effacement, pas de fuite mémoire, les pointeurs intelligents ont la responsabilité du bloc mémoire qu'ils doivent gérer.

Introduction
- Intro-A - Exemple global

I - scoped_ptr<>
- I-A - Fonctions membres
- I-B - Fonctions externes
- I-C - Exemple

II - scoped_array<>
- II-A - Fonctions membres
- II-B - Fonctions externes
- II-C - Exemple

III - shared_ptr<>
- III-A - Fonctions membres
- III-B - Fonctions externes
- III-C - Exemple

IV - shared_array<>
- IV-A - Fonctions membres
- IV-B - Fonctions externes
- IV-C - Exemple

V - weak_ptr<>
- V-A - Fonctions membres
- V-B - Fonctions externes
- V-C - Exemple

VI - intrusive_ptr<>
- VI-A - Fonctions membres
- VI-B - Fonctions externes
- VI-C - Exemple

Conclusion

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Introduction

Les pointeurs intelligents sont la base des techniques actuelles de gestion de la mémoire. Le paradigme classique est de céder la gestion de la mémoire dynamique allouée à un pointeur intelligent. Cette manière de faire peut permettre de désallouer automatiquement la mémoire lors d'une levée d'exception.

Le but de ces pointeurs est simple : ne plus s'occuper de l'effacement au bon moment de la mémoire allouée dynamiquement. C'est une espèce de garbage collector. Dès que le pointeur est effacé, la mémoire associée peut être désallouée si plus personne n'y fait référence. Malheureusement, le seul pointeur intelligent fourni par le standard, **std::auto_ptr<>**, donne soit trop de latitude soit pas assez. C'est pourquoi plusieurs auteurs se sont attaqués à cet Himalaya, et voici une tentative d'ascension.

Cette bibliothèque est templatée et n'utilise pas de bibliothèque compilée .lib/.a ou .dll/.so. Toutes les **classes** et les **fonctions** vivent dans le namespace **boost**.

Intro-A - Exemple global

Pourquoi utiliser les pointeurs intelligents alors que tout marche bien sans ? Voici un petit exemple pour simplifier la tâche et l'écriture du destructeur. Prenons donc un gestionnaire de textures. On connaît le nombre de textures à gérer au départ, on doit gérer leur construction et à priori leur destruction.

Maintenant, on va utiliser **boost::scoped_array<>** ainsi que **boost::shared_ptr<>**.

header

```
#include <boost/scoped_array.hpp>
#include <boost/shared_ptr.hpp>

class Gestionnaire
{
  boost::scoped_array<boost::shared_ptr<Texture> > textures;
};
```

On obtient donc un comportement équivalent à un tableau de **Texture*** de dimension fixe, on l'appelle de manière semblable :

```
(*textures[i]).faireQqch();
```

Maintenant, comment construire et remplir ce tableau ? Au départ, on ne connaît pas forcément la taille du tableau. Si on la connaît, on peut initialiser directement **textures**, mais dans le cas général, on ne peut pas.

Code réduit du constructeur

```
boost::shared_ptr<Texture> chargeTexture(unsigned int i)
{
  // Charge une texture et la renvoie sous la forme d'un pointeur intelligent
}

Gestionnaire::Gestionnaire(...)
{
  unsigned int texturesSize = chercherTailleTextures();
  textures.reset(new boost::shared_ptr<Texture>[texturesSize]);
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Code réduit du constructeur

```
    for(unsigned int nouvelleTexture = 0; nouvelleTexture < chercherTailleTextures;
   ++nouvelleTexture)
    {
      textures[nouvelleTexture] = chargeTexture(nouvelleTexture);
    }
  }
```

Maintenant que les textures existent, on peut les renvoyer directement lorsqu'on en a besoin :

Récupération d'une texture

```
  boost::shared_ptr<Texture> Gestionnaire::getTexture(unsigned int i)
  {
    return textures[i];
  }
```

Comme vous le pouvez remarquer, nous n'avons pas défini de destructeur et la destruction de ce tableau. La raison est simple : lorsque le gestionnaire va se détruire, il n'y aura plus de référence sur les textures. Donc la mémoire occupée par ces dernières se libérera "toute seule". C'est là un des avantages de Boost.Smart_ptr.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

I - scoped_ptr<>

Le pointeur intelligent **scoped_ptr<>** est une version plus stricte des **std::auto_ptr<>** dans le sens où **scoped_ptr<>** ne peut être copié. Donc la mémoire gérée par un **scoped_ptr<>** sera détruite lors de la destruction du pointeur. Aucune des fonctions associée à cette classe ne lèvera d'exception.

I-A - Fonctions membres

boost/scoped_ptr.hpp

```
template<class T> class scoped_ptr
{
public:
    typedef T element_type;
    explicit scoped_ptr(T * p = 0);
    explicit scoped_ptr(std::auto_ptr<T> p);
    ~scoped_ptr();
    void reset(T * p = 0);
    T & operator*() const;
    T * operator->() const;
    T * get() const;
    operator bool () const;
    bool operator! () const;
    void swap(scoped_ptr & b);
};
```

Tout d'abord un **typedef** est défini pour accéder au type de données depuis d'autres fonctions ou classes, principalement les fonctions ou classes **template**s.

En ce qui concerne les fonctions définies, tout d'abord 2 constructeurs explicites, l'un prenant un pointeur, l'autre prenant un **std::auto_ptr<>** qui sera réinitialisé. Le destructeur détruira naturellement l'espace mémoire gérée.

Une fonction **reset** permet de gérer un nouvel espace mémoire tout en détruisant l'espace actuel. Trois fonctions permettent de récupérer le pointeur donné en paramètre - sans céder la gestion ! -, **operator***, **operator->**, **get** qui permettent de récupérer l'objet ou un pointeur vers l'objet. En revanche, les 2 opérateurs effectuent une assertion sur la non nullité du pointeur... 2 fonctions **operator bool** et **operator!** permettent d'utiliser le pointeur intelligent comme un pointeur, à savoir tester sa nullité. Enfin, la fonction **swap** échange les pointeurs entre 2 pointeurs intelligents, beaucoup de fonctions utilisent la fonction **swap**.

I-B - Fonctions externes

```
template<class T> inline void swap(scoped_ptr<T> & a, scoped_ptr<T> & b);
```

Une seule fonction externe existe, elle appelle en fait la fonction membre swap. C'est une surcharge de la fonction swap<> accessible facilement grâce à la résolution des noms, ce qu'on appelle **Koenig Lookup**.

I-C - Exemple

Utiliser ce pointeur intelligent pour gérer l'implémentation privée d'une classe non copiable - qui a dit un singleton ?? -

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
#include <boost/scoped_ptr.hpp>
class exemple
{
public:
 exemple()
 :pointeur(new exempleImplentation)
 {}

 ~exemple()
 {
 }

 utiliser()
 {
  pointeur->effectuerAction();
 }
private:
 boost::scoped_ptr<exempleImplentation> pointeur;
}
```

Utiliser le pointeur dans une fonction qui récupère un pointeur vers une classe parente.

```
#include <boost/scoped_ptr.hpp>
double optimiser(const parameterType& parameters)
{
 boost::scoped_ptr<baseClass> optimizer = OptimizerFactory(parameters);
 return optimizer();
}
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

II - scoped_array<>

scoped_array<> est la version pour les tableaux de **scoped_ptr<>**. Les deux différences sont l'appel à **delete[]** au lieu de **delete** et l'existence d'une fonction **operator[]**.

II-A - Fonctions membres

boost/scoped_array.hpp

```
template<class T> class scoped_array
{
public:
    typedef T element_type;
    explicit scoped_array(T * p = 0);
    ~scoped_array();
    void reset(T * p = 0);
    T & operator[](std::ptrdiff_t i) const;
    T * get() const;
    operator bool () const;
    bool operator! () const;
    void swap(scoped_array & b);
};
```

operator[] permet en fait d'accéder au tableau d'éléments de la même manière que pour un pointeur vers un tableau classique.

II-B - Fonctions externes

```
template<class T> inline void swap(scoped_array<T> & a, scoped_array<T> & b);
```

II-C - Exemple

*Le principal intérêt de **scoped_array<>** sur un **std::vector<>** est de fixer la taille d'un tableau. Dans les autres cas, un **std::vector<>** est meilleur.*

```
#include <boost/scoped_array.hpp>

class utiliseUnTableauFixe
{
public:
 utiliseUnTableauFixe(unsigned int tailleFixe)
 :tableauFixe(new int(tailleFixe))
 {
 }
private:
 boost::scoped_array<int> tableauFixe;
}
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

III - shared_ptr<>

Ce pointeur est le plus important. En effet, il implémente un compteur de références, on peut définir un objet pour détruire le pointeur, en fait, on peut gérer comme on veut presque n'importe quoi !

III-A - Fonctions membres

Il y a beaucoup de fonctions donc on va séparer les membres en plusieurs blocs.

boost/shared_ptr.hpp

```
template<class T> class shared_ptr
{
public:
    typedef T element_type;
    typedef T value_type;
    typedef T * pointer;
    typedef typename detail::shared_ptr_traits<T>::reference reference;
```

Quelques typedef utiles, comme le type du contenu - deux fois -, le type du pointeur et un type un peu compliqué qui n'est en fait que la référence du type contenu, ceci afin de pouvoir renvoyer une référence vers l'objet lors des appels à **operator*** par exemple. On remarque donc que **shared_ptr<>** a un design légèrement meilleur que **scoped_ptr<>** à ce niveau.

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

    template<class Y>
    explicit shared_ptr( Y * p );

    template<class Y>
    explicit shared_ptr(weak_ptr<Y> const & r);

    template<class Y>
    shared_ptr(shared_ptr<Y> const & r);

    template<class Y>
    shared_ptr(shared_ptr<Y> const & r, detail::static_cast_tag);

    template<class Y>
    shared_ptr(shared_ptr<Y> const & r, detail::const_cast_tag);

    template<class Y>
    shared_ptr(shared_ptr<Y> const & r, detail::dynamic_cast_tag);

    template<class Y>
    shared_ptr(shared_ptr<Y> const & r, detail::polymorphic_cast_tag);

    template<class Y>
    explicit shared_ptr(std::auto_ptr<Y> & r);
```

Il existe une sacré quantité de constructeurs qu'on peut utiliser. Outre le constructeur qui ne prend aucun argument, un constructeur prenant en agurment... un pointeur - on remarquera que le pointeur à stocker n'est pas forcément de même type que le pointeur qui sera effectivement stocké -, on peut construire un **shared_ptr<>** à partir d'un **weak_ptr<>** - plus à propos de ce pointeur intelligent dans la dernière partie -, à partir d'un **shared_ptr<Y>** si on peut transformer un **Y*** en **T*** de manière implicite, de manière explicite à l'aide d'un **static_cast<>**, d'un **const_cast<>**, d'un **dynamic_cast<>** ou d'un **polymorphic_cast<>** - un cast proposé par Boost -. Enfin, on peut construire un **shared_ptr<>** à partir d'un **auto_ptr<>**, tout comme c'était le cas pour un **scoped_ptr<>**.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Un constructeur qui mérite toute notre attention est le deuxième, **template<class Y, class D> shared_ptr(Y * p, D d);**, car outre un pointeur, il prend en argument un objet **d** de type **D** qui sera chargé de détruire l'objet avec l'instruction **d(p)** . Le seul impératif est que le constructeur par copie de **D** ne doit pas lever d'exception.

Seul le constructeur par défaut et le constructeur par copie ne lève pas d'exception.

```
template<class Y, class D> shared_ptr(Y * p, D d);
shared_ptr & operator=(shared_ptr const & r);

template<class Y>
shared_ptr & operator=(shared_ptr<Y> const & r);

template<class Y>
shared_ptr & operator=(std::auto_ptr<Y> & r);
```

Les opérateurs d'affectation peuvent copier un std::auto_ptr<>, ou n'importe quel autre shared_ptr<Y>, après la copie, le compteur est commun aux 2 shared_ptr<>. Les opérateurs prenant un shared_ptr<> ne lèvent pas d'exception, puisqu'aucun nouveau compteur n'est créé.

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

On peut réinitialiser le pointeur intelligent avec rien, un nouveau pointeur ou un pointeur avec une classe d'effacement.

```
reference operator* () const;
T * operator-> () const;
T * get() const;
```

L'opérateur de déréférencement * retourne une référence sur l'objet pointé, l'opérateur -> retourne le pointeur vers l'objet, tout comme la fonction **get**. Aucune de ces fonctions ne lève d'exception, en revanche les 2 premières utilisent une assertion sur la non nullité du pointeur.

```
operator bool () const;
bool operator! () const;
```

Les opérateurs classiques testant la nullité du pointeur sont fournis. Ces fonctions ne lèvent pas d'exception.

```
bool unique() const;
long use_count() const;
```

Ces fonctions membes sont spécifiques au pointeur shared_ptr<>. La première retourne **true** si le pointeur est le seul à faire référence à l'espace mémoire. **count** permet de retourner le nombre de pointeurs référençant l'espace mémoire. Ces fonctions ne lèvent pas d'exception.

```
void swap(shared_ptr<T> & other);
```

Comme pour **scoped_ptr<>**, une fonction **swap** permet d'échanger l'espace mémoire référencé. Cette fonction ne lève pas d'exception.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    template<class Y> bool _internal_less(shared_ptr<Y> const & rhs) const;
    void * _internal_get_deleter(std::type_info const & ti) const;
};
```

Ces fonctions ne sont pas à utiliser dans les cas normaux.

III-B - Fonctions externes

```
template<class T, class U> inline bool operator==(shared_ptr<T> const & a, shared_ptr<U> const &
 b);
template<class T, class U> inline bool operator!=(shared_ptr<T> const & a, shared_ptr<U> const &
 b);
template<class T, class U> inline bool operator<(shared_ptr<T> const & a, shared_ptr<U> const & b);
```

Ces fonctions externes permettent de comparer deux shared_ptr<>, à savoir s'ils pointent vers le même espace mémoire, s'ils pointent vers des espaces différents ou si les pointeurs ont un certain classement en mémoire.

```
template<class T> inline void swap(shared_ptr<T> & a, shared_ptr<T> & b);
```

Ceci est la fonction externe **swap**, qui fonctionne comme la fonction membre **swap**.

```
template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> shared_static_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> shared_dynamic_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> shared_polymorphic_cast(shared_ptr<U> const & r);
template<class T, class U> shared_ptr<T> shared_polymorphic_downcast(shared_ptr<U> const & r);
```

Les 4 dernières fonctions de cette série ne sont pas à utiliser, elles sont présentes par compatiblité avec le passé. Les 3 premières retournent en fait un nouveau **shared_ptr<>** avec le constructeur par cast vu précédemment.

```
template<class T> inline T * get_pointer(shared_ptr<T> const & p);
```

Cette fonction externe appelle simplement la fonction membre **get**.

```
template<class Y> std::ostream & operator<< (std::ostream & os, shared_ptr<Y> const & p);
template<class D, class T> D * get_deleter(shared_ptr<T> const & p);
```

La première fonction permet d'envoyer sur un flux de sortie le pointeur interne tandis que la deuxième retourne un pointeur vers l'objet chargé d'effacer la classe.

III-C - Exemple

*On peut utiliser la fonctionnalité unique de cette classe qui prévoit un objet d'effacement pour surveiller les effacements et déclarer les destructeurs privés ou protégés. Grâce à cela, on peut créer et effacer dans une bibliothèque dynamique en interdisant de le faire à l'extérieur de la classe, cela peut servir lors de vérifications de fuites mémoire. Cela transforme le **shared_ptr<>** en une sorte de **shared_ressource**.*

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
#include <boost/shared_ptr.hpp>
class exemple
{
 struct deleter
 {
  void operator()(exemple* p)
  {
   delete p;
  }
 }
 friend struct deleter;
public:
 static boost::shared_ptr<exemple> createExemple()
 {
  return boost::shared_ptr<exemple>(new exemple(), exemple::deleter());
 }
protected:
 ~exemple()
 {
 }
}
```

*A force, on a tout son code qui prend en argument des **shared_ptr<>**. Si une fonction externe à une classe doit être appelée par une méthode interne et qu'elle prend un pointeur intelligent en argument, on peut donner à une instance d'une classe la possibilité de retourner un **shared_ptr<>** pointant vers lui-même. En fait, on utilise la classe **weak_ptr<>** pour pouvoir créer plusieurs **shared_ptr<>** avec le même compteur, mais ce **weak_ptr<>** est caché dans la classe grâce à la classe **enable_shared_from_this<>**.*

```
#include <boost/shared_ptr.hpp>
#include <boost/enable_shared_from_this.hpp>

class exemple : public enable_shared_from_this<exemple>
{
public:
 void callDoSomething()
 {
  doSomething(shared_from_this());
 }
}

void doSomething(boost::shared_ptr<exemple>)
{
 // Faire quelque chose avec ce pointeur intelligent
}
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

IV - shared_array<>

La version pour tableau de shared_ptr<>, cette classe ne définissant vaiment que **operator[]** en plus pour simuler un comportement de tableau.

IV-A - Fonctions membres

boost/shared_array.hpp

```
template<class T> class shared_array
{
public:
    typedef T element_type;
    explicit shared_array(T * p = 0);
    template<class D> shared_array(T * p, D d);
    void reset(T * p = 0);
    template <class D> void reset(T * p, D d);
    T & operator[] (std::ptrdiff_t i) const;
    T * get() const;
    operator bool () const;
    bool operator! () const;
    bool unique() const;
    long use_count() const;
    void swap(shared_array<T> & other);
};
```

Comme indiqué, à peu de choses près, seul l'opérateur **[]** est rajouté dans la liste des fonctions membres. On voit que les fonctions de conversion n'existent plus. Dommage.

IV-B - Fonctions externes

```
template<class T> inline bool operator==(shared_array<T> const & a, shared_array<T> const & b)
template<class T> inline bool operator!=(shared_array<T> const & a, shared_array<T> const & b)
template<class T> inline bool operator<(shared_array<T> const & a, shared_array<T> const & b)
template<class T> void swap(shared_array<T> & a, shared_array<T> & b);
```

Ici aussi, les fonctions de conversion ne sont pas disponibles, ainsi que les fonction **get_deleter**, **get_pointer** et **operator<<**.

IV-C - Exemple

On peut se servir de la capacité de comptage pour faire des copies légères de vos classes lorsque les copies sont fréquentes.

```
#include <boost/shared_array.hpp>
class matrix
{
 boost::shared_array<double> innerMatrix;
public:
 matrix(unsigned int height, unsigned int width)
 :innerMatrix(new double[height * width])
 {
 }

 matrix(const matrix& other)
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    :innerMatrix(other.innerMatrix)
    {
    }
  }

  const matrix operator+(const matrix lhs, const matrix rhs)
  {
   matrix added(...);
   // Code de l'addition
   return added;
  }
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

V - weak_ptr<>

weak_ptr<> est une classe d'aide à **shared_ptr<>**. Elle retient le pointeur à gérer mais n'endosse aucune reponsabilité à son sujet, le compteur associé ne bougeant pas. Dans l'exemple **III-C**, si on utilisait un **shared_ptr<>**, la classe ne serait jamais désallouée puisqu'un pointeur à responsabilité existerait toujours, tant que l'instance existe, ce qui n'est pas possible, ... Cela donne un cycle que **weak_ptr<>** permet de rompre automatiquement. C'est donc une classe indispensable.

V-A - Fonctions membres

boost/weak_ptr.hpp

```
template<class T> class weak_ptr
{
public:
    typedef T element_type;
    weak_ptr();

    template<class Y>
    weak_ptr(weak_ptr<Y> const & r);
    template<class Y>
    weak_ptr(shared_ptr<Y> const & r);
    template<class Y>
    weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y>
    weak_ptr & operator=(shared_ptr<Y> const & r);

    shared_ptr<T> lock();
    long use_count() const;
    bool expired() const;
    void reset()
    void swap(this_type & other);
    void _internal_assign(T * px2, detail::shared_count const & pn2);
    template<class Y> bool _internal_less(weak_ptr<Y> const & rhs) const;
};
```

Outres les classiques constructeurs et fonctions - maintenant, vous devez savoir à quoi ils servent à l'aide des précédents pointeurs intelligents -, une fonction **lock** permet de créer un **shared_ptr<>**, la fonction **use_count** retourne le nombre de pointeurs à responsabilité utilisés, **expired** retourne vrai si le compte des pointeurs à responsabilité est nul et **swap** échange les contenus des pointeurs. Les 2 dernières fonctions sont à nouveau des fonctions internes.

V-B - Fonctions externes

```
template<class T, class U> inline bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);
template<class T> void swap(weak_ptr<T> & a, weak_ptr<T> & b);
template<class T> shared_ptr<T> make_shared(weak_ptr<T> const & r);
```

Seul un opérateur de comparaison existe, contrairement aux autres pointeurs. La fonction **swap** est toujours identique aux autres tandis que la fonction **make_shared** fait appel à **lock** pour créer un nouveau pointeur à responsabilité.

V-C - Exemple

Un exemple pour cette partie se trouve être l'exemple du **II-C**, le pointeur **weak_ptr<>** étant caché dans la classe.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

VI - intrusive_ptr<>

Certaines classes proposent un compteur interne ou parfois l'allocation d'un compteur n'est pas une solution à envisager. Pour cela, la classe **intrusive_ptr<>** délègue l'incrémentation et la décrémentation du compteur à l'instance pointée.

VI-A - Fonctions membres

boost/intrusive_ptr.hpp

```
template<class T> class intrusive_ptr
{
public:
    typedef T element_type;
    intrusive_ptr();
    intrusive_ptr(T * p, bool add_ref = true);
    template<class U> intrusive_ptr(intrusive_ptr<U> const & rhs);
    intrusive_ptr(intrusive_ptr const & rhs);
    ~intrusive_ptr();
    template<class U> intrusive_ptr & operator=(intrusive_ptr<U> const & rhs);
    intrusive_ptr & operator=(intrusive_ptr const & rhs);
    intrusive_ptr & operator=(T * rhs);
    T * get() const;
    T & operator*() const;
    T * operator->() const;
    operator bool () const;
    bool operator! () const;
    void swap(intrusive_ptr & rhs);
};
```

Les constructeurs et destructeurs - remarquez que le destructeur est explicite ici, contrairement à de nombreux cas - appellent les fonctions **intrusive_ptr_add_ref** et **intrusive_ptr_release**. Ces fonctions sont à déclarer dans le même espace de nom que les classes à "protéger". **intrusive_ptr_release** est aussi chargé de la destruction de l'objet pointé. Les autres fonctions sont classiques.

VI-B - Fonctions externes

```
template<class T, class U> inline bool operator==(intrusive_ptr<T> const & a, intrusive_ptr<U>
 const & b);
template<class T, class U> inline bool operator!=(intrusive_ptr<T> const & a, intrusive_ptr<U>
 const & b);
template<class T> inline bool operator==(intrusive_ptr<T> const & a, T * b);
template<class T> inline bool operator!=(intrusive_ptr<T> const & a, T * b);
template<class T> inline bool operator==(T * a, intrusive_ptr<T> const & b);
template<class T> inline bool operator!=(T * a, intrusive_ptr<T> const & b);
```

Comme une instance est en fait un pointeur intelligent en lui-même puisqu'il contient le compteur, on peut comparer des pointeurs avec des **intrusive_ptr<>**, d'où la définition des opérateurs de comparaison qui n'existaient pas pour les autres pointeurs intelligents.

VI-C - Exemple

On peut reprendre le même exemple que III-C sur les fonctions attendant un pointeur intelligent comme argument.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
#include <boost/intrusive_ptr.hpp>
namespace monTest
{
 class Compteur
 {
  unsigned int references;
 public:
  Compteur():references(0)
  {}
  virtual ~Compteur() {}

  friend void intrusive_ptr_add_ref(Compteur* p)
  {
   ++p->references;
  }
  friend void intrusive_ptr_release(Compteur* p)
  {
   if(--p->references == 0)
    delete p;
  }
 protected:
  Compteur& operator=(const Compteur&)
  {
   return *this;
  }
 private:
  Compteur(const Compteur& other);
 };

 class ClassTest : public Compteur
 {
 //
  void callDoSomething()
  {
   /// Cet appel va transformer this en intrusive_ptr<> grâce à un appel à un constructeur
 implicite
   doSomething(this);
  }
 };

 void doSomething(boost::intrusive_ptr<ClassTest> classTest)
 {
   //
 }
}
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Conclusion

Nous voici à la fin de l'introduction aux pointeurs intelligents de Boost. Leur utilisation devrait être systématique dès qu'il y a des allocations dynamiques afin de limiter les fuites mémoire - même plus besoin de penser au **delete** -.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.