

# Pointeurs intelligents

par Loïc Joly

Date de publication : 4 août 2008

Dernière mise à jour : 5 août 2008

Ce document présente un outil indispensable à l'écriture de code correct en C++ : Les pointeurs intelligents. Après une présentation du problème que ces pointeurs aident à résoudre, il décortique comment il est possible de créer un tel pointeur, et enfin présente les pointeurs intelligents les plus courants, et comment les utiliser.

I - Introduction.....	3
I-A - À qui est destiné ce document.....	3
I-B - Historique.....	3
II - Les pointeurs intelligents, pourquoi, comment ?.....	3
II-A - Problèmes des pointeurs nus.....	3
II-A-1 - Ne pas libérer de la mémoire allouée dynamiquement.....	4
II-A-2 - Libérer plusieurs fois de la mémoire allouée dynamiquement.....	5
II-A-3 - Accéder à la valeur pointée par un pointeur invalide.....	5
II-A-4 - Un pointeur nu est peu explicite.....	5
II-B - Qu'est-ce qu'un pointeur intelligent.....	5
II-C - Notre propre pointeur intelligent : ValuePtr.....	6
II-C-1 - Problématique.....	6
II-C-2 - Structure de base.....	7
II-C-3 - Ressembler à un pointeur.....	7
II-C-4 - Implémenter notre sémantique.....	8
II-C-5 - Flexibilité pour le type pointé.....	9
II-C-6 - Ressembler à un pointeur : La suite.....	11
II-C-7 - Conclusion.....	11
III - Les pointeurs intelligents standards.....	12
III-A - shared_ptr.....	12
III-A-1 - Utilisation de base.....	12
III-A-2 - Compatibilité entre shared_ptr.....	13
III-A-3 - Risque de fuite mémoire : Initialisation.....	13
III-A-4 - Risque de fuite mémoire : Cycle.....	14
III-A-5 - Obtenir un shared_ptr à partir de this.....	14
III-A-6 - Spécifier une fonction de désallocation.....	15
III-A-7 - Aliasing.....	16
III-A-8 - Inconvénients des shared_ptr.....	16
III-B - weak_ptr.....	17
III-B-1 - Utilisation de base.....	17
III-B-2 - Ordre des dépendances.....	18
III-C - unique_ptr.....	19
III-C-1 - Utilisation de base.....	19
III-C-2 - Avantage par rapport à un shared_ptr.....	19
IV - Conclusion.....	19

## I - Introduction

### I-A - À qui est destiné ce document

Ce document s'adresse à un public ayant déjà des notions de base en C++, en particulier les notions de pointeurs, d'exceptions et l'utilisation de templates sont supposées connues. Mais les notions présentées ne sont pas avancées, et les connaître est utile à toute personne désirant écrire un programme en C++ robuste.

Ce document ne reprend pas toutes les fonctions des pointeurs intelligents, mais seulement celles qui me semblaient les plus courantes. Une bonne référence sur le sujet peut donc être un complément intéressant à cet article.

Il y a d'autres ressources sur ce sujet sur le site, par exemple [Boost.SmartPtr : les pointeurs intelligents de Boost](#) de Matthieu Brucher ou [Gérer ses ressources de manière robuste en C++](#) par Aurélien Regat-Barrel

Les sections « Historique » et « Notre propre pointeur intelligent : ValuePtr » ne sont pas indispensables à la compréhension du reste de l'article.

### I-B - Historique

La notion de pointeur intelligent existe depuis un certain temps dans le langage. C++98 (le standard actuellement en vigueur, 98 signifiant qu'il a été publié en 1998) définit même dans sa bibliothèque un pointeur intelligent, nommé `auto_ptr`. Cette première tentative n'est pas sans problèmes, et il est généralement déconseillé d'utiliser cette classe.

Après la sortie du standard, un certain nombre de personnes intéressées par l'évolution du langage se sont rassemblées pour fonder ce qui allait devenir **boost**, une collections de bibliothèques d'intérêt général pour les développeurs C++, ayant pour vocation de servir de terrain d'expérimentation en grande nature pour le prochain standard. La bibliothèque qui a probablement le plus servi à assurer le succès de cette initiative est la bibliothèque `smart_ptr`, qui contient un certain nombre de pointeurs intelligents, parmi lesquels `boost::shared_ptr` et `boost::weak_ptr`, mais aussi `boost::scoped_ptr`.

Par la suite, le comité de normalisation a publié, sous le nom TR1, une liste de bibliothèques qui ferait partie du prochain standard C++. On peut voir le TR1 comme une version 1.1 du standard. A l'heure où cet article est écrit, la plupart des compilateurs commencent à proposer une implémentation de TR1. Ce dernier contient deux pointeurs intelligents, `std::tr1::shared_ptr` et `std::tr1::weak_ptr`, qui reprennent les versions issues de boost.

Enfin, le prochain standard, nommé temporairement C++0x, contiendra aussi des pointeurs intelligents. Même si la situation reste en théorie susceptible d'évoluer, cette partie du standard est assez stabilisée, et il est prévu qu'il contienne `std::shared_ptr`, `std::weak_ptr` et `std::unique_ptr` (qui remplace `std::auto_ptr`, conservé uniquement pour préserver le code existant, et `boost::scoped_ptr`).

On peut avoir un peu peur de se perdre entre tous ces pointeurs intelligents, d'autant que chaque version a des différences subtiles avec les autres. Il n'empêche que ces versions partagent aussi beaucoup de choses en commun, et qu'une sorte de compatibilité ascendante existe. Si un choix est à réaliser, ce document se placera dans l'optique C++0x, mais à part ce qui concerne `unique_ptr`, il devrait être directement utilisable si vous utilisez les pointeurs intelligents de boost ou de TR1.

## II - Les pointeurs intelligents, pourquoi, comment ?

### II-A - Problèmes des pointeurs nus

Avant de voir les différents types de pointeurs intelligents, il est utile de revenir sur les pointeurs classiques du C++, que je nommerai pointeurs nus par opposition aux pointeurs intelligents. Ces pointeurs présentent un certain

nombre de problèmes qui en rendent l'usage difficile et risqué, les différents pointeurs intelligents ayant pour objectif de corriger ces problèmes.

Il y a globalement trois risques d'erreur, et un problème de clarté, avec les pointeurs nus.

## II-A-1 - Ne pas libérer de la mémoire allouée dynamiquement

A partir du moment où on alloue dynamiquement de la mémoire, il faut la désallouer quand on a fini de s'en servir. Sinon, la mémoire reste réservée, et au fur et à mesure que le temps passe, notre programme va "manger" la mémoire disponible sur la machine, conduisant à des baisses de performance et à des explosions, c'est ce qu'on nomme une fuite mémoire.

Cette contrainte pose deux problèmes : Elle oblige déjà à une grande discipline de codage, afin de ne pas oublier un delete quelque part, et en plus, elle est très difficile à mettre en œuvre correctement.

```
int g();

int f()
{
    int *i = new int(42);
    g();
    delete i;
}

class A
{
public:
    A() : i(new int(314)), j(new int(42)) {}
    ~A()
    {
        delete j;
        delete i;
    }
private:
    // Fonctions déclarées privées pour empêcher le compilateur de générer
    // un constructeur de recopie et un opérateur d'affectation par défaut.
    A(A const &);
    A& operator= (A const &);

    int *i;
    int *j;
};
```

Dans ces deux cas, on peut penser avoir correctement fait le travail de désallocation. Mais ce n'est pas le cas. Si par exemple la fonction g lance une exception, le delete i ne sera jamais appelé.

Dans le second cas, même problème, si le new int(42) échoue, par manque de mémoire, une exception est lancée. Cette exception fait que la classe A sera considérée comme n'ayant jamais été construite, et on ne passera donc pas dans son destructeur. La mémoire allouée pour i ne sera donc jamais libérée.

Le plus difficile, c'est que tant qu'une exception n'est pas lancée à un moment bien précis, le code va tourner sans problèmes. Et même si l'exception est lancée, une fuite mémoire n'est pas forcément très visible et peut donc passer inaperçue pendant longtemps. Ce genre d'erreur risque donc fortement de ne pas être détectée pendant la phase de tests, jusqu'au jour où elle deviendra très gênante, quand le code sera déployé...

La solution classique à ce problème en C++ est le **RAII**. Dans les deux exemples cités, le RAI1 classique résoudrait le problème. Par contre, le RAI1 lie la durée de vie des objets avec la portée de la variable locale. C'est souvent un peu simpliste, puisque justement, si on a utilisé de l'allocation dynamique, c'est que l'on souhaite décorrélérer la durée de vie de l'objet avec la portée où il est déclaré. Les pointeurs intelligents peuvent être vus comme une extension du RAI1 ayant une gestion plus fine de la durée de vie de l'objet pointée. Parmi ceux-ci, boost::scoped\_ptr représente ni plus ni moins que la mise en place de RAI1 pour gérer de la mémoire.

## II-A-2 - Libérer plusieurs fois de la mémoire allouée dynamiquement

Le code suivant possède des erreurs :

```
int *i = new int(14);
delete i;
delete i; // Erreur : Pas deux désallocations
int *j;
delete j; // Erreur : Désallocation d'une zone non allouée
int *k = NULL;
delete k; // Ok : On peut faire delete NULL sans problèmes
```

La règle est qu'on ne peut désallouer qu'une zone qui est allouée (ou NULL). Ce cas se présente moins souvent que le cas précédent.

## II-A-3 - Accéder à la valeur pointée par un pointeur invalide

Le dernier type de problème pouvant arriver, c'est de tenter de déréférencer un pointeur invalide. Ça arrive le plus souvent quand plusieurs pointeurs pointent sur une même zone de mémoire, que l'on fait delete sur l'un des pointeurs, mais que les autres pointeurs ne sont pas au courant et continuent de tenter d'y accéder.

```
int *i = new int(42);
int *j = i;
delete i;
*j = 2;
```

## II-A-4 - Un pointeur nu est peu explicite

Prenons par exemple une fonction :

```
MaStructure const *getInformation();
```

Il est évident que cette fonction va retourner un pointeur sur une zone de mémoire contenant les informations que l'on désire. Mais qu'est-on sensé faire de ce pointeur quand on a fini d'utiliser ces informations ? Rien ? L'effacer avec delete ? L'effacer avec free ? Autre chose ? On n'en sait rien a priori, toutes ces possibilités ayant des cas d'application concrets. Certes, la documentation associée à la fonction getInformation, si elle est écrite correctement, doit nous fournir la réponse, mais il serait plus agréable de ne pas avoir à se reposer sur une documentation pour ce genre de choses.

## II-B - Qu'est-ce qu'un pointeur intelligent

Si les pointeurs nus présentent tous ces problèmes, ne serait-il pas possible de les remplacer par autre chose qui ne les ait pas ? C'est de cette idée que sont nés les pointeurs intelligents. Tout comme il existe différents scénarios types d'utilisation des pointeurs, il existe différents pointeurs intelligents, chacun répondant à un besoin particulier.

Un pointeur intelligent est une classe qui encapsule la notion de pointeur, tout en offrant une sémantique de plus haut niveau. On a vu que les principaux problèmes liés aux pointeurs nus étaient liés à la durée de vie des objets pointés, et au lien entre celle-ci et la durée de vie des pointeurs eux-mêmes. C'est donc principalement autour des opérations liées à la durée de vie des pointeurs (création, copie, destruction...) qu'on ajoutera de l'intelligence, le reste de la classe étant juste là pour donner un accès agréable au pointeur sous-jacent.

Si l'on voulait donner une définition d'un pointeur intelligent, on pourrait donc dire qu'il s'agit d'une classe que l'on utilise presque comme un pointeur, mais qui possède un mécanisme gérant la durée de vie des objets pointés.

Dans du code C++ moderne, utilisant ce genre de techniques, on se retrouve donc à avoir très peu de delete, la mémoire étant presque systématiquement gérée automatiquement.

Parmi les usages classiques des pointeurs, il y en a deux qui ne sont généralement pas couverts par les pointeurs intelligents :

- Les pointeurs sur des tableaux de caractères, ayant pour but de représenter des chaînes de caractères, pour lesquels les classes `std::string` et `std::wstring` apportent les mêmes avantages, tout en offrant en plus un certain nombre d'opérations spécifiques à des chaînes de caractères
- Les pointeurs ayant pour but de gérer des tableaux de taille dynamique, rôle géré directement par les conteneurs de la bibliothèque standard, et en particulier `std::vector`.

Il y a aussi certains styles de programmation pour lesquels des pointeurs intelligents sont moins indispensables, en particulier quand chaque objet appartient de manière unique et non équivoque à un gestionnaire s'occupant de sa durée de vie, et que par design, on sait qu'on n'aura pas de destruction prématurée.

## II-C - Notre propre pointeur intelligent : ValuePtr

Afin de rendre plus concrète la notion de pointeur intelligent, et d'étudier les mécanismes du C++ qui rendent leur écriture possible, nous allons maintenant définir notre propre classe de pointeurs intelligents, correspondant à un scénario non prévu dans le standard. Ceux qui sont uniquement intéressés par l'utilisation des pointeurs intelligents sans désirer savoir ce qui se passe sous le capot peuvent passer cette section. D'autant plus que c'est dans cette section que seront utilisés les aspects les plus avancés du C++ de l'article.

La classe que nous allons écrire ne sera pas de qualité industrielle (par exemple, pas de gestion de la thread safety, peu de gestion de l'exception safety...), elle a juste pour but d'illustrer les principaux mécanismes mis en jeu.

### II-C-1 - Problématique

On veut manipuler des formes mathématiques, des cercles, des rectangles... Dans le monde idéal, on écrirait du code comme ça :

```
vector<Shape> v;
v.push_back(Rectangle(0, 0, 10, 20));
v.push_back(Circle(0, 0, 10));
for(vector<Shape>::iterator it = v.begin();
    it != v.end();
    ++it)
{
    it->draw();
}

vector<Shape> v2 = v;
for(vector<Shape>::iterator it = v2.begin();
    it != v2.end();
    ++it)
{
    it->move(2, 2);
}
```

Sauf que comme on le sait, ça ne marche pas ainsi. Il y a ce qu'on appelle du slicing. Si on essaye par exemple de mettre un `Circle` dans notre vecteur, on va se retrouver quelque part à l'intérieur du vecteur à faire un code équivalent à `Shape s = aCircle`. Or, un objet `a` en C++ a un type bien déterminé, ici `Shape`, avec un arrangement mémoire connu à la compilation. On ne peut pas faire entrer un `Circle` dans de la mémoire prévue pour un `Shape`, il n'y a pas assez de place. Cette opération va donc convertir le `Circle` en `Shape`, ce qui correspond à supprimer toutes les variables membres ajoutées au niveau de la classe dérivée, à couper (slice) l'objet pour n'en garder que la partie définie dans la classe de base.

La façon d'utiliser du polymorphisme en C++ passe par des pointeurs (ou des références). On a une zone mémoire allouée pour un Circle, mais on va y accéder par l'intermédiaire d'un pointeur sur un Shape, et c'est ce pointeur que l'on va mettre dans notre conteneur :

```
vector<Shape*> v;
v.push_back(new Rectangle(0, 0, 10, 20));
v.push_back(new Circle(0, 0, 10));
for(vector<Shape*>::iterator it = v.begin();
    it != v.end();
    ++it)
{
    (*it)->draw();
}

vector<Shape*> v2 = v;
for(vector<Shape*>::iterator it = v2.begin();
    it != v2.end();
    ++it)
{
    (*it)->move(2, 2);
}
```

Les problèmes sont multiples : On doit désormais s'occuper de désallouer correctement ce qui a été alloué (ce que je n'ai pas fait ici), de plus, on a sans forcément le vouloir une sémantique d'entité, alors qu'initialement on avait une sémantique de valeur. Ainsi, la deuxième partie du code modifie aussi le contenu de v, ce qui n'était pas le cas dans la version précédente.

Nous sommes dans le cas où l'on utilise des pointeurs alors que l'on souhaite manipuler des données ayant une sémantique de valeur, uniquement parce que l'on veut du polymorphisme. L'idiome classique pour gérer ce genre de cas est l'idiome lettre enveloppe. On va ici voir une autre approche, un peu plus générique dans sa mise en œuvre : On va demander à l'utilisateur de nos classes de manipuler des pointeurs intelligents vers les objets, mais ces pointeurs auront la particularité de copier les objets pointés lorsqu'on copie le pointeur, permettant de restaurer une sémantique de valeur.

## II-C-2 - Structure de base

Le premier point, c'est de choisir comment nommer notre classe. On l'appellera ValuePtr, pour indiquer qu'il s'agit d'un pointeur avec une sémantique de valeur.

On veut pouvoir créer des ValuePtr qui pointent sur divers types de données, des Shape, des Circle, des Rectangle, mais aussi d'autres types que l'on n'a pas encore envisagés. On va donc faire de ValuePtr un template, qui aura comme argument le type sur lequel le pointeur intelligent va pointer.

Enfin, notre classe va encapsuler un pointeur, et aura donc une donnée membre de ce type. Voici donc la structure de base de notre pointeur intelligent.

```
template<class T>
class ValuePtr
{
private:
    T* myPtr;
};
```

## II-C-3 - Ressembler à un pointeur

On veut que notre pointeur intelligent ressemble en terme d'utilisation à un pointeur. Les deux opérations les plus fondamentales que l'on peut effectuer sur un pointeur p, c'est accéder à l'objet pointé, par l'écriture \*p, ou accéder au contenu de l'objet pointé, par l'écriture p->.

L'écriture `*p` n'est en pratique pas utilisée si souvent que ça, mais il n'est pas compliqué de la fournir. On va définir l'opérateur `*` unaire pour qu'il effectue cette tâche. Comme on veut qu'à partir de cet opérateur, l'utilisateur accède directement à l'objet pointé et non à une copie, cet opérateur va retourner une référence vers le contenu du pointeur.

```
template<class T>
class ValuePtr
{
public:
    T& operator*() const {return *myPtr;}
private:
    T* myPtr;
};
```

De son côté, l'écriture `p->` est à la base de l'utilisation du pointeur. On va cette fois la reproduire en surchargeant l'opérateur `->` sur notre classe :

```
template<class T>
class ValuePtr
{
public:
    T* operator->() const {return myPtr;}
private:
    T* myPtr;
};
```

L'opérateur `->` est un peu spécial. Alors qu'en général, `a @ b` se traduit par `operator@(a, b)` ou `a.operator@(b)`, dans le cas de l'opérateur `->`, `a->b` se traduit par `(a.operator->())->b`. Donc, notre opérateur `->` retourne le pointeur nu interne, et le programme va transmettre au pointeur nu ce que l'utilisateur a voulu faire à notre `ValuePtr`. Contrairement au cas précédent, le pointeur nu n'est utilisé que comme une valeur temporaire, auquel l'utilisateur n'a pas un accès direct, et il ne peut donc pas faire de bêtises avec.

## II-C-4 - Implémenter notre sémantique

Nous voulons que quand on crée un `ValuePtr`, il prenne la responsabilité d'un pointeur que l'on passe en argument du constructeur. Ainsi, quand on détruit le `ValuePtr`, il va automatiquement détruire la donnée pointée par `ptr`.

```
template<class T>
class ValuePtr
{
public:
    ValuePtr(T* ptr) : myPtr(ptr) {}
    ~ValuePtr() {delete myPtr;}
    T* operator->() {return myPtr;}
private:
    T* myPtr;
};
```

Que doit-il se passer quand l'utilisateur copie le pointeur ? On veut tout simplement que la valeur pointée soit copiée elle aussi. Une première tentative pourrait être de passer par le constructeur de copie de la valeur pointée :

```
template<class T>
class ValuePtr
{
public:
    ValuePtr(ValuePtr const &other) :
        myPtr(new T (*(other.myPtr)))
    {}
    ValuePtr &operator=(ValuePtr const &other)
    {
        // Je sais que ce code ne gère pas l'auto-affectation
        // mais le but est juste de présenter le concept
        delete myPtr;
        myPtr = new T(*(other.myPtr));
    }
};
```

```

        return *this;
    }
    // ...
};

```

Le problème est que notre principal cas d'utilisation est pour des classes faisant partie d'une hiérarchie polymorphe. On peut donc avoir un `ValuePtr<Shape>` qui en fait pointe sur un `Circle`, mais dans la copie, on va créer un nouvel objet du type `Shape`. Il nous faut utiliser l'idiome du **constructeur de copie virtuel** : Nous allons imposer à notre type `T` d'avoir une fonction (virtuelle en général) `clone` qui produit un double de l'objet existant. On va tant qu'à faire en profiter pour corriger les problèmes d'auto-affectation et d'exception safety de l'opérateur d'affectation.

```

template<class T>
class ValuePtr
{
public:
    ValuePtr(ValuePtr const &other) :
        myPtr(other->clone())
    {}
    void swap(ValuePtr &other)
    {
        std::swap(myPtr, other.myPtr);
    }
    ValuePtr &operator=(ValuePtr const &other)
    {
        ValuePtr<T> temp(other);
        swap(temp);
    }
    // ...
};

```

Nous avons désormais un `ValuePtr` fonctionnel, mais très minimaliste. Voyons comment on peut en rendre l'usage plus simple pour l'utilisateur.

## II-C-5 - Flexibilité pour le type pointé

Actuellement, on impose au type pointé d'implémenter l'idiome du constructeur virtuel par une fonction nommée `clone()`. Pas `Clone()`, ni `Copy()`, ni `copie()`, ni... C'est très restrictif. Trop.

La solution future, quand le C++0x sera sorti, sera simplement de définir dans le concept associé à notre template que l'on a besoin de faire `p.clone()` sur le paramètre template, et de laisser l'utilisateur utiliser les concept\_map pour faire correspondre sa fonction de copie avec ce que veut le concept. Voyons étape par étape comment y parvenir :

On déclare dans un concept qu'un objet `Clonable` est un objet qui possède une fonction `clone` retournant un pointeur du même type. Ce concept est auto afin que toutes les classes ayant cette fonction soient immédiatement considérées comme répondant au concept (sinon, il faudrait définir un concept map dans tous les cas).

```

auto concept Clonable<class T>
{
    T* T::clone();
}

```

Ensuite, on utilise ce concept pour définir notre pointeur intelligent. Ça a deux effets :

- Tout d'abord, si jamais dans l'implémentation de notre pointeur intelligent, on essaye d'utiliser sur un objet de type `T` une fonctionnalité autre que celle définie dans `Clonable`, le code échouera.
- Ensuite, si on essaye d'instancier un pointeur intelligent sur un type non `Clonable`, l'instanciation échouera (avec un message d'erreur qui devrait avoir du sens pour l'utilisateur).

```

template <Clonable T>
class ValuePtr
{

```

```
public:
    ValuePtr(ValuePtr const &other) : myPtr(other->clone())
    {
    }
    // ...
};
```

Pour l'instant on n'a rien résolu, notre classe A par exemple n'étant a priori pas Clonable :

```
class A
{
public:
    A* copy() {return new A(*this);}
};
```

Mais finalement, la personne désirant créer un ValuePtr<A> peut dire que si, A est en fait Clonable, et que quand le pointeur intelligent désire appeler la fonctionnalité clone promise dans le concept, il peut en fait appeler la fonction copy sur l'objet en cours. Il suffit pour ça d'écrire une concept\_map :

```
concept_map Clonable<A>
{
    A* A::clone() {this->copy();}
};
```

Et voilà, le tour est joué.

La solution dans les frontières du langage actuel consiste à passer par l'intermédiaire de ce qu'on appelle une classe de traits. Comme d'habitude, on résout le problème en ajoutant une indirection supplémentaire :

```
template<class T>
class CloneTraits
{
    T* clone(T* t) {return t->clone();}
}

template<class T>
class ValuePtr
{
public:
    ValuePtr(ValuePtr const &other) :
        myPtr(CloneTraits::clone(other.myPtr))
    {}
    // ...
};
```

A priori, on n'a pas gagné grand-chose, juste la complexité apportée par une classe supplémentaire. Par contre, la personne qui souhaite utiliser notre pointeur intelligent avec un type ayant une autre écriture pour clone peut toujours spécialiser la classe CloneTraits :

```
class A
{
public:
    A* copy();
};

template<>
class CloneTraits<A>
{
    A* clone(A* a) {return a->copy();}
};
```

Si vous voulez plus de détails à propos du fonctionnement de cette technique, vous pouvez aller voir [Présentation des classes de Traits et de Politiques en C++](#) par Alp Mestan

## II-C-6 - Ressembler à un pointeur : La suite

Il y a bien des opérations que l'on peut faire sur un pointeur qui ne sont pas prises en compte par notre type. Par exemple, on peut convertir un pointeur sur une classe dérivée vers un pointeur sur une classe de base, ou plus généralement, si U est convertible en T, U\* est convertible en T\* :

```
template<class T>
class ValuePtr
{
public:
    template<class U> ValuePtr(ValuePtr<U> const &uPtr) :
        myPtr(uPtr.get()->clone())
    {}
    T* get() {return myPtr;}
    // ...
};
```

Deux remarques : Déjà, la déclaration de ce constructeur ne dispense pas du constructeur de copie, même si on pourrait croire qu'il s'agit d'un cas plus général de ce dernier. En effet, un constructeur template n'est jamais pris en compte quand il s'agit de trouver un constructeur de copie. Ensuite, ValuePtr<U> étant un type distinct de ValuePtr<T>, on ne peut pas accéder à la donnée privée uPtr.myPtr, d'où la présence de la fonction get, qui peut de toute façon être utile en elle même.

Parmi les opérations supportées par les pointeurs, il y a les opérations arithmétiques, mais celles-ci servent à gérer des tableaux, et sont donc peu utiles pour notre cas.

Il y a aussi la gestion de pointeur nul (par exemple, un ValuePtr créé avec le constructeur par défaut), et le fait qu'un tel pointeur doive se convertir en false dans un contexte où un booléen est attendu. Le plus basique pour y parvenir est (le explicit devant operator bool n'est possible qu'en C++0x. Pour obtenir du code valide en C++98, il faut simplement ne pas le mettre ; on s'expose alors à quelques inconvénients, lire [safe bool](#) pour plus d'informations) :

```
template<class T>
class ValuePtr
{
public:
    explicit operator bool() const
    {
        return myPtr;
    }
};
```

Enfin, on pourrait vouloir permettre la conversion depuis notre pointeur intelligent vers un pointeur nu. La méthode pour le faire consisterait à surcharger operator T\*() mais ce n'est pas forcément une bonne idée. En effet, cette conversion implicite donnerait un accès direct au pointeur nu, et permettrait de fausses manipulations. Bien qu'un tel accès soit probablement nécessaire dans certains cas, il semble plus judicieux de ne le fournir que par une fonction, ce qui oblige un appel explicite, et permet de s'assurer qu'il s'agit bien d'un choix conscient de l'utilisateur.

## II-C-7 - Conclusion

On peut voir que si définir un pointeur intelligent n'est pas très compliqué en principe, en définir un bon commence à devenir plus difficile. Et encore, je suis passé à côté de plein d'aspects (multithreading, par exemple). D'où l'intérêt d'avoir à notre disposition des pointeurs mitonnés aux petits oignons au fil des ans par des experts du domaine.

## III - Les pointeurs intelligents standards

### III-A - shared\_ptr

Le modèle derrière ce pointeur est celui où plusieurs pointeurs permettent l'accès à une même donnée, sans que l'un de ceux-ci soit investi du rôle particulier consistant à être le responsable de la durée de vie de l'objet. Cette responsabilité est partagée (d'où le nom shared) entre tous les pointeurs pointant à un moment donné sur l'objet.

Comment dans ce cas la mémoire sera-elle libérée ? En fait, quand un pointeur est détruit, il vérifie s'il n'était pas le dernier à pointer sur son objet. Si c'est le cas, il détruit l'objet, puisqu'il est sur le point de devenir inaccessible. Afin de connaître cette information, un compteur de référence est associé à l'objet, à chaque fois qu'un nouveau pointeur pointe sur l'objet, le compteur de référence est incrémenté. A chaque fois que le pointeur arrête de pointer sur cet objet (parce qu'il est détruit, ou s'apprête à pointer sur un autre), le compteur est décrémenté. Si le compteur atteint 0, il est temps de détruire l'objet.

Ce fonctionnement assez simple cache certaines subtilités ou usage avancés, que nous allons voir dans la suite de ce chapitre.

#### III-A-1 - Utilisation de base

Comme il est usuel, un shared\_ptr est un template ayant comme paramètre le type d'élément sur lequel il doit pointer. Prenons un petit code d'exemple :

```
void f()
{
    shared_ptr<int> a(new int (42));
    cout << a.use_count() << endl;
    shared_ptr<int> b = a;
    cout << a.use_count() << endl;
    {
        shared_ptr<int> c;
        shared_ptr<int> d (new int (314));
        cout << a.use_count() << endl;
        c = a;
        cout << a.use_count() << endl;
        d = c;
        cout << a.use_count() << endl;
    }
    cout << a.use_count() << endl;
    b.reset();
    cout << a.use_count() << endl;
}
```

La fonction use\_count affiche la valeur du comptage de référence d'un shared\_ptr. Elle ne sert qu'à des fins de débogage, ou d'explication comme ici. Ce programme va commencer par afficher 1, c'est à dire qu'au début, il n'y a qu'un seul pointeur pointant sur l'objet (ici, un entier valant 42). On peut noter que c'est une bonne habitude de stocker dès que possible un pointeur nu dans un pointeur intelligent, afin de s'assurer qu'il n'aura pas l'occasion de fuir. A tel point que le standard définit une fonction make\_shared qui évite toute présence d'un pointeur nu dans le code utilisateur, et que l'on aurait pu utiliser ainsi :

```
shared_ptr<int> a = make_shared<int>(42);
```

Cette fonction a aussi l'avantage d'être plus sûre (voir plus loin) et d'offrir un gain de performances par rapport au code précédent. Seul problème : Elle demande des techniques spécifiques à C++0x pour pouvoir être implémentée, et il faut donc apprendre encore quelque temps à s'en passer.

Revenons à notre premier exemple. La deuxième ligne affiche 2 : On a construit un autre shared\_ptr qui pointe sur la même zone. La troisième ligne affiche toujours 2 : On a construit un pointeur c sur un entier, mais sans l'initialiser. Un

tel pointeur ne pointe sur rien. C'est un peu l'équivalent d'un pointeur nul, pour des `shared_ptr`. De même, le pointeur d pointe sur un autre entier. Il n'y a donc aucune raison de modifier le comptage de référence vers notre entier 42.

Par contre, à la ligne suivante, ce comptage passe à 3 : On a fait pointer `c` vers le même objet, le comptage augmente donc. À la ligne suivante, le compteur est encore incrémenté, puisque `d` pointe désormais aussi sur notre objet. À cette occasion, `d` s'est arrêté de pointer sur l'entier valant 314, et le comptage de référence de cet objet a donc diminué. Comme il a atteint 0 (il n'y avait que `d` pour pointer dessus), cet objet a donc été détruit.

Quand on sort du scope, les variables `c` et `d` sont détruites, notre comptage de référence redescend donc à 2. Puis, on appelle `b.reset()`, qui a le même effet que si on avait écrit `b = shared_ptr<int>()`; `b` ne pointe plus sur rien après cet appel. La dernière ligne que l'on affiche est donc 1. A la fin de la fonction, `a` sera détruit, faisant passer le comptage de référence à 0, et l'entier valant 42 sera détruit.

### III-A-2 - Compatibilité entre `shared_ptr`

Il existe un certain nombre de conversions possibles entre des pointeurs nus. Les conversions équivalentes existent entre les `shared_ptr`. En voici quelques exemples :

```
class A { /* ... */ };
class B : public A { /* ... */ };

B* b1(new B);
A* a(b1);
B* b2 = dynamic_cast<B*>(a);

shared_ptr<B> sb1(new B);
shared_ptr<A> sa(sb1);
shared_ptr<B> sb2 = dynamic_pointer_cast<B>(sa);
```

On peut noter en particulier que le mot clef `dynamic_cast` est remplacé pour les `shared_ptr` par une fonction `dynamic_pointer_cast` qui s'utilise de manière semblable (sauf que le paramètre template est `B`, et non pas `shared_ptr<B>`), et qui retourne un pointeur ne pointant sur rien si la conversion n'a pu avoir lieu. Il existe de même une fonction `static_pointer_cast` et mimant l'effet d'un `static_cast` pour un pointeur nu.

### III-A-3 - Risque de fuite mémoire : Initialisation

Il y a deux cas où malgré l'utilisation de `shared_ptr`, on a toujours des fuites mémoires. Le premier est lié non pas aux `shared_ptr`, mais au fait que, temporairement, la mémoire n'a pas été gérée par des `shared_ptr`. Soit le code suivant, a priori bien intentionné :

```
int f(shared_ptr<int> i, int j);
int g();

f(shared_ptr<int> (new int (42)), g());
```

Ce code respecte bien le conseil d'enrober au plus vite un pointeur nu dans un pointeur, mais il présente un risque de fuite mémoire. En effet, le compilateur doit effectuer 4 actions quand il voit cette ligne :

- 1 Créer un entier valant 42 dans une nouvelle zone mémoire allouée pour l'occasion par `new`
- 2 Créer un `shared_ptr` avec le pointeur obtenu à l'étape 1
- 3 Appeler la fonction `g`
- 4 Appeler la fonction `f`

Il y a quelques contraintes d'ordre entre ces opérations, par exemple 2 ne peut avoir lieu qu'après 1, et 4 qu'à la fin. Par contre rien n'empêche le compilateur de choisir l'ordre : 1 3 2 4.

Dans ce cas, si l'appel de `g` lance une exception, comme le `shared_ptr` n'a pas encore eu le temps de prendre possession de la mémoire fraîchement allouée, il n'aura pas la possibilité de la libérer. A ce problème, deux solutions :

Soit ne jamais créer de `shared_ptr` temporaires, en remplaçant le code précédent par :

```
int f(shared_ptr<int> i, int j);
int g();

shared_ptr<int> si (new int (42));
f(si, g());
```

Soit ne pas allouer de pointeur nu du tout, ce qui évite qu'ils puissent fuir (il faut le C++0x pour pouvoir faire ça) :

```
int f(shared_ptr<int> i, int j);
int g();
f(make_shared<int>(42), g());
```

### III-A-4 - Risque de fuite mémoire : Cycle

Le plus gros problème à l'utilisation d'un `shared_ptr` est probablement qu'il ne gère pas correctement les cycles. Considérons le code suivant (dans du vrai code, les cycles seraient évidemment moins faciles à détecter) :

```
class A
{
    // ...
    shared_ptr<B> myB;
};

class B
{
    // ...
    shared_ptr<A> myA;
};

shared_ptr<A> a = new A;
shared_ptr<B> b = new B;
cout << a.use_count() << ", " << b.use_count() << endl;
a->myB = b;
cout << a.use_count() << ", " << b.use_count() << endl;
b->myA = a;
cout << a.use_count() << ", " << b.use_count() << endl;
a.reset();
b.reset();
```

Si l'on s'intéresse aux compteurs de référence de nos objets, on constate qu'à la première ligne, ils valent tous deux 1, rien de surprenant jusque là. À la seconde ligne d'affichage, ils valent respectivement 1 et 2. En effet, deux pointeurs pointent sur l'objet de type B : `b`, et `a->myB`. À la troisième ligne d'affichage, les valeurs sont toutes deux de 2.

Après les resets, les objets A et B sont devenus inaccessibles depuis notre code, mais `myB` dans l'objet de type A fait survivre l'objet de type B en maintenant son comptage de référence à 1, et `myA` dans ce dernier fait survivre l'objet de type A. Nos deux objets se font donc survivre mutuellement. Nous avons une fuite mémoire. C'est le principal défaut des `shared_ptr`s par rapport à un système comme un glaneur de cellules. La classe `weak_ptr`, qui fait l'objet du chapitre suivant, permet de s'en sortir dans ce genre de situations.

### III-A-5 - Obtenir un shared\_ptr à partir de this

Il y a deux cas où l'on veut obtenir un `shared_ptr` à partir du pointeur nu `this`. Le premier, c'est dans une fonction membre d'une classe, si on a besoin d'appeler sur nous-même une fonction externe à la classe qui prend en paramètre un `shared_ptr` (pour que ça marche, il faut bien entendu que l'on soit certain que l'objet courant est déjà géré par `shared_ptr`).

Il n'y a pas de solutions non intrusives à ce problème. Il faut modifier la classe afin qu'elle sache d'elle-même comment retrouver le `shared_ptr` lui correspondant. Pour ça, on fait dériver la classe d'une classe nommée `enable_shared_from_this` :

```
class A;
void registerA(shared_ptr<A> ptr);

class A : public enable_shared_from_this<A>
{
    void f()
    {
        registerA(shared_from_this());
    }
};
```

On remarque que `enable_shared_from_this` est un template, qui prend en paramètre template la classe même dans laquelle on veut obtenir cette fonctionnalité.

Le second cas est un peu particulier, c'est quand on veut obtenir la même chose, mais dans le constructeur de la classe. Le problème, c'est que lors de sa construction, l'objet n'est pas encore construit, et donc ne peut pas encore avoir été associé à un `shared_ptr`. Reprenons l'exemple précédent :

```
class A;
void registerA(shared_ptr<A> ptr);

class A
{
public:
    A()
    {
        registerA(???);
    }
};
```

La solution à ce problème est de modifier légèrement l'architecture du code, afin de passer par une factory au lieu de vouloir faire le travail dans un constructeur.

```
class A;
void registerA(shared_ptr<A> ptr);

class A
{
private:
    A() {}
public:
    static shared_ptr<A> create()
    {
        shared_ptr<A> result(new A);
        registerA(result);
        return result;
    }
};
```

### III-A-6 - Spécifier une fonction de désallocation

Il y a des cas où l'on n'a pas créé un objet à l'aide de `new`, mais à l'aide d'une autre fonction. Par exemple, supposons que l'on utilise une bibliothèque qui nous permette de manipuler des `Toto*`, que l'on obtient et détruit à l'aide des fonctions suivantes :

```
Toto* createToto();
void deleteToto(Toto *t);
```

Si l'on écrit un simple `shared_ptr<Toto>`, quand il va vouloir détruire l'objet, il va appeler `delete` dessus, et non pas la fonction spécifique `deleteToto`. Est-ce à dire que l'on doit se passer de la sûreté et du confort des `shared_ptr` dans ce cas ? Heureusement non, les `shared_ptr` ont un second argument dans leur constructeur qui permet de spécifier comment l'objet géré doit être détruit :

```
shared_ptr<Toto> createSafeToto()
{
    return shared_ptr<Toto> (createToto(), &deleteToto);
}
```

### III-A-7 - Aliasing

Il peut arriver qu'un utilisateur souhaite manipuler une sous partie d'un objet par `shared_ptr`. Supposons une classe :

```
struct Voiture
{
    vector<Roue> mesRoues;
};
```

Supposons que nos voitures soient gérées par l'intermédiaire des `shared_ptr`, et intéressons nous au code qui travaille avec les roues. On a envie que ce code influe sur la durée de vie d'une voiture, et que tant que ce code possède une roue, la voiture à laquelle la roue est attachée ne soit pas détruite.

On peut envisager plusieurs façons d'y parvenir. Par exemple, la roue peut avoir un `shared_ptr` sur sa voiture (attention aux boucles !). Une autre solution serait que le code gérant les roues prenne en paramètre des `shared_ptr<Voiture>`, mais ce n'est pas très propre car ça introduit un couplage inutile entre ce code et le code de voiture (que faire par exemple si je veux utiliser le même code sur des roues en vrac non encore montées ?).

Finalement, les `shared_ptr` présentent une autre option, plus claire, à ce problème. L'idée est de spécifier un `shared_ptr` sur une sous-partie de l'objet mais qui partage son comptage de référence avec l'objet dans sa totalité.

```
struct Voiture
{
    vector<Roue> mesRoues;
};

shared_ptr<Roue> getRoue(shared_ptr<Voiture> const &v, int id)
{
    // Premier argument : Où est mon comptage de référence
    // Second argument : Où est-ce que je pointe
    return shared_ptr<Roue>(v, &v->mesRoues[id]);
}
```

### III-A-8 - Inconvénients des shared\_ptr

Le premier inconvénient des `shared_ptr`, c'est qu'ils ne gèrent pas les cycles, on en a parlé.

Un second inconvénient peut être les performances. Sur deux points, ils sont moins performants que des pointeurs nus (mais ils apportent plus de choses, donc il ne faut pas conclure trop vite) :

- Il y a besoin d'espace supplémentaire pour gérer le comptage de référence. De plus, cet espace peut demander une allocation mémoire supplémentaire lors de la création, sauf si la fonction `make_shared` (C++0x) est utilisée.
- Chaque opération de copie du pointeur doit incrémenter le compteur de référence, ce qui peut être assez coûteux dans un contexte multithread. Comparé à un glaneur de cellule, le coût est probablement comparable, selon l'usage qui en est fait, mais cette opération a lieu pendant l'opération elle-même, et non pendant un moment où le système attend, ce qui est plus perceptible.

La move semantic offerte par C++0x (et qui commence à être disponible dans certains compilateurs (gcc par exemple) peut nettement réduire le nombre de ces coûteuses opérations de copie. En attendant cette fonctionnalité, certaines implémentations ont spécialisé certaines classes (comme `vector<shared_ptr<T>>` pour Visual C++/TR1) afin de minimiser ces copies dans des cas d'utilisation très courants.

## III-B - weak\_ptr

`weak_ptr` a été conçu spécifiquement pour travailler en collaboration avec `shared_ptr`, pour casser les cycles. L'idée de base est de dire qu'en fait, parmi les pointeurs sur un objet, certains (les `shared_ptr`) se partagent la responsabilité de faire vivre ou mourir ce dernier, le possèdent, et d'autres (les `weak_ptr`) y ont un simple accès, mais sans aucune responsabilité associée.

Par exemple, imaginons une base de données géographiques. On pourrait avoir une classe Région et une classe Département. Une région contiendrait une liste de départements, et un département aurait besoin de savoir dans quelle région il est situé. Si l'on n'y prend pas garde, on a donc un lien cyclique entre régions et départements. Dans ce cas, on pourra décider que la région connaît ses départements sous forme de `shared_ptr`, et que le département ne possède qu'un `weak_ptr` sur sa région.

Un `weak_ptr` n'impacte donc pas le comptage de référence de l'objet sur lequel il pointe. Mais quel avantage peut-il bien apporter par rapport à un simple pointeur nu qui pointerait sur le même objet qu'un `shared_ptr` ? La sécurité. En effet, au moment où le compteur de référence de l'objet passe à 0, ce dernier est détruit. Si un autre pointeur essaye alors d'accéder à cet objet, c'est un comportement indéfini. Avec un `weak_ptr`, l'utilisateur est averti que l'objet pointé est désormais mort, et qu'il doit se passer de lui. Nous allons voir comment.

### III-B-1 - Utilisation de base

Il est possible de créer un `weak_ptr` soit à partir d'un `shared_ptr`, soit à partir d'un autre `weak_ptr`. Contrairement à `shared_ptr`, il n'est pas possible d'en créer un à partir d'un pointeur nu, un `weak_ptr` n'étant là que pour travailler en collaboration avec un `shared_ptr`.

La première grosse différence que l'on voit entre les deux types de pointeurs à l'utilisation, c'est que `weak_ptr` n'est pas vraiment un pointeur intelligent, il lui manque en effet un surcharge de l'opérateur `->`.

Le mode d'emploi pour accéder à l'objet pointé est de créer un `shared_ptr` à partir du `weak_ptr`, et de travailler à partir de ce nouveau pointeur. Pour ça, deux méthodes sont possibles, le constructeur de `shared_ptr` prenant un `weak_ptr` en paramètre et la fonction `lock` (qui est souvent plus explicite):

```
void f(weak_ptr<T> weakData)
{
    shared_ptr<T> sharedData = weakData.lock();
    // ou : shared_ptr<T> sharedData(weak_data);
    if (sharedData)
    {
        sharedData->onPeutTravaillerAvec();
    }
    else
    {
        // l'objet pointé a été détruit
    }
}
```

Pour quelle raison cette obligation ? Imaginons l'espace d'un instant qu'on puisse travailler directement avec un `weak_ptr` :

```
void f(weak_ptr<T> weakData)
{
    if (weakData.expired())
```

```

    {
        // l'objet pointé a été détruit, c'est le
        // rôle de la fonction expired de nous en avertir
    }
    else
    {
        weakData->onPeutTravaillerAvec(); // ?
    }
}

```

Dans un environnement multithread, ce code qui semble pourtant bien intentionné peut avoir un problème si l'objet est détruit entre le moment où on teste expired et le moment où l'on appelle la fonction.

Qu'à cela ne tienne, diront certains, il suffit que l'opérateur -> teste si l'objet existe encore, et lance une exception si ce n'est pas le cas, en mettant les verrous qui vont bien pour que l'objet survive le temps que l'on appelle l'opérateur->. Mais même là, ça ne marche pas : Au moment où la fonction onPeutTravaillerAvec() est appelée, le code de l'opérateur -> a totalement fini de s'exécuter, et rien n'empêche donc l'objet d'être détruit alors qu'on travaille dessus.

Le passage par un shared\_ptr permet donc, si le comptage de référence est sur le point de passer à 0 dans un autre thread, de prolonger la durée de vie de l'objet pointé le temps qu'on en ait fini avec lui.

En pratique, l'utilisation de expired, tout comme de use\_count est très rare, et, sauf pour du code de test ou de débogage indique probablement une mauvaise compréhension du fonctionnement du code en multithread.

### III-B-2 - Ordre des dépendances

Quand on a une boucle entre deux objets A et B, deux solutions sont possibles pour la casser. Donner à A un shared\_ptr sur B, et à B un weak\_ptr sur A, ou l'inverse. Il y a des cas où l'ordre dans lequel le pointeur doit être fort ou faible apparaît comme évident, d'autres moins.

En particulier, dans le cas de collection, j'ai vu des gens argumenter (très) longuement sur le sujet. Supposons par exemple un tiroir rempli de chaussettes, chaque chaussette ayant besoin de savoir où elle est rangée.

Première possibilité :

```

struct Tiroir
{
    vector<shared_ptr<Chaussette> > monContenu;
};
struct Chaussette
{
    weak_ptr<Tiroir> monTiroir;
};

```

Seconde possibilité :

```

struct Tiroir
{
    vector<weak_ptr<Chaussette> > monContenu;
};
struct Chaussette
{
    shared_ptr<Tiroir> monTiroir;
};

```

La première possibilité paraît souvent plus naturelle, et reproduit l'idée assez logique que si l'on jette un tiroir à la poubelle, on a automatiquement jeté les chaussettes qu'il contient (sauf une chaussette qui serait tenue par quelqu'un d'autre). La seconde approche part plutôt du mode de pensée que tant qu'on utilise une chaussette, il faut pouvoir la ranger, et donc le tiroir doit survivre tant qu'il lui reste une chaussette.

Les deux points de vue ont des arguments, et le bon choix réside probablement dans la manière dont le programme compte utiliser tiroir et chaussettes.

### III-C - unique\_ptr

unique\_ptr (utilisable uniquement dans C++0x, utiliser boost::scoped\_ptr pour avoir un comportement semblable, mais moins riche, en attendant) représente un pointeur qui, comme son nom l'indique, est le seul pointeur à pointer sur un objet. Quand le pointeur est détruit, l'objet est détruit.

Que se passe-t-il quand on copie ce pointeur ? Il ne peut rien se passer, car la copie en est interdite. Par contre (et contrairement à scoped\_ptr), il est possible de déplacer ce pointeur, pour transférer la responsabilité de l'objet pointé à quelqu'un d'autre.

#### III-C-1 - Utilisation de base

L'une des principales utilisations de unique\_ptr est de mettre en place le RAII pour des classes pas forcément prévues pour ça. A cette fin, on utilisera assez souvent les possibilités de spécifier la fonction de désallocation, qui sont les mêmes que pour shared\_ptr.

```
void *createProcess();
void deleteProcess(void *p);

void travailleAvecUnNouveauProcessus()
{
    unique_ptr<void> process(createProcess(), &deleteProcess);
    // on peut travailler avec p, il sera automatiquement
    // détruit qqand on quitte la fonction.
}
```

#### III-C-2 - Avantage par rapport à un shared\_ptr

Il y a principalement deux avantages. Le premier est en terme de performance. unique\_ptr ne demande pas de compteur de référence, c'est un pointeur très léger.

Le second est en terme de documentation de code. Les cas d'utilisation d'un unique\_ptr et ceux d'un shared\_ptr sont en fait assez différents, et il est pratique d'avoir une classe qui indique clairement au lecteur du code dans quel cas on se trouve.

## IV - Conclusion

En standard, le C++ ne dispose pas de mécanisme de glaneur de cellules (garbage collector), ce qui a priori pourrait le rendre plus complexe à utiliser qu'un langage en disposant. Mais en fait, avec les pointeurs intelligents, on possède un autre mécanisme pour résoudre le même problème de gestion de ressources. Il est de fait très rare en C++ bien écrit de voir le mot clef delete (sauf bien évidemment dans l'implémentation de structures très bas niveau comme les pointeurs intelligents), et la gestion des ressources devient alors une tâche relativement transparente.

Par rapport à un glaneur de cellules, les compromis sont différents :

- La libération des ressources est déterministe, à la destruction du pointeur (unique\_ptr) ou du dernier pointeur utilisant un objet (shared\_ptr), ce qui est particulièrement utile pour des ressources en nombre plus limité que la mémoire (des descripteurs graphiques...), ou étant susceptible d'être utilisées par plusieurs applications (fichiers...). Cette simplicité aide aussi lors du débogage.
- La gestion des cycles pose un problème, demandant de réfléchir sur le sens des dépendances sur une structure de donnée complexe.

- Les performances sont impactées différemment. Un ramasse-miettes a tendance à être plus gourmand en mémoire, mais demande du temps processeur moins souvent, et généralement à des moments où ce dernier ne fait rien.
- Les différents types de pointeurs intelligents permettent de mettre en œuvre des stratégies diverses à l'intérieur d'un même programme.

Le C++98 est assez pauvre en pointeurs intelligents, mais avec ce qu'apporte le standard de fait boost, ou encore mieux ce que nous fournira le C++0x, on a à notre disposition tout un arsenal devant lequel les fuites mémoire et autres accès invalides n'ont qu'à bien se tenir !

Merci beaucoup à **diogene** et à **Aszarsha** pour la relecture de cet article