# Approfondir const en C++

Christophe [Groove] Riccio
12/07/2007

## Introduction

Ce document a pour but de présenter quelques bonnes exemples d'usages des références et des pointeurs dans le cadre de la problématique d'écrire un code robuste. Beaucoup de programmeurs C++ utilisent des pointeurs dans leurs codes sans vraiment se préoccuper du niveau accès qu'ils donnent aux variables pointées. Il s'agit ici d'approfondir la compréhension du mot clé ***const*** et d'éveiller son utilisation dans diverses situations.

## Différence en référence et pointeur

En dehors de la syntaxe, c'est à dire, l'opérateur "." à la place de l'opérateur "->" il y a très peu de différences entre une référence et un pointeur. En fait, dernière pour le compilateur, une référence est traitée comme un pointeur. C'est un pur produit du langage C++ sans fondement matériel.
Les différences sont :

- 1. Une référence ne peut-être nulle en aucun cas
- 2. Elle doit être initialisée avec une variable à sa création
- 3. Elle ne référencera jamais une autre variable

```
int  A = 1;
int B = 2;

int& r = A; // Créé une référence sur A
int* p = &A; // Créé un pointeur sur A

p = &B; // Change la variable pointé
r = B; // Change la valeur de la variable référencée
*p = B; // Change la valeur de la variable pointé
```

Il est tout à fait possible de créer un pointeur, puis de l'initialiser, pas une référence :

```
int* p; // Ok
p = &A; // Ok

int& r; // Erreur: 'r' : references must be initialized
r = A;
```

## Pointer sur variable constant & Pointer constant sur variable

L'usage des pointeurs avec le mot clef ***const*** est très fréquent sous la forme suivante :

```
const int* p = &A; // Pointer sur variable constante
```

Cela signifie que le pointeur ne permet pas de modifier la valeur de A. Cependant, ce n'est pas la seule utilisation possible de ***const*** avec un pointeur. Nous pouvons aussi attribuer à un pointeur un comportement spécifique d'une référence : L'obligation d'initialisation et l'impossibilité de changer la variable pointé.

Il est possible d'offrir les propriétés 2 et 3 de la section précédente d'une référence à un pointeur en le déclarant comme pointeur constant sur variable :

```
int* const p = &A; // Pointer constant sur variable, ~ int& r = A
```

Cette forme de pointeur est alors similaire à une référence mise à part le fait que ce pointeur peut-être initialisée à nulle.

Il est également possible de cumuler les deux propriétés ainsi :

```
const int* const p = &A; // Pointer constant sur variable constante ~ const int& r = A
```

Cette forme de pointeur est alors similaire à une référence constante mise à part le fait que ce pointeur peut-être initialisée à nulle.On parle alors de pointeur constant sur une variable constante.

**Paramètres de fonctions : Référence ou copie de variable ?**

Il s'agit dans ce paragraphe de répondre à la problématique de vitesse que pose les références ou les pointeurs. Dans la plus part des cas et selon la bonne règle des 20 – 80, l'utilisation de références ou de variables à un impact négligeable sur l'efficacité d'un code. Cependant, pour la beauté du geste et les 20% de cas ou l'impact peu avoir une certaine importance (qui reste très faible !), voyons ce qu'il en est.

Nous avons noté qu'une référence est en faite interprétée comme un pointeur par le compilateur. En conséquence, passer un pointeur en paramètre d'une fonction revient à copier un pointeur donc une copie d'une variable de 32 ou 64 bits suivant le système. A cela, il faut également ajouter le coût du déréférencement du pointeur. Si l'on copie un entier codé sur 32 bits alors le coût est dans tous les cas inférieur car il n'y a pas le coût du déréférencement. De manière générale, tous les types primitifs et les énumérations gagnent à être passé par copie et toutes les structures et classes gagnent à être passées par référence. Dans tous les cas, par défaut les paramètres d'une fonction doivent être déclaré avec le mot clé ***const***, au nom de la robusté mais aussi de certaines opportunités d'optimisations qu'apportent ce qualificatif pour le compilateur.

**Priorités d'utilisations des références et pointeurs constants**

Dans ce paragraphe, il s'agit de mettre au clair le prototype qu'une fonction par défaut lorsqu'on créé une nouvelle fonction membre d'une classe afin d'assurer le maximum de robustesse. In fait, il s'agit par ce prototype par défaut de réduire au maximum les droits d'accès aux membres de la classe ou aux paramètres. Chaque droit d'accès étant ainsi plus méticuleusement donné suivant un réel besoin. Le résultat est le suivant:

```
class C
{
public:
	void f(const T& type) const;
	const T2& f(const T3& type) const;
};
```

**Fonction membre const**

L'utilisation du mot clé ***const*** sur une fonction membre est un mécanisme de base nécessaire pour indiquer que la fonction ne modifie pas le contenu de la classe. Cependant, suivant le prototype de la fonction, ce n'est pas pour autant qu'elle ne donne pas un accès permettant de modifier la classe.

```
class C
{
	int& GetData() const;
...
};
```

Voilà une forme très incongru qui ressemble à un accesseur mais qui n'en est pas un. Si ***GetData*** retourne une variable membre de ***C***, alors cette variable est modifiable depuis l'extérieur de la classe ... ce qui est tout sauf sure. Si ***GetData*** retourne une variable créé dans ***GetData*** alors nous avons immanquablement un beau crash car la variable est détruite en quittant la fonction ***GetData***. En fait, tant qu'à avoir l'idée d'écrire une telle fonction, autant déclarer une variable membre non constante public, c'est même un choix moins risqué !

**Variables membres constantes**

Il arrive fréquemment que l'on créé des instances dont une partie des valeurs des variables membres ne changera jamais. Un tel comportement indique qu'il serait pertinent de songer à utiliser des variables membres constantes. Toutes les variables membres constantes doivent être initialisées dans le constructeur.

```
class C
{
public:
	C(int V) :
		V(V)
	{}

	const int V;
};
```

En plus, du fait de garantir le fait que la variable ne sera jamais modifiée, un autre avantage est de permettre un accès public en lecture seul à la variable sans nécessiter la création d'un accesseur.

Il est également possible de créer des références ou des pointeurs constants membres. Il s'agit d'une démarche intéressante lorsque l'on souhaite que la classe ne puisse pas modifier la valeur de l'instance alors qu'une autre partie du code est chargé de la modifier. Attention ! Ce type de stratégie comporte un risque important au niveau de l'ordre de destruction des variables. Si l'instance référencée est détruite avant l'instance référençante alors il y aura un crash lors de toute utilisation de la référence ... car elle référence une variable qui n'existe plus.

**Accès en lecture seul à des variables membres non constantes**

L'utilisation de références constantes comme variables membres peut offrir encore d'autres avantages au niveau des accès aux données. Il est en effet possible de donner des accès en lecture seul à un ensemble de variables au moyen d'une référence constante unique et de structures imbriquées :

```
class C
{
	struct S
	{
		struct S1
		{
			int A;
```

```
                int B;
        };

        struct S2
        {
                int A;
                int B;
        };

        S1 s1;
        S1 s2;

        int A;
        int B;
    };

public:
    C(const C::S& v) :
        r(s),
        s(v)
    {}

    const S& r;

private:
    S s;
};
```

J'ai personnellement utilisé cette stratégie à plusieurs reprises dans le cas d'utilisations de fichiers XML. Dans mon cas, la classe ***C*** était mon parseur XML pour un type de fichier spécifique (respectant une DTD ou un Schema XML) et la structure S était mon document XML contenant toutes les données chargées, chaque structure imbriquée étant un élément XML.

**Variables anonymes**

Dans certains cas, il n'est pas nécessaire de nommer une variable car il n'y a aucun appel de fonctions ou utilisation de variables membres.

```
class Truc;
std::vector<Truc> trucs;
trucs.push_back(Truc("coucou"));
```

Ceci n'est possible que dans où le paramètre de la fonction qui prend la variable anonyme en paramètre est ***const*** ou passé par copie. Ce peut-être une référence simple mais une référence constante. Visual Studio 8 acceptera une référence non constante mais c'est un comportement non standard qui ne sera pas accepté par GCC et qui d'un point de vue logique, laisse perplexe. En effet, on utilise une référence non constante généralement pour récupérer le résultat d'une fonction or l'objet n'étant pas nommé, nous n'avons aucun moyen d'accéder au cet objet.