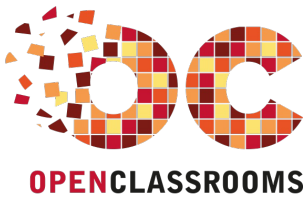
La const-correctness expliquée aux Zéros

Par De passage



www.openclassrooms.com

Dernière mise à jour le 9/07/2011

Sommaire

La const-correctness expliquée aux Zéros

Par De passage
Mise à jour : 09/07/2011
Difficulté : Intermédiaire Durée d'étude : 1 heure

Bonjour ami Zéro.

Nous allons parler ici de la const-correctness. Derrière ce nom barbare se cache un élément important du C++ qui est généralement mal maîtrisé, voir complètement ignoré par les débutants.
Avec ce tuto, la constance n'aura plus de secrets pour vous.

Vous ne devriez pas avoir besoin d'un bon niveau en C++ pour suivre ce tutoriel. Le Zéro moyen à qui il s'adresse est celui qui a déjà écrit quelques classes et fonctions, même très simples, sans se soucier de problèmes de constance.
Il est possible que vous ne soyez pas familiers avec certaines notions évoquées ici. Je vous ai généralement laissé un lien vers un article (le plus accessible possible) pour que vous puissiez découvrir ou réviser ce dont il est question.
Sommaire du tutoriel :

Introduction

Qu'est-ce que la const-correctness ?

On pourrait (parce qu'on ne le fait jamais) traduire le terme de **const-correctness** par « correction de la constance », le mot « correction » renvoyant bien sûr à la qualité de ce qui est correct. La const-correctness est donc tout simplement la manière d'utiliser correctement la constance dans un code, et un code écrit de cette manière est dit **const-correct**.

La constance en C++ est matérialisée par un mot-clé : `const` qui, vous le savez, permet notamment de déclarer une variable comme étant constante, c'est-à-dire que sa valeur n'est pas sensée changer au cours de l'exécution du programme.
La const-correctness inclue également l'utilisation d'un mot-clé et d'un opérateur que l'on emploie moins mais que nous allons étudier ici : `mutable` et `const_cast`. Ces deux éléments permettent dans une certaine mesure d'ignorer la constance d'une variable.

Pourquoi écrire un code const-correct ?

L'existence de `mutable` et de `const_cast` nous amène à cette terrible vérité : une variable constante peut voir son état modifié durant l'exécution du programme.

...

Mais dans ce cas, quel est l'intérêt de déclarer une variable constante si elle peut être modifiée ?

Certains programmeurs pensent que le mot-clé `const` permet au compilateur de faire des optimisations. C'est faux la plupart du temps, on n'est jamais à l'abri d'un `const_cast` ou d'un `mutable` qui viendrait rompre l'immutabilité supposée de notre objet. Je n'entrerais pas dans le détail des cas où la présence de `const` permet ou non des optimisations, si cela vous intéresse, je vous renvoie à cet article (en anglais) d'Herb Sutter qui traite de cette question bien mieux que je ne saurais le faire. Nous allons tout de même aborder une de ces optimisations, très importante, facile à mettre en place et qui à elle seule va justifier l'écriture d'un code const-correct.

Hormis ce cas la const-correctness est utile au développeur comme sécurité lorsqu'il programme, puisque modifier accidentellement une variable constante entrainera une erreur de compilation. Elle sert également de documentation. Par exemple, utiliser des paramètres constants indique immédiatement à l'utilisateur que la fonction ne modifiera pas les variables qu'on lui envoie.

Au cours de ce tutoriel, nous allons approfondir ces avantages et en découvrir de nouveaux. Vous verrez que bientôt la const-correctness vous deviendra indispensable.

Éléments de syntaxes et généralités

Les valeurs constantes

Comme le veut la politique du site, nous allons partir de zéro et reprendre la base de l'utilisation de `const`. La const-correctness touchant le développement d'une application, la plupart des exemples que je vais présenter n'auront d'intérêt qu'à la compilation, ne vous en étonnez pas.

Le mot-clé `const` empêche toute modification ultérieure de la valeur d'une variable. L'**état** de cette variable ne pourra plus être modifié. Essayer de le faire malgré tout entrainera une erreur à la compilation. Dans le cas d'une classe ou d'une structure, tous les sous-objets de l'objet constant sont également constants.

Déclarer une variable constante se fait comme ceci :

Code : C++

```
int const maVariable;
```

Cette déclaration peut vous sembler étonnante. Vous avez certainement appris, via le tuto de M@teo21 par exemple, qu'une constante se déclarait :

Code : C++

```
const int maVariable;
```

Cela n'est pas tout à fait exact. En fait, la règle générale est que le mot-clé `const` s'applique à ce qui se trouve directement à sa gauche ou, s'il n'y a rien qui le précède dans la déclaration à ce qui se trouve à sa droite. Ajoutons à cela que dans le cas d'un type composé (comme un pointeur ou un tableau), la constance est applicable exclusivement à l'élément directement à gauche (ou à droite s'il n'y a rien à gauche) et non à l'ensemble de la déclaration.

Pour s'en convaincre, examinons le code suivant :

Code : C++

```
int i, j;
int const * p = &i;

*p = j; //Erreur, la valeur pointée par p est constante.
p = &j; //Correct, le pointeur p n'est pas constant
```

Seule la deuxième ligne compile, ce qui corrobore bien l'affirmation faite plus haut. Pour ma part, je préfère mettre le const à gauche lorsque c'est possible pour faciliter la lecture et éviter une confusion si le lecteur ne connait pas la règle.

Si vous avez essayé de compiler les exemples précédents, vous vous serez aperçu que cela ne fonctionne pas. La raison est simple : une variable constante doit être initialisée lorsqu'elle est déclarée. Et c'est assez logique : en C++ si vous n'initialisez pas une variable celle-ci peut prendre n'importe quelle valeur, et comme vous ne pourrez la modifier par la suite, le résultat serait être assez hasardeux.
Je me permets parfois de ne pas initialiser les variables dans les exemples pour ne conserver que ce qui a de l'importance. De votre côté n'oubliez pas d'attribuer une valeur à chaque constante que vous déclarerez.

Question subsidiaire : comment déclarer un pointeur constant ? Un pointeur constant sur une valeur constante ?

Si vous avez compris ce que je viens de dire, cela ne devrait poser aucun problème.

Voici la réponse :

Code : C++

```
int * const p; //Pointeur constant sur un int non constant
const int * const q; //Pointeur constant sur un int constant
```

Un pointeur constant ne peut voir l'adresse qu'il contient modifiée. A la syntaxe près, il est donc identique à une référence et l'on préfèrera systématiquement utiliser cette dernière.

Une dernière petite remarque pour clore ce paragraphe :

Code : C++

```
int& const i;
```

Cette déclaration est absurde. Vous le savez maintenant, ici le mot-clé **const** ne s'applique pas à int, et il est impossible de réinitialiser une référence pour en faire un alias d'une autre variable. En clair, une référence est toujours constante, c'est le type pointé qui peut l'être ou non (toutefois, par abus de langage, on désignera par « référence constante » une référence vers un type constant).
La norme interdit cette syntaxe mais certains compilateurs s'ils ne sont pas bien réglés laisserons passer l'erreur. Notez qu'il est possible d'arriver à une déclaration de ce genre via l'utilisation de **typedef**. Dans ce cas on ne considère pas ça comme une erreur : un type plusieurs fois qualifié constant étant simplement constant.

Les fonctions constantes

Le terme de fonction constante ne peut s'appliquer qu'à une **fonction membre**. Et pour cause ! Le principe d'une fonction constante est qu'elle ne modifie pas l'objet sur lequel elle est appelée.

Une fonction constante est définie comme ceci :

Code : C++

```
struct Exemple
{
    void bar() const
    {
    }

//Ou, si l'implémentation est dissociée de la déclaration :
    void foo() const;
};

void Exemple::foo() const
{
}
```

Comme pour les valeurs constantes, le mot-clé vient se placer à droite de ce qui est qualifié (le prototype) mais avant le corps de la fonction. Notez la répétition du mot-clé lorsque l'implémentation est séparée de la définition : elle est obligatoire.

Si votre fonction doit être virtuelle pure, la déclaration est la suivante :

Code : C++

```
struct Exemple
{
    virtual void foo() const = 0;
};
```

Pour s'assurer qu'aucune instruction exécutée dans votre code ne viendra modifier l'état de votre objet au moment de l'appel, tous les attributs utilisés dans votre fonction constante sont considérés comme étant constants. De même, la valeur pointée par le pointeur **this** sera aussi constante. En fait, c'est parce que votre objet (pointé par **this**) est considéré comme étant constant que tous ses sous-objets (ses attributs, que vous utilisez dans votre fonction) seront constants.

Code : C++

```
struct Exemple
{
    void foo() const
    {
        mon_int = 0; //Erreur, mon_int est constant.
    }

    int mon_int;
};
```

Fort heureusement, ceci ne s'applique qu'à l'objet sur lequel est appelée la fonction, et non aux paramètres, même s'ils sont du même type.

Code : C++

```
struct Exemple
{
    void foo(Exemple& exemple) const
    {
        exemple.mon_int = 0; //Ok, mon_int appartenant à exemple
n'est pas constant.
    }

    int mon_int;
};
```

L'intérêt de la const-correctness, enfin !

Une optimisation bien utile : le passage de paramètres par référence constante

Je vous parlais d'une optimisation en introduction, nous y voilà. Imaginez un objet très lourd en mémoire. Imaginez maintenant que vous deviez le passer en paramètre d'une fonction qui ne le modifiera pas (donc pas besoin de pointeurs/références à priori). Une implémentation naïve de cette fonction pourrait être :

Code : C++

```
void maFonction(UneClasseAvecBeaucoupDeDonnees unObjetTresLourd)
{
    //Utilisation d' unObjetTresLourd
}
```

Ici, l'objet est copié en mémoire, on va se retrouver avec deux instances de notre classe très lourde alors que finalement, si nous n'avions pas écrit de fonction et mis le code directement dans le main, nous n'aurions eu besoin que d'un objet.

Et si nous voulions à l'intérieur de notre fonction passer cet objet à une autre fonction ? L'objet en question serait à nouveau dupliqué, multipliant ainsi les instances inutiles et consommant beaucoup plus de ressources que nécessaires.

C'est pour échapper à ce problème que l'on passe les objets par référence. Le principe est simplement de remplacer notre paramètre par une référence (ou un pointeur, mais c'est moins pratique) pour éviter la copie de l'objet. Notre prototype deviendrait donc :

Code : C++

```
void maFonction(const UneClasseAvecBeaucoupDeDonnees&
```

```
unObjetTresLourd);
```

Au lieu de copier l'intégralité de l'objet, on se contente de créer une référence (très légère) qui pointe dessus et qui s'utilisera de la même façon. Bien entendu, comme notre objet n'est pas sensé être modifié par la fonction nous déclarons notre référence comme étant constante.
De manière générale, on préfèrera toujours passer un objet par référence constante plutôt que par copie, même si le gain semble minime. C'est l'exception qui confirme la fameuse First Rule of Program Optimization (Don't do it !).

Attention, tout cela ne s'applique qu'aux objets, c'est-à-dire aux instances d'une classe, et non aux types fondamentaux (int, char, double...) !
Ceux-là sont déjà suffisamment légers pour que l'on se permette de les passer systématiquement par copie.
Ne m'écrivez surtout pas une horreur comme `void maFonction(const bool&);`

Notons deux avantages supplémentaires au passage par référence : premièrement l'usage d'une référence vous permet de tirer parti du polymorphisme d'inclusion et secondement il vous autorise à passer en paramètre de vos fonctions des objets à sémantiques d'entité qui sont par définition non copiables.
Maintenant vous n'avez plus aucune excuse pour ne pas implémenter vos sémantiques correctement 

Références contre références constantes, un choix pas si anodin

Utiliser des références pour alléger les appels de fonction c'est bien beau, mais certains trouveront encore à me dire que sans les const qui se promènent le prototype serait plus lisible. Un commentaire indiquant qu'on ne touchera pas à l'état de l'objet pointé et hop on aura un code plus clair et tout aussi efficace.

J'aimerai que cela soit aussi simple, malheureusement si vous tentez l'expérience, vous allez vite vous retrouver confronté à des ennuis.
Faisons l'essai avec un cas récurrent, celui de la surcharge des opérateurs. Reprenons par exemple une classe définie dans le tuto sur la surcharge des opérateurs :

Code : C++

```
class Duree
{
    public:
        explicit Duree(int heures = 0, int minutes = 0, int secondes
= 0) : my_time(heures*3600 + minutes*60 + secondes) {}

    private:
        friend Duree operator+(Duree&,Duree&); //Essayons avec des
références non constantes
        unsigned int my_time;
};
```

Je l'ai un peu simplifiée mais elle joue le même rôle.

Définissons maintenant l'opérateur d'addition et testons voir s'il fonctionne.

Code : C++

```
Duree operator+(Duree& lhs, Duree& rhs) //Normalement on ne définit
pas exactement cet opérateur de cette manière mais je me le permet
ici par souci de simplicité.
{
    Duree tmp;
    tmp.my_time = lhs.my_time + rhs.my_time;
    return tmp;
}

int main()
{
    Duree d1(1,56,10), d2(2,3,50);
```

```
    Duree d3 = d1+d2;
}
```

Jusqu'ici pas de problème, mais le gros intérêt de la surcharge des opérateurs c'est de pouvoir chainer les opérations. Essayons :

Code : C++

```
Duree d4 = d1 + d2 + d3;
```

Et là, c'est le drame. Le compilateur nous sort une erreur incompréhensible du genre :

error: no match for 'operator+' in 'operator+((* & d1), (* & d2)) + d3'|

Que s'est-t-il passé ? Eh bien, cher ami zéro, nous venons d'avoir la preuve que **const** n'est pas qu'un accessoire pour le développeur C++.

La raison de notre erreur est la suivante : une référence non constante ne peux pas être initialisée par un objet temporaire. Or c'est précisément ce que doit renvoyer l'opérateur d'addition.
Dans notre cas, le résultat de d1 + d2 est calculé en premier et l'objet récupéré, qui est temporaire (appelons le d5) est passé en paramètre de l'opération suivante : d5 + d3.
Nous n'avons d'autre choix que d'ajouter **const** pour transformer nos références en références constantes qui, elles, peuvent être initialisées avec un objet temporaire.

Compilez (n'oubliez pas de changer la déclaration friend dans la classe), ça marche, notre problème est résolu et à l'avenir vous utiliserez des références constantes dans vos prototypes  -

const fait des vagues

Supposons, et c'est légitime, que nous voulions afficher notre classe Duree. Ajoutons donc une fonction membre comme le ferait la plupart des zéros :

Code : C++

```
class Duree
{
    public:
        explicit Duree(int heures = 0, int minutes = 0, int secondes
= 0) : my_time(heures*3600 + minutes*60 + secondes) {}
        void afficher() //Être SRP-correct ou ne pas l'être ? --
private-joke
        {
            std::cout << my_time / 3600 << "h" << my_time % 3600 /
60 << "m" << my_time % 60 << "s";
        }

    private:
        friend Duree operator+(const Duree&,const Duree&);
        unsigned int my_time;
};
```

Notre classe devrait maintenant ressembler à cela.

Essayons maintenant de créer une fonction tout ce qu'il y a de plus normale, prenant un objet de type Duree en paramètre et qui appellerait finalement la fonction Duree::afficher().

Code : C++

```
void maFonction(const Duree& duree) //Vous aviez pensé à la
référence constante n'est-ce pas ? ;)
{
    //Du code ...
    duree.afficher();
```

```
}
```

Comme vous pouvez vous en douter, cela ne va pas fonctionner. L'objet duree est constant, on ne peut pas appeler une de ses fonctions membres comme cela.

La réponse à ce problème est évidemment notre mot-clé favori : **const**. Vous vous souvenez des fonctions constantes ? Le mot-clé const à la fin du prototype nous indique que la fonction ne modifie pas l'objet qui l'appelle, et cette information est valable également pour le compilateur !
Ce dernier sait dans ce cas que la fonction est inoffensive et peut en accorder l'utilisation par un objet constant.

Corrigeons donc notre fonction et compilons :

Code : C++

```
class Duree
{
    public:
        explicit Duree(int heures = 0, int minutes = 0, int secondes
= 0) : my_time(heures*3600 + minutes*60 + secondes) {}
        void afficher() const
        {
            std::cout << my_time / 3600 << "h" << my_time % 3600 /
60 << "m" << my_time % 60 << "s";
        }

    private:
        friend Duree operator+(const Duree&,const Duree&);
        unsigned int my_time;
};

void maFonction(const Duree& duree)
{
    duree.afficher();
}
```

Ça fonctionne, nous avons surmonté la dernière difficulté (si l'on peut dire) introduite par la const-correctness  .

Vous avez pu le remarquer à travers ces exemples, ajouter **const** en un point de votre code a des répercussions sur le tout le reste, c'est pourquoi il est important de commencer à produire du code const-correct dès le début d'un projet. Dans le cas contraire vous allez devoir reprendre une grande partie de ce qui a déjà été fait et vous heurter à de nombreuses erreurs de compilation qui vont se déclarer en cascade et vous noyer sous la masse de corrections à apporter (c'est du vécu). Je ne le répèterais jamais assez, utilisez le mot-clé **const** autant que possible.

Allons plus loin dans la constance !

Arrivés à ce stade, nous avons déjà vu l'essentiel de ce qui touche à la const-correctness. Normalement vous devriez pouvoir écrire un code const-correct qui fera l'admiration de vos proches (ne rêvons pas trop non plus). Pourtant nous n'avons pas fini de parler de ce détail que vous avez pu trouver parfaitement anodin à vos débuts.

Utilisez const autant que possible, mais pas trop.

Comme toutes les bonnes choses (à part le Logiciel Libre évidemment ), il ne faut pas abuser de **const** !
Voici un exemple relativement fréquent d'un emploi absolument inutile de ce mot-clé :

Code : C++

```
void maFonction(const int);
```

Même si nous n'allons pas modifier le paramètre dans la fonction, l'utilisateur n'en a que faire, puisque vous ne récupérez qu'une copie. Dans ce cas, vous avez pris la peine d'écrire 5 caractères de trop, qui alourdissent votre prototype et n'apportent rien sémantiquement.
En bref, pour les types fondamentaux contentez-vous d'une copie non constante (si vous avez pensé « il manque la référence »

c'est qu'il faut que vous relisiez la note à la fin du paragraphe sur le passage par référence constante).

Autre emploi contestable de const :

Code : C++

```
class Exemple
{
    public :
        const int get() const {return my_int;}
    private :
        int my_int;

};
```

Bien entendu la fonction doit être constante, seul le retour est concerné. Ici l'intérêt du `const` est également nul. Le retour n'est pas une référence ni un pointeur, donc l'utilisateur ne pourrait de toutes façons pas modifier l'état de l'objet (mon compilateur a même la gentillesse de me prévenir que le qualificateur n'a aucun effet dans ce cas).

La situation est presque identique dans le cas d'objets, à ceci près qu'un retour constant permet d'éviter l'absurdité sémantique suivante :

Code : C++

```
class Exemple
{
    public :
        std::string get() const {return my_str;}
    private :
        std::string my_str;

};

int main()
{
    Exemple e;
    e.get() = "gné ?";
}
```

Ce code compile sans problème. A vous de juger si cela vaut la peine d'ajouter un `const`.

Dernier cas, peut-être plus problématique.

Code : C++

```
class Exemple
{
    public :
        const std::string& get() const {return my_str;}
    private :
        std::string my_str;
};

//Ou encore
const Exemple& foo(const Exemple& e)
{
    //Du code
    return e;
}
```

Je vous ai promis de ne pas rentrer dans des détails d'optimisation, alors pour faire simple : renvoyez une copie, personne ne vous en voudra. De toutes façons le compilateur optimise le retour la majeure partie du temps, alors n'alourdissez pas le prototype de vos fonctions inutilement.

Un cas dangereux (et donc à bannir) :

Code : C++

```
const Exemple& foo()
{
    return Exemple();
}
```

Ici votre compilateur devrait vous avertir : vous renvoyez une référence constante sur un objet qui est détruit à la fin de la portée (oui, ce n'est pas la variable créée par l'instruction Exemple() qui est renvoyée, seulement une référence dessus). Autrement dit vous allez manipuler un objet qui n'existe plus. Je vous laisse imaginer les conséquences.

Constance et typage

Vous l'avez peut-être deviné, je vous le confirme : utiliser **const** modifie le type de vos variables.
Plus clairement, pour un type `T` donné, **const** `T` est d'un type différent.
Mais quelque chose devrait alors vous interpeler. Reprenons un des premiers exemples de cet article :

Code : C++

```
int i = 0, j = 1;
int const * p = &i;

*p = j;
p = &j;
```

Ici nous faisons pointer notre pointeur sur un `int` constant sur un `int`... non constant.

C'est une opération que vous avez sans doute fait sans vous apercevoir, mais si le typage du C++ était plus fort, cette opération ne serait pas acceptée.
En C++ le transtypage d'un pointeur sur un type non constant à un pointeur sur un type constant est implicite, ce qui veut dire que vous n'avez pas à vous en préoccuper. En revanche l'opération inverse n'est pas permise ! Il est impossible d'initialiser directement un pointeur ou une référence sur un type non constant avec un pointeur sur un type constant. Dans le cas inverse vous pourriez modifier l'objet constant comme s'il ne l'était pas. 

Dans ce cas, pourquoi peut-on le faire lorsqu'il s'agit de types non composés ? Tout simplement parce qu'une copie est créée dans ce cas. En réalité peu importe que la variable à copier soit ou non constante, elle n'est qu'un modèle et ne sera pas modifiée. Elle peut donc être systématiquement considérée comme étant constante, ce qui est le type le plus restrictif.

Une question pour vous : quelle est la condition pour que soit correcte la surcharge d'une fonction ?
La réponse est : que les types de paramètres soient différents.
On pourrait donc s'attendre à pouvoir surcharger une fonction attendant une variable d'un type T non constant avec une version attendant un T constant. Et bien non. En fait, la surcharge n'est possible que dans le cas de types composés (pointeurs, références, tableaux...).

Code : C++

```
void foo(const std::string&){std::cout << "Je prends une référence
constante en paramètre" << std::endl;}
void foo(std::string&)      {std::cout << "Je prends une référence
non constante en paramètre" << std::endl;}
//Essayez de retirer les références : si ça compile, changez de
compilateur.

int main()
{
    std::string ex = "";
    foo("ex");
    foo(ex);
}
```

Comment le compilateur procède-t-il pour déterminer quelle fonction sera appelée ? La règle est simple : si cela est possible, la version de la fonction prenant en paramètre une référence non constante sera appelée. Dans le cas contraire c'est la version prenant une référence constante qui sera appelée.

Dans notre cas :

Code : Console

```
Je prends une référence constante en paramètre
Je prends une référence non constante en paramètre
```

Notez qu'il n'est pas possible de jouer avec la constance des pointeurs, seulement avec le type pointé. Autrement dit, dans le cas d'une surcharge, un pointeur constant sur T (n'importe quel type) sera considéré équivalent à un pointeur non constant sur T.

Autre conséquence de cette différence de types : une erreur peut survenir lorsque vous utilisez des classes templates. Par exemple, un test d'égalité de types statique (qui vérifie à la compilation si deux types sont égaux, si si ça peut avoir de l'intérêt) échouerait si les paramètres passés étaient `int` et `const int`.
Je vous le dit parce que cela m'est déjà arrivé mais c'est assez rare, rassurez-vous. 
Si jamais vous ne trouvez pas l'origine d'une erreur de compilation en utilisant une classe template, vérifiez la const-correctness de votre code.

En revanche, toujours avec les templates, il y a une erreur dont vous n'avez pas à vous soucier, c'est l'accumulation de `const` :

Code : C++

```
template<typename T>
void foo(const T);

int main()
{
    const int i;
    foo(i);
}
```

Comme nous l'avons dit précédemment, nous sommes autorisés à accumuler autant de `const` que nous voulons sur un seul type, le résultat sera toujours le même : un type constant.

Dernière petite chose, en manipulant des paramètres templates vous pouvez recevoir des classes comme des types fondamentaux.
Quid alors d'un passage de paramètres dont le type dépends d'un template ?
Choisirez-vous le passage par référence constante ou par copie ?
En règle général mon conseil est de choisir la référence constante. Vous n'êtes jamais à l'abri de l'utilisation d'une classe extrêmement lourde, et le passage d'un `bool` par référence constante sera toujours moins dommageable que la copie d'un `std::array<int, 10000>`.

Si vous désirez quand même faire un choix plus précis, sachez qu'une bibliothèque de boost (call_traits) le fait pour vous, tout en réglant quelques autres menus problèmes. Mais souvenez-vous de ce que Donald Knuth vous dirait : « Early optimization is the root of much evil ».

Quelques détails sur les objets constants

Revenons si vous le voulez bien sur le cas des objets constants. Il y a quelques petits détails que je voudrais aborder. Ceux-ci n'ont pas une grande importance mais tant qu'à faire essayons d'être exhaustifs sur le sujet.

Comme vous le savez, un objet dont la classe ne définit aucun constructeur est si possible construit à l'aide d'un constructeur trivial par défaut lorsqu'on l'instancie. Ceci n'est pas le cas pour un objet constant.

Code : C++

```
struct Exemple
{
};

int main()
{
    const Exemple e; //Erreur ! Exemple n'a pas de constructeur
    Exemple f; //Ok, appel au constructeur trivial
}
```

La raison à cela est qu'une variable constante doit obligatoirement être initialisée à la construction. Or, dans ce cas-là, l'appel au constructeur par défaut n'est pas une initialisation. D'ailleurs, si vous rajoutez des attributs à notre classe Exemple, vous verrez que le constructeur par défaut ne les initialise pas.

Pour conclure, retenez qu'il faut toujours initialiser explicitement une valeur constante, à moins qu'il ne s'agisse d'une instance d'une classe pour laquelle est défini un constructeur par défaut.

Les objets constants doivent donc nécessairement être initialisés à la construction. Cela vaut aussi pour les attributs constants, et la seule manière que nous avons d'initialiser un sous-objet à sa construction, c'est dans la liste d'initialisation de l'objet englobant :

Code : C++

```
struct Exemple
{
    const int i;
    Exemple(int x) : i(x) //Seul moyen d'initialiser un attribut constant
    {
    }
};
```

Petite exception : les attributs statiques constants. Comme il est impossible de passer par la liste d'initialisation, il va falloir recourir à une autre syntaxe.

Code : C++

```
struct Exemple
{
    static const int mon_attribut_statique;
};

const int Exemple::mon_attribut_statique = 0;
```

Vous connaissez sans doute cette syntaxe, elle est identique à celle permettant d'initialiser un attribut statique non constant. Seulement cette fois vous ne pouvez pas vous en passer.

Deuxième point à aborder : vous vous souvenez qu'un objet constant étend sa constance à tous ses sous-objets. Ceci ne vaut pas pour les objets pointés par les pointeurs ou références membres. Un exemple sera sans doute plus parlant :

Code : C++

```
struct Exemple
{
    int* ptr; //Cela vaut aussi pour une référence
    void foo() const {*ptr = 0;} //OK...
};
```

Cela est un peu contre-intuitif, je vous l'accorde, mais pas complètement absurde. Le but d'un pointeur est en effet de référencer une autre variable. Cette dernière n'appartient donc pas à l'objet qui contient le pointeur, et la constance ne s'étend pas jusqu'à elle.

Dernière chose, bien utile pour les conteneurs : il est possible de surcharger une fonction membre en proposant une version constante et une version non constante prenant des paramètres de même type ! Voyons plutôt :

Code : C++

```
#include <string>
#include <iostream>

class Exemple
```

```
{
    public :
        int get() const
        {
            return mon_attribut;
        }

        int& get() //Je vous rappelle que le type de retour n'est
pas discriminant pour déterminer si la surcharge est possible.
        {
            return mon_attribut;
        }

    private :
        int mon_attribut;
};

void afficher(const Exemple& e)
{
    std::cout << e.get() << std::endl;
    // Notez que dans cette fonction, la ligne suivante ne
compilerait pas, bien que la variable renvoyée par get soit une
copie :
    // e.get() = 1; // Erreur
    //
    // En revanche, comme indiqué précédement, si get renvoyait un
std::string, ceci compilerait
    // e.get() = "gné"; // Ok
    //
    // Toutefois, e.mon_attribut ne serait pas modifié.
}

int main()
{
    Exemple exemple; //Mon objet n'est pas initialisé souvenez-vous.
mon_attribut peut potentiellement avoir n'importe quelle valeur.
    exemple.get() = 1; //Syntaxe étrange, mais remplacez get par
l'opérateur d'indexation (operator[])... Ça y est, vous y êtes ?
    afficher(exemple);
}
```

En pratique, je vous déconseille fortement ce type de code (renvoyer une référence sur un attribut) qui brise complètement l'encapsulation de votre classe. Notez comme pour la surcharge des paramètres que le compilateur appellera en priorité la version non constante de la fonction.
Je pense que vous saisissez l'intérêt de cette surcharge : vous pouvez ainsi proposer des services différents selon que l'objet sur lequel cette fonction sera appelé est ou non constant. Exemple : un accès en lecture dans le cas d'un conteneur constant (c'est donc quelque chose qu'il faudra systématiquement faire dans ce cas).

const_cast et mutable : altérer la constance d'une variable

const_cast : les origines du mal

Attention ! Utiliser **`const_cast`** c'est **mal**.

Je caricature un peu, mais c'est pour que vous compreniez bien qu'utiliser **`const_cast`** ne se fait pas à la légère. Il n'y a en effet que très peu d'occasions où cela est justifié.

Commençons par le commencement. **`const_cast`** est un opérateur de conversion qui permet de modifier la cv-qualification d'une variable.

La sivi quoi ?

En C++ les mots-clés **const** et **volatile** sont appelés des **cv-qualifiers**. Nous n'entrerons pas dans le détail de l'utilité de **volatile**, sachez simplement que d'un point de vue syntaxique, ce qualificateur se comporte exactement comme **const**. Tout type faisant intervenir un de ces mots-clés est dit **cv-qualified**. Plus précisément, il peut être **const-qualified**, **volatile-qualified** ou même **const-volatile-qualified**. S'il ne rentre dans aucune de ces catégories, c'est qu'il est **cv-unqualified**.

Généralisons ce que nous avons vu à propos du typage : un type cv-qualified est différent d'un cv-unqualified et il est même différent d'un type qui n'est pas cv-qualifié de la même façon. La conséquence en est qu'un pointeur ou une référence sur un type d'une certaine cv-qualification ne peut pointer sur un type doté d'une cv-qualification différente (à l'exception près de la conversion implicite indiquée précédemment).

Ça va vous suivez ? Si les termes techniques vous embrouillent, ne vous inquiétez pas, il n'est pas absolument nécessaire de retenir tout cela. Je vous le donne surtout à titre informatif, et pour que nous parlions la même langue dans la suite du tutoriel.

const_cast permet d'outrepasser la limitation citée précédemment en transformant un pointeur (ou une référence) sur un type cv-qualified en un pointeur sur un type qui n'a pas la même cv-qualification (**cv_cast** aurait été un nom plus approprié).

Pourquoi est-ce mal ? **const_cast** permet d'outrepasser la constance d'une variable, or le mot-clé **const** n'est pas là pour faire joli. Si quelqu'un déclare une valeur constante c'est pour une bonne raison. La modifier c'est prendre le risque d'altérer significativement le fonctionnement de l'application. Si quelqu'un vous passe un objet par référence constante, il s'attend à ce que celui-ci ne soit pas modifié. Si vous le faites, c'est l'utilisateur de votre fonction qui va se retrouver avec des erreurs dont il ne pourra déterminer la source.

La syntaxe de l'opérateur pour un type T est celle-ci :

Code : C++

```
const T i;
T* j = const_cast<T*>(&i);
T& k = const_cast<T&>(i);
```

Si vous utilisez un type POD, le compilateur va sans doute optimiser sauvagement votre code et remplacer à la compilation la valeur contenue dans i là où la variable a été utilisée. Du coup vous ne vous retrouverez pas avec le résultat attendu (ici i ne sera pas réellement modifiée, ouf). En revanche ça ne sera pas le cas si vous utilisez un type non POD, et là tous les excès sont permis.
Rappelons au cas où que l'opération inverse est transparente et ne justifie pas l'usage de **const_cast**.

Il n'y a que quelques cas dans lequel l'usage de **const_cast** est justifié. L'un, rare, est celui où, produisant du code const-correct (ce que vous allez faire maintenant n'est-ce pas ?) vous devez utiliser une bibliothèque qui n'est pas const-correct et qui vous demande des choses aussi absurdes que de lui passer des pointeurs ou références sur des variables non constantes qu'elle ne va pas modifier.
Mon conseil personnel serait plutôt de vous orienter vers un code véritablement écrit en C++ mais on n'a pas toujours le choix.

Un autre cas, plus raisonnable est celui de l'écriture de deux versions de la même fonction : constante et non constante. Ici, **const_cast** peut être utilisé de manière propre (c'est-à-dire dans un cadre limité et sans risque) pour éviter la duplication de code (réécrire deux fois la même fonction).

Code : C++

```
struct Exemple
{
    public :
        Exemple(int i) : my_int(i) {}
        inline int get() const {return const_cast<Exemple*>(this)->get();}
        inline int& get() {return my_int;}

    private :
        int my_int;
};
```

Dans cet exemple, le **const_cast** ne posera jamais problème puisque nous sommes certains que l'état de l'objet ne sera pas modifié dans le processus. La référence sur l'attribut est en effet copiée avant d'être renvoyée à l'utilisateur de la classe.

En un mot comme en cent, faites très attention à ce que vous faites avec **const_cast** (en tout cas ne l'utilisez pas pour outrepasser la constance d'une variable).

mutable : un objet constant dont l'état varie ?

Le mot-clé **mutable** permet de définir une variable membre d'une classe dont la valeur pourra changer même si l'objet qui le contient est constant. A priori l'intérêt peut sembler nul, et le mot-clé à reléguer aux oubliettes avec **const_cast**. Ce n'est pas tout à fait exact, mais il est vrai que l'emploi de mutable doit être l'exception et jamais la règle.

Un peu de vocabulaire : le standard définit le mot-clé **mutable** sous le terme de **storage-class-specifier** (ce que l'on peut traduire littéralement par « spécificateur de classe de stockage »).
Il ne peut être appliqué qu'à une variable non constante et sans autre spécificateur de stockage (qui sont **static**, **thread_local**, **extern** et **register**).
Dans le cas d'un type composé, il s'applique là où cela est possible. En revanche il ne peut être utilisé pour qualifier un pointeur constant, même pointant sur une variable non constante.

Pour la syntaxe, mettez-le en début de déclaration. Contrairement à **const** sa position n'a pas d'importance.

Un petit exemple pour résumer tout ça.

Code : C++

```
struct Exemple
{
    mutable int i;              //Ok
    mutable int* p;             //Ok, le pointeur comme l'entier
pointé seront mutables
    mutable const int* q;       //Ok, seul le pointeur est mutable
    mutable const int* const r; //Erreur
    mutable int* const s;       //Erreur
    mutable int& t;             //Ok
    mutable const int& u;       //Passe sur mon compilateur mais
absurde
};
```

Très bien, mais quel est l'intérêt de **mutable** ?

En fait, le C++ privilégie la constance sémantique (le sens du code) des variables plutôt que leur constance physique (la représentation en mémoire de la variable).
Un objet constant doit apparaître immuable pour l'utilisateur mais il ne l'est pas nécessairement. Un exemple fréquent lorsque l'on aborde ce mot-clé est celui du caching.
Supposons qu'un de vos objets ai une fonction constante renvoyant le résultat d'un calcul assez lourd. Pour limiter un peu les dégâts, vous aimeriez pouvoir ne refaire ce calcul que lorsque ses paramètres sont modifiés mais sans perdre le caractère constant de votre fonction. Cela est possible en utilisant une variable **mutable** dans laquelle vous stockerez le résultat calculé. Pour l'utilisateur, le changement est transparent, seule la vitesse d'exécution peut varier.

D'autres applications de mutable peuvent être le comptage de références et la synchronisation de données.

Souvenez-vous toutefois que l'utilisation de mutable n'est pas souvent justifiée. N'y recourrez que lorsque vous êtes certains que cela ne vient pas briser la const-correctness de votre code en autorisant des objets constants à utiliser des fonctions et des attributs auxquels il ne devrait pas recourir.
Nous voilà au terme de cet article. J'espère que vous en avez apprécié la lecture et qu'il vous aura convaincu de l'intérêt de la const-correctness tout en vous donnant les outils pour la mettre en place facilement. N'oubliez pas le plus important : habituez-vous à utiliser const au maximum sans que cela ne devienne absurde et vous écrirez rapidement du code const-correct sans vous en apercevoir  .

Si vous relevez une erreur, une imprécision, un manque ou tout simplement que vous avez un commentaire à faire ou une question à poser sur ce que j'ai écrit, n'hésitez pas à m'en faire part.

Je remercie à ce propos Freedom, lmghs et boli pour leur relecture attentive et leurs remarques pertinentes sur l'ensemble du tutoriel.

Sources :
http://en.wikipedia.org/wiki/Const-correctness
http://www.parashift.com/c++-faq-lite/ [...] rectness.html
http://www.gotw.ca/gotw/006.htm (Voir également le #81 cité en introduction pour les optimisations liés à const)
Et bien entendu le C++ International Standard !