

const et les classes

J.-C. Chappelier, J. Sam

version 1.1 d'avril 2020

Le mot réservé « `const` » se retrouve dans plusieurs contextes, en particulier lorsqu'on a affaire à des classes; ce qui est parfois source de confusions. Essayons de clarifier ses emplois.

1 `const` et les variables

Avant tout, quelques rappels hors programmation « orientée objet » :

1. le mot réservé « `const` » qualifie un nom de variable pour indiquer qu'*au travers de ce nom*, la valeur ne peut pas être modifiée¹;
2. `const` s'applique toujours à ce qui le précède; sauf s'il n'y a rien devant, auquel cas il s'applique à ce qui suit.

Pour le point 1. :

```
int const i(3);
```

empêche de modifier la valeur de `i` : le code suivant ne compilera pas :

```
i = 5;
```

Pour l'aspect « *au travers de ce nom* » du point 1. :

```
int const i(3);
```

```
int* ptr(&i);
```

`ptr` pointe sur `i`, mais sans être lui-même `const`.

(Attention ! C'est justement de la *mauvaise* programmation !... mais c'est possible, moyennant quelques ajouts syntaxiques que nous ne détaillerons justement pas ici.)

Alors

```
i = 5;
```

ne compilera toujours pas, mais par contre

```
*ptr = 5;
```

est tout à fait possible et pourrait¹ donc changer la valeur de `i` ! Le `const` sur `i` ne veut donc pas dire que la valeur de `i` ne peut pas changer dans l'absolu, mais bien qu'elle ne peut pas changer *au travers du nom* `i`.

1. A noter cependant qu'en C++, contrairement au C, une variable « `const` » *peut* être considérée comme constante par le compilateur (et optimisée en conséquence).

Pour le second point : on peut tout aussi bien écrire

```
int const i(3);
```

que

```
const int i(3);
```

Dans les deux cas, c'est bien l'int qui est const.

La situation se complique avec les pointeurs/références :

```
const int* ptr1;  
int const* ptr2;  
int* const ptr3(&i);
```

Qui est quoi ?

`ptr1` est un pointeur sur un « `const int` », exactement ce qu'aurait du être `ptr` dans l'exemple ci-dessus pour ne pas pouvoir modifier la valeur de `i` : au travers de `ptr1`, on ne peut pas modifier la valeur (de type `int`) pointée.

`ptr2` est exactement de même type que `ptr1`.

`ptr3` par contre est un « pointeur const » sur un `int` : ici c'est l'adresse pointée par `ptr3` qui ne peut pas être modifiée (au travers de `ptr3`) : `ptr3` pointe toujours au même endroit (sur `i` dans cet exemple ; et sous réserve qu'il n'y a pas d'autre accès à `ptr3`, pas de pointeur sur ce pointeur ; -). Par contre, la valeur pointée par `ptr3` peut tout à fait être modifiée.

Et on peut bien sûr vouloir protéger les deux, l'adresse et la valeur pointée :

```
int const * const ptr4(&i);
```

Mais c'est quand même assez rare ; -) ...

2 const et les classes

Dans le cadre de la programmation « orientée objet », il y a quatre utilisation possibles de `const` : trois pour les méthodes et, bien sûr, une pour les variables/attributs.

Commençons par cette dernière, la plus simple : un attribut `i` déclaré

```
const int i;
```

veut simplement dire qu'une fois initialisée (par un constructeur), la valeur de cet attribut ne peut plus être modifiée (au travers de ce nom `i`), même par une méthode de la classe (autre qu'un constructeur).

C'est exactement le même rôle que dans le rappel précédent, à la seule différence que les attributs sont initialisés par les constructeurs et non pas directement. (Un constructeur pourra par contre – et *devra*, même – tout à fait donner une valeur à cet attribut.)

Pour les méthodes, on peut trouver `const` à trois endroits différents :

```
CONST1 type_de_retour methode(CONST2 parametre) CONST3;
```

Par exemple :

```
const int& f(const Point& p) const;
```

Le premier `const` (« CONST1 »), plutôt rare car pas souvent nécessaire, sert à dire que la valeur de retour ne peut pas être affectée/modifiée; on ne peut pas écrire quelque chose comme :

```
obj.f(point) = 33;
```

Le second `const` (« CONST2 ») est tout à fait « classique » et n'est pas liée aux méthodes; il existe aussi de la même façon pour les fonctions usuelles. Il sert à dire que l'argument reçu au travers de ce paramètre ne sera pas modifié, et donc que

1. si l'on fait « `obj.f(point)` », `point` n'est pas modifié
(Dans un passage par valeur, ceci est bien sûr totalement inutile! Cela ne prend vraiment de sens que dans un passage par référence.)
2. la méthode ne peut pas faire des choses comme « `parametre = 33` ».

Le troisième `const` enfin (« CONST3 ») est spécifique aux méthodes. Il sert à dire que l'appel à cette méthode ne modifie pas l'instance courante, i.e. lorsque l'on fait « `obj.f(point)` », on est sûr que `obj` ne sera pas modifié.

Conséquence : si l'on a un

```
const Objet obj(bla1, bla2);
```

alors, bien sûr, on ne peut appeler que des « méthodes `const` » (au sens du CONST3 ci-dessus) sur cet objet !

Sinon l'aspect « `const` » de la déclaration de cet objet ne peut pas être garanti et le compilateur nous dira que l'appel à une méthode non-`const` « *discards qualifier* » : cet appel annulerait (« *discard* ») la *qualification* `const` de l'objet `obj`.