1 Le commentaire de fin de ligne

En C ANSI, un commentaire peut être introduit en n'importe quel endroit où un espace est autorisé[1] en le faisant précéder de /* et suivre de */. Il peut alors éventuellement s'étendre sur plusieurs lignes.

En C++, vous pouvez en outre utiliser des "commentaires de fin de ligne" en introduisant les deux caractères : //. Dans ce cas, tout ce qui est situé entre // et la fin de la ligne est un commentaire. Notez que cette nouvelle possibilité n'apporte qu'un surcroît de confort et de sécurité ; en effet, une ligne telle que :

```
cout << "bonjour\n" ; // formule de politesse
```

peut toujours être écrite ainsi :

```
cout << "bonjour\n" ; /*formule de politesse*/
```

Vous pouvez mêler (volontairement ou non !) les deux formules. Dans ce cas, notez que, dans :

```
/* partie1 // partie2 */ partie3
```

le commentaire "ouvert" par /* ne se termine qu'au prochain */ ; donc *partie1* et *partie2* sont des commentaires, tandis que *partie3* est considéré comme appartenant aux instructions. De même, dans :

```
partie1 // partie2 /* partie3 */ partie4
```

le commentaire introduit par // s'étend jusqu'à la fin de la ligne. Il concerne donc *partie2, partie3 et partie 4.*

Remarques

1 Le commentaire de fin de ligne constitue le seul cas où la fin de ligne joue un rôle significatif, autre que celui d'un simple séparateur.

2 Si l'on utilise systématiquement le commentaire de fin de ligne, on peut alors faire appel à /* et * pour inhiber un ensemble d'instructions (contenant éventuellement des commentaires) en phase de mise au point.

2 Déclarations et initialisations

2.1 Règles générales

C++ s'avère plus souple que le C ANSI en matière de déclarations. Plus précisément, en C++, il n'est plus obligatoire de regrouper au début les déclarations effectuées au sein d'une fonction ou au sein d'un bloc. Celles-ci peuvent être effectuées où bon vous semble, pour peu

1. Donc, en pratique, n'importe où, pourvu qu'on ne "coupe pas en deux" un identificateur ou une chaîne constante.

qu'elles apparaissent avant que l'on en ait besoin : leur portée reste limitée à la partie du bloc ou de la fonction suivant leur déclaration.

Par ailleurs, les expressions utilisées pour initialiser une variable scalaire peuvent être quelconques, alors qu'en C elles ne peuvent faire intervenir que des variables dont la valeur est connue dès l'entrée dans la fonction concernée.

Voici un exemple incorrect en C ANSI et accepté en C++ :

```
main()
{
   int n ;
     .....
   n = ...
     .....
   int q = 2*n - 1 ;
     .....
}
```

La déclaration tardive de *q* permet de l'initialiser[1] avec une expression dont la valeur n'était pas connue lors de l'entrée dans la fonction (ici *main*).

2.2 Cas des instructions structurées

L'instruction suivante est acceptée en C++ :

```
for (int i=0 ; ... ; ...)
{
.....
}
```

Là encore, la variable *i* a été déclarée seulement au moment où l'on en avait besoin. Sa portée est, d'après la norme, limitée au bloc régi par l'instruction *for*. On notera qu'il n'existe aucune façon d'écrire des instructions équivalentes en C.

Cette possibilité s'applique à toutes les instructions structurées, c'est-à-dire aux instructions *for*, *switch*, *while* et *do...while*.

Remarque

Le rôle de ces déclarations à l'intérieur d'instructions structurées n'a été fixé que tardivement par la norme ANSI. Dans les versions antérieures, ce genre de déclaration était certes autorisé, mais tout se passait comme si elle figurait à l'extérieur du bloc ; ainsi, l'exemple précédent était interprété comme si l'on avait écrit :

```
int i ;
for (i=0 ; ... ; ... )
{
   .....
}
```

1. N'oubliez pas qu'en C (comme en C++) il est possible d'initialiser une variable automatique scalaire à l'aide d'une expression quelconque.

3 La notion de référence

En C, les arguments et la valeur de retour d'une fonction sont transmis par valeur. Pour simuler en quelque sorte ce qui se nomme "transmission par adresse" dans d'autres langages, il est alors nécessaire de "jongler" avec les pointeurs (la transmission se fait toujours par valeur mais, dans ce cas, il s'agit de la valeur d'un pointeur). En C++, le principal intérêt de la notion de référence est qu'elle permet de laisser le compilateur mettre en œuvre les "bonnes instructions" pour assurer un transfert par adresse. Pour mieux vous en faire saisir l'intérêt, nous vous proposons de vous rappeler comment il fallait procéder en C.

3.1 Transmission des arguments en C

Exemple 1

Considérons l'exemple suivant, qui illustre le fait qu'en C les arguments sont toujours transmis par valeur :

```
#include <iostream>
using namespace std ;
main()
{
       void echange (int, int) ;
       int n=10, p=20 ;
       cout << "avant appel :   " << n << " " << p << "\n" ;
       echange (n, p) ;
       cout << "apres appel :   " << n << " " << p << "\n" ;
}

void echange (int a, int b)
{
       int c ;
       cout << "debut echange : " << a << " " << b << "\n" ;
       c = a ; a = b ; b = c ;
       cout << "fin echange   : " << a << " " << b << "\n" ;
}
```

```
avant appel :   10 20
debut echange : 10 20
fin echange   : 20 10
apres appel :   10 20
```

Les arguments sont transmis par valeur

Lors de l'appel d'*echange*, il y a transmission des valeurs de *n* et de *p* ; on peut considérer que la fonction les a recopiées dans des emplacements locaux, correspondant à ses arguments formels *a* et *b*, et qu'elle a effectivement "travaillé" sur ces copies.