Chapitre 4

Spécificités C++

1. Commentaires

1.1 Comme en C

Pour couvrir toute une zone (une ou plusieurs lignes à la fois)

```
/* quelque chose à mettre en commentaire
     sur plusieurs lignes …
     La suite de ce commentaire! */
```

Sinon

```
#ifdef un_commentaire
     quelque chose à mettre en commentaire
     sur plusieurs lignes aussi!
#endif
```

- Le symbole « # » est traité par le préprocesseur.
- L'instruction « ifdef » signifie que si la variable « un_commentaire » a été définie alors on exécute le contenu du bloc limité par « ifdef » et « endif ».

- On définie la variable « un_commentaire » dans le fichier avant son utilisation sinon dans la ligne de commandes du compilateur.

1.2 À la sauce C++ et Java

Pour couvrir une seule ligne

```
// Quelque chose à mettre en commentaire que sur une ligne!
```

1.3 Imbrication de commentaires

Attention à ces quelques dangereuses situations

```
if (b>a) {
     // temp = a; //swap de a & b
     // etc.
}
```

Commentaire inutile mais OK !

Par contre, dans l'exemple suivant, c'est une belle pagaille! Des commentaires imbriqués qui vont générer une erreur à la compilation.


```
if (b>a) {
     /* temp = a; /* swap de a & b */
     etc. */
}
```

L'erreur est à ce niveau. Les caractères **`*/`** vont se retrouver seuls d'où l'erreur de compilation!

C'est l'équivalent de **`/* temp = a; swap de a & b */`**

On peut imbriquer les deux styles de commentaires :

- Entre **`/*`** et **`*/`** le style **`//`** n'a pas une signification spéciale.
- Dans **`//`** le style de commentaires **`/*`** et **`*/`** n'ont pas de signification spéciale.

Trop de commentaires tue le commentaire! Pas besoin d'en rajouter !

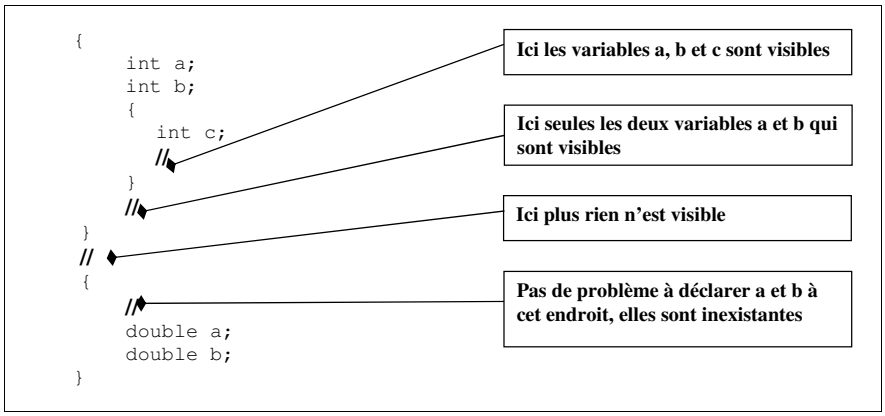
```
// Boucle for !
for (int i=0 ; i<10 ;i++) {

}
```

Commentaire inutile!
On s'est bien que c'est une boucle for !

Dans le précédent exemple, il était plus judicieux de nous dire par exemple à quoi peut-elle servir cette boucle for.

2. Portée d'une variable

- En C++ il y a plusieurs catégories de visibilité : fichier, fonction, bloc, classe, prototype de la fonction, espace de nom.
- Visibilité au niveau du fichier : la portée de la variable s'étend de l'endroit où elle a été déclarée dans le fichier jusqu'à la fin de ce fichier. Cette possibilité n'est offerte que dans les langages C et C++ puisque Java est un langage pur objet. La variable est appelée dans ce cas « variable globale » car sa portée est globale.
- Visibilité bloc : la durée de vie d'une variable est celle du bloc où elle a été déclarée sauf dans le cas d'une variable globale.



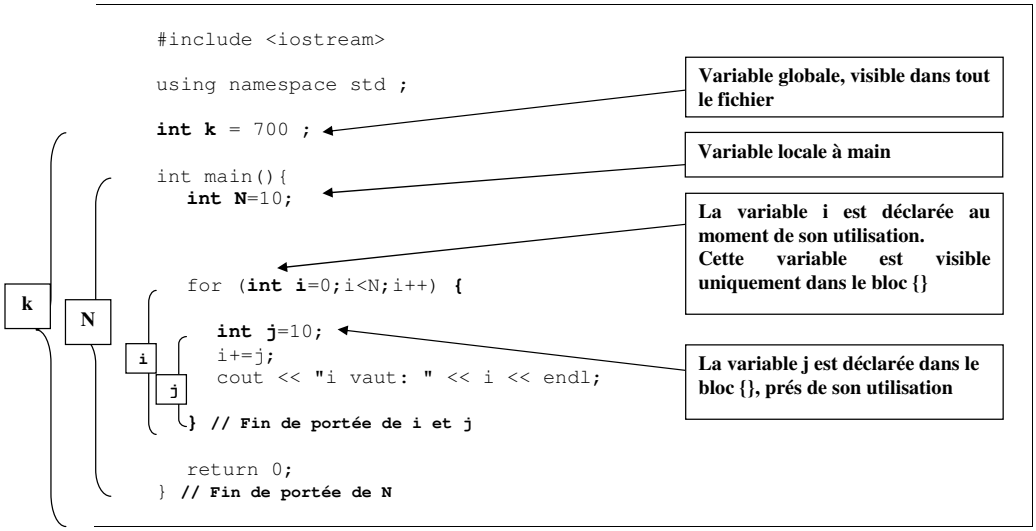
Attention aussi aux déclarations dans les instructions de contrôle

```
if (int x=fonction()) {traitement(x);} ← OK

while(int x==1) {x=fonction();} ← OK
```

(La déclaration est valable aussi pour un `switch`, à faire par vous).

```
do {x=fonction();} while(int x==1) ← FAUX
```

La variable `x` a été déclarée après son utilisation.



3. Généralisation de la résolution de portée

```
#include <iostream>

using namespace std;

int main() {

   int i=500;

   cout << "i du main vaut avant: " << i << endl;

   for (int i=0; i< 1; i++)
         cout << "i boucle vaut: " << i << endl;

   cout << "i du main vaut après: " << i << endl;

   return 0;
}
```

En sortie:

```
i du main vaut avant: 500
i boucle vaut: 0
i du main vaut après: 500
```

Il est impossible d'afficher la valeur de la variable `i` de la méthode **`main`** à partir de la boucle `for`.

```
#include <iostream>

using namespace std;

int i = 1000;

int main() {
   int i=500;
   cout << "i du main vaut avant: " << i << endl;
   cout << "i en dehors du main vaut avant: " << ::i << endl;
   for (int i=0; i< 1; i++)
         cout << "i boucle vaut: " << i << endl;
   cout << "i du main vaut après: " << i << endl;
   cout << "i en dehors du main vaut après: " << ::i << endl;

   return 0;
}
```

En sortie:

```
i du main vaut avant: 500
i en dehors du main vaut avant: 1000
i boucle vaut: 0
i du main vaut apres: 500
i en dehors du main vaut apres: 1000
```

- Le symbole « `::` » désigne l'opérateur unaire de résolution de portée.
- On l'appelle aussi l'opérateur de visibilité (scope qualifier operator)
- Dans cet exemple, « `::` » fait référence à l'espace de nom anonyme (qui ne porte pas de nom).

4. Type bool

- En C, il n'y a pas de type booléen. On utilise **`int`** ou bien un type prédéfini.

- Par contre en C++, le type **`bool`** a été défini.
- Une variable du type `bool` peut prendre la valeur **`0`** ou **`1`** ou :

```
0 → false
1 → true
```

```
#include <iostream>

using namespace std;

int main() {
     bool valide=false;
     cout << valide << endl; // affiche: 0
     valide = true;
     cout << valide << endl; // affiche: 1
     return 0;
}
```

5. const et define

5.1. #define

La commande permet:

- La définition de macros (sera étudiée un peu plus loin dans le cours, voir l'utilisation de la fonction **inline**).

- La substitution de texte

```
#define dimension 10
```

- C'est une directive au préprocesseur qui demande le remplacement lexical de la chaîne **dimension** par la valeur **10**.
- Aucun espace mémoire n'est alloué pour stocker **10**.

```
const int dimension=10;
```

- C'est une directive au compilateur : l'espace mémoire de taille **int** est réservé à la variable **dimension** qui doit être initialisée.

- Cette variable a été initialisée avec la valeur **10**.
- Elle ne pourra pas être modifiée.

Privilégier « **const** » à « **define** », c'est plus sécuritaire car le compilateur connaît l'existence de la variable et peut repérer les erreurs lors de son utilisation.
Par ailleurs, la variable devient disponible dans la table des symboles. De ce fait, le débogueur connaît aussi son existence.

5.2. Les constantes

- La valeur d'une constante ne peut pas être modifiée.

- L'adresse d'une constante ne peut pas être affectée à une variable (ce point sera développé dans le chapitre relatif à l'utilisation des pointeurs).

```
const int j=100;
j=50; // Erreur la variable j est constante, on ne peut pas modifier sa valeur
```