Chapitre 4

Les fonctions, les références , les constantes.

Les Fonctions en C++

1- Prototype et caractéristiques d'une fonction

✓ **Le passage d'arguments à une fonction se fait au moyen d'une liste d'arguments.**

✓ **La signature d'une fonction est le nombre et le type de chacun de ses arguments.**

✓ **Surcharge : Possibilité d'avoir plusieurs fonctions ayant le même nom mais des signatures différentes.**

✓**Possibilité d'avoir des valeurs par défaut pour les paramètres, qui peuvent alors être sous-entendus au moment de l'appel.**

Exemple

```
int mult (int a=2, int b=3)    { return a*b; }
```

```
mult (3,4) => 12
mult (3)   => mult(3,3) ,9
mult ( )   => mult(2,3) ,6
```

2- Pointeurs sur des fonctions

✓ **Il est parfaitement possible d'utiliser des pointeurs sur des fonctions en C++. Il s'agit de pointeurs particuliers qui désignent le point d'entrée d'une fonction dans le programme.**

✓ **On les déclare ainsi:**

```
int (*pf1)(double);
long (*pf2)(void);      //Les parenthèses sont obligatoire
void (*pf3)(int, float);
```

✓ pf1 **est un pointeur sur une fonction ayant un argument double, et de résultat entier.**

Exemple :

```
int unefonction(double d);
int (*pf1)(double) = &unefonction;  //Initialisation du pointeur de fonction
```

Ensuite, on peut appeler la fonction pointée comme suit:

```
int  i = (*pf1) (10.1);
```

✓ **Noter que pour affecter l'adresse d'une fonction à un pointeur, l'adéquation des arguments et du résultat doit être parfaite. Par exemple, les deux initialisations suivantes seront refusées par le compilateur :**

```
int fonct1(unsigned u);
void fonct2( const char *s);

int (*p1)(int) = &fonct1; // non, incorrect
void (*p2)(char *s) = &fonct2; // idem
```

✓ **On doit dans ce cas faire un changement de type :**

```
int (*p1)(int) = (int (*)(int)) &fonct1; // ok
void (*p2)(char *s) = (void (*)(char*)) &fonct2; // ok
```

✓**Les pointeurs de fonctions sont des pointeurs spéciaux, et aucune opération arithmétique n'est permise sur eux.**

✓ **Si par exemple on écrit :**

```
p1++;   // Error : Size of this expression is unknown or zero
```

Les Références en C++

1- Déclaration d'une référence :

- ✓ En plus des pointeurs, le C++ permet de créer des références.
- ✓ Les références sont des synonymes d'identificateurs.
- ✓ Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

- ✓ ***Note :*** - Les références n'existent qu'en C++.
  - Le C ne permet pas de créer des références.

Par exemple:

Si « *id* » est le nom d'une variable, il est possible de créer une référence « *ref* » de cette variable.
Les deux identificateurs *id* et *ref* représentent alors la même variable, et celle-ci peut être accédée et modifiée à l'aide de ces deux identificateurs indistinctement.

✓ **Toute référence doit se référer à un identificateur : il est donc impossible de déclarer une référence sans l'initialiser.**

✓ **De plus, la déclaration d'une référence *ne crée pas un nouvel objet comme c'est le cas pour la déclaration d'une variable par exemple.* En effet, les références se rapportent à des identificateurs déjà existants.**

✓ **La syntaxe de la déclaration d'une référence est la suivante :**

type &référence = identificateur;

✓ **Après cette déclaration, la référence peut être utilisée partout où l'identificateur peut l'être. Ce sont des synonymes.**

Exemple . *Déclaration de références*

```
int i=0;
int &ri=i;   // Référence sur la variable i.
ri=ri+i;        // Double la valeur de i (et de ri).
```

✓**Il est possible de faire des références sur des valeurs numériques.**

✓**Dans ce cas, les références doivent être déclarées comme étant constantes:**

```
const int &ri=3;  // Référence sur 3.
int &ri=4;    // Erreur ! La référence n'est pas constante.
```

2- Lien entre les pointeurs et les références

✓**Les références et les pointeurs sont étroitement liés. En effet, une variable et ses différentes références ont la même adresse, puisqu'elles permettent d'accéder à un même objet.**

✓**Utiliser une référence pour manipuler un objet revient donc exactement au même que de manipuler un pointeur constant contenant l'adresse de cet objet.**

✓**Les références permettent simplement d'obtenir le même résultat que les pointeurs, mais avec une plus grande facilité d'écriture.**

✓ **Cette similitude entre les pointeurs et les références se retrouve au niveau syntaxique. Par exemple, considérons le morceau de code suivant :**

```
int i=0;
int *pi=&i;
*pi=*pi+1;   // Manipulation de i via pi.
```

✓ **Maintenant, comparons avec le morceau de code équivalent suivant :**

```
int i=0;
int &ri=i;
ri=ri+1;      // Manipulation de i via ri.
```

✓**Nous constatons que la référence ri peut être identifiée avec l'expression *pi, qui représente bien la variable i.**

✓**Ainsi, la référence ri encapsule la manipulation de l'adresse de la variable i et s'utilise comme l'expression *pi.**

✓**La différence se trouve ici dans le fait que les références doivent être initialisées.**

✓**Les références sont donc beaucoup plus faciles à manipuler que les pointeurs, et permettent de faire du code beaucoup plus sûr.**

3- Passage de paramètres par variable ou par valeur

Il y a deux méthodes pour passer des variables en paramètre dans une fonction : le passage par valeur et le passage par variable.

3.1. Passage par valeur

✓**La valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.**

C'est-à-dire:

➢**Si l'expression passée en paramètre est une variable, son contenu est copié dans la variable locale.**

➢ **Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.**

Exemple:

```
void test(int j)     /* j est la copie de la valeur passée en
                        paramètre */
{
    j=3;             /* Modifie j, mais pas la variable fournie
                        par l'appelant. */
}

int main(void)
{
    int i=2;
    test(i);         /* Le contenu de i est copié dans j.
                        i n'est pas modifié. Il vaut toujours 2. */
    test(2);         /* La valeur 2 est copiée dans j. */
    return 0;
}
```

3.2. Passage par variable

✓**La deuxième technique consiste à passer non plus la valeur des variables comme paramètre, mais à passer les variables elles-mêmes.**

✓ **Il n'y a donc plus de copie, plus de variable locale.**

✓**Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.**

3.3. Avantages et inconvénients des deux méthodes

✓Les passages par variables sont plus rapides et plus économes en mémoire que les passages par valeur, puisque les étapes de la création de la variable locale et la copie de la valeur ne sont pas faites.

✓Il faut donc éviter les passages par valeur dans les cas d'appels récursifs de fonction ou de fonctions travaillant avec des grandes structures de données (matrices par exemple).

✓Les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètre.

✓ Si l'on veut se prévenir de la destruction accidentelle des paramètres passés par variable, il faut utiliser le mot clé const. Dans ce cas, le compilateur interdira alors toute modification de la variable dans la fonction appelée.

3.4. Comment passer les paramètres par variable en C ?

✓Il n'y a qu'une solution : passer l'adresse de la variable.

✓Cela constitue donc une application des pointeurs.

Exemple:

```c
void test(int *pj)  /* test attend l'adresse d'un entier... */
{
    *pj=2;          /* ... pour le modifier. */

}

int main(void)
{
    int i=3;
    test(&i);       /* On passe l'adresse de i en paramètre. */
    /* Ici, i vaut 2. */
    return 0;
}
```

3.5. Passage de paramètres par référence

Le passage de paramètres par variable présente beaucoup d' inconvénients

✓ **la syntaxe est lourde dans la fonction, à cause de l'emploi de l'opérateur * devant les paramètres ;**

✓ **la syntaxe est dangereuse lors de l'appel de la fonction, puisqu'il faut systématiquement penser à utiliser l'opérateur & devant les paramètres.**

✓**Un oubli devant une variable de type entier implique directement la valeur de l'entier sera utilisée à la place de son adresse dans la fonction appelée (plantage assuré, essayez avec scanf).**

Le C++ permet de résoudre tous ces problèmes à l'aide des références. Au lieu de passer les adresses des variables, il suffit de passer les variables elles-mêmes en utilisant des paramètres sous la forme de références. La syntaxe des paramètres devient alors :

type identificateurfct(type &identificateur)

Exemple:

```
void test(int &i)   // i est une référence du paramètre constant
{
    i = 2;    // Modifie le paramètre passé en référence.

}

int main(void)
{
    int i=3;
    test(i);
    // Après l'appel de test, i vaut 2.
    // L'opérateur & n'est pas nécessaire pour appeler
    // test.
    return 0;
}
```

✓ **Il est recommandé, pour des raisons de performances, de passer par référence tous les paramètres dont la copie peut prendre beaucoup de temps (en pratique, seuls les types de base du langage pourront être passés par valeur).**

✓ **Bien entendu, il faut utiliser des références constantes au maximum afin d'éviter les modifications accidentelles des variables de la fonction appelante dans la fonction appelée.**

✓ **En revanche, les paramètres de retour des fonctions ne devront pas être déclarés comme des références constantes, car on ne pourrait pas les écrire si c'était le cas.**

Exemple:

```
typedef struct
{
    ...
} structure;

void ma_fonction(const structure & s)
{
    ...
    return ;
}
```

✓Dans cet exemple, s est une référence sur une structure constante.

✓Le code se trouvant à l'intérieur de la fonction ne peut donc pas utiliser la référence s pour modifier la structure (on notera cependant que c'est la fonction elle-même qui s'interdit l'écriture dans la variable s).

✓**Un autre avantage des références constantes pour les passages par variables est que si le paramètre n'est pas une variable ou, s'il n'est pas du bon type, une variable locale du type du paramètre est créée et initialisée avec la valeur du paramètre transtypé.**

```
void test(const int &i)
{
    ...         // Utilisation de la variable i
                // dans la fonction test. La variable
                // i est créée si nécessaire.
    return ;
}

int main(void)
{
    test(3);    // Appel de test avec une constante.
    return 0;
}
```

Au cours de cet appel, une variable locale est créée (la variable i de la fonction test), et 3 lui est affectée.

Les Constantes en C++

- ✓ **Il est possible en C++ de définir des données constantes.**

  const double Pi = 3.141592;

- ✓ **Toute tentative d'écriture se soldera par un refus très net du compilateur :**

  Pi += 1; // refusé !!

- ✓ **Si l'on tente d'utiliser des pointeurs :**

  double *dp = Π

  on obtient une erreur.

- ✓ **Il faut en effet écrire**

  const double *dp = Π

  double const *dp = Π

✓ **Cette déclaration signifie que *dp est constant. En conséquence, toute occurrence de *dp dans le programme est remplacée par la constante.**

✓**En effet, le pointeur lui-même n'est pas constant, on peut l'incrémenter: dp++;**

✓ **Quant aux références, on peut parfaitement les utiliser : En fait, si on initialise la référence sur une constante, ceci provoque la création d'une variable provisoire.**

```
const double &d = Pi;   // valable
d++;                    // refuser,
```

✓On peut parfaitement définir des tableaux constants :

```
const int table[3] = { 1, 2, 3 };
```

et des pointeurs constants (ne pas confondre avec les pointeurs sur des constantes) :

```
double *const dc = &d;
```

✓Dans ce cas, l'opération dc++ par exemple est interdite, puisqu'il s'agit d'un pointeur constant.

✓Par contre, on peut écrire :

```
(*dc)++;                // équivaut à d++ ;
```

✓On ne peut donc pas initialiser dc avec &Pi.

✓Il existe aussi des pointeurs constants pointant sur des constantes :

```
const int *const dcc = table;
```

✓Dans ce cas, on ne peut modifier ni dcc ni *dcc.