Chapitre 4

Les fonctions, les références , les constantes.

Fonction en C ++:

Dans beaucoup de langages, on trouve deux sortes de « modules », à savoir:

-Les fonctions, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments qui correspondent à des informations qui lui sont transmises et elle fournit un unique résultat.

- Les procédures (terme Pascal) ou sous-programme (terme Fortran ou Basic) qui élargissent la notion de fonction.

Pour élaborer une fonction C/C++, il faut coder les instructions qu'elle doit exécuter, en respectant certaines règles syntaxiques. Ce code source est appelé définition de la fonction. Une définition de fonction spécifié:

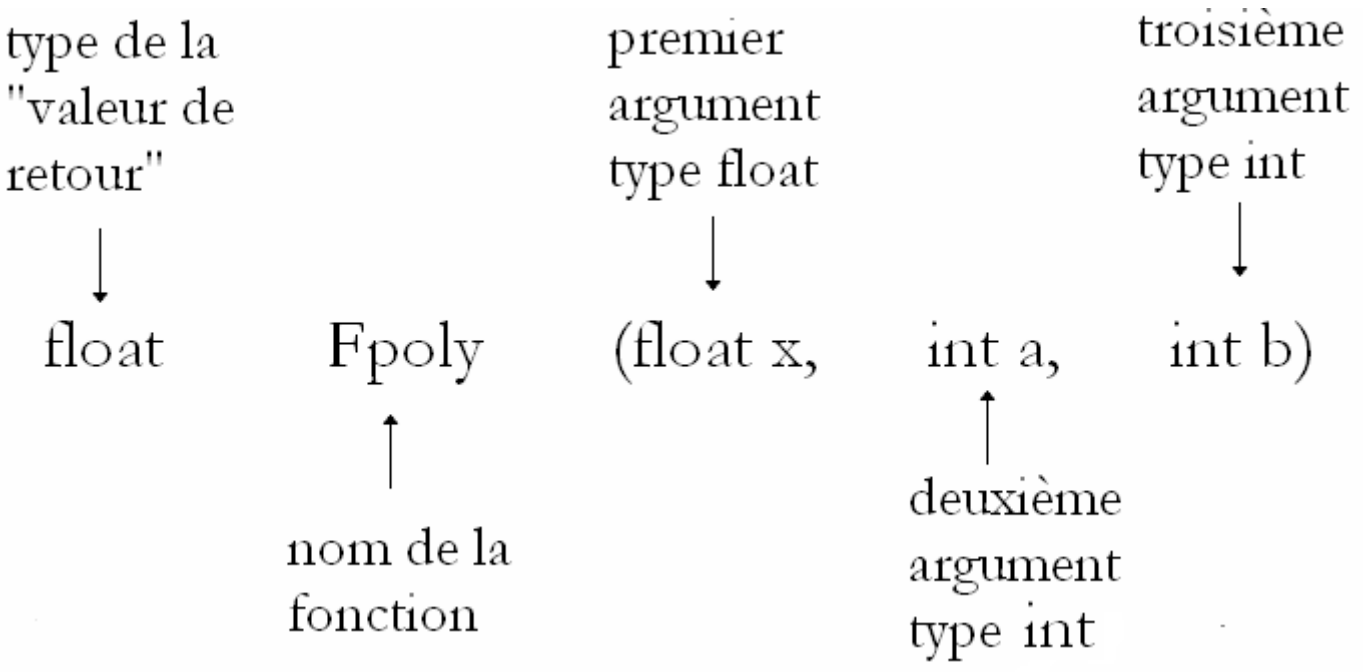
- **La classe de mémorisation de la fonction;**
- **Le type de la valeur renvoyée par la fonction;**
- **Le nom de la fonction;**
- **Les paramètres (arguments) qui sont passés à la fonction pour y être traités;**
- **Les variables locales et externes utilisés par la fonction;**
- **D'autres fonctions invoquées par la fonction;**
- **Les instructions que exécute la fonction.**

Exemple d'illustration

Je vous propose d'examiner tout d'abord un exemple de fonction correspondant à l'idée usuelle que l'on se fait d'une fonction, c'est-à-dire recevant des arguments et fournissant une valeur.

Dans cet exemple on va tester

$$f(x)=a\,x + b, \text{ avec } a=4 \text{ et } b=3.$$


type de la "valeur de retour"
premier argument type float
troisième argument type int
float
Fpoly
(float x,
int a,
int b)
nom de la fonction
deuxième argument type int

```
float Fpoly(float x, int a, int b)
{
  return(a*x+b);
}
```

float Fpoly (float ,int ,int);

Elle sert à prévenir le compilateur que Fpoly est une fonction donnée par 3 arguments et une valeur de retour de type float.

y=Fpoly (x ,m ,n);

Appel de la fonction Fpoly avec les arguments x, m et n.

Arguments muets et arguments effectifs

1- Arguments muets:

Les noms des arguments figurants dans l'en-tête de la fonction se nomment des arguments muets ou encore des arguments formels. Leur rôle est de permettre, au sein de la fonction, de décrire ce quelle doit faire.

2- Arguments effectifs:

Les arguments fournis (transmis) lors de l'utilisation (l'appel) de la fonction se nomment des arguments effectifs.

L’instruction return

Voici quelques règles générales concernant cette instruction:

- L’instruction return peut mentionner n’importe quelle expression.
- L’instruction return peut apparaître à plusieurs reprise dans une fonction.
- Si le type de l’expression figurant dans return est différent du type du résultat tel qu’il a été déclaré dans l’en-tête, le compilateur mettra automatiquement en place des instruction de conversion.

Exercice

1- Déclarez et appelez une fonction qui calcul la valeur absolue d'entier.

2- On dispose de 4 entiers. Calculez l' entier le plus grand. Pour cela :
Déclarer et appeler plusieurs fois une fonction qui compare deux entiers et qui renvoie le plus grand.

Fonctions sans retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise à la fois dans l'en-tête et dans sa déclaration à l'aide du mot clé void. Par exemple:

Déclaration : void FoncSansRt(int);

En-tête: void FoncSansRt(int a)

Naturellement, la définition d'une telle fonction ne doit pas contenir aucune instruction return.

Quand une fonction ne reçoit aucun arguments, on place le mot clé void à la place de la liste des arguments
Par exemple:

Déclaration : float FoncSansAr(void);

En-tête: float FoncSansAr(void)

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour.
Par exemple:

Déclaration : void FoncSansRtSansAr(void);

En-tête: void FoncSansRtSansAr (void)

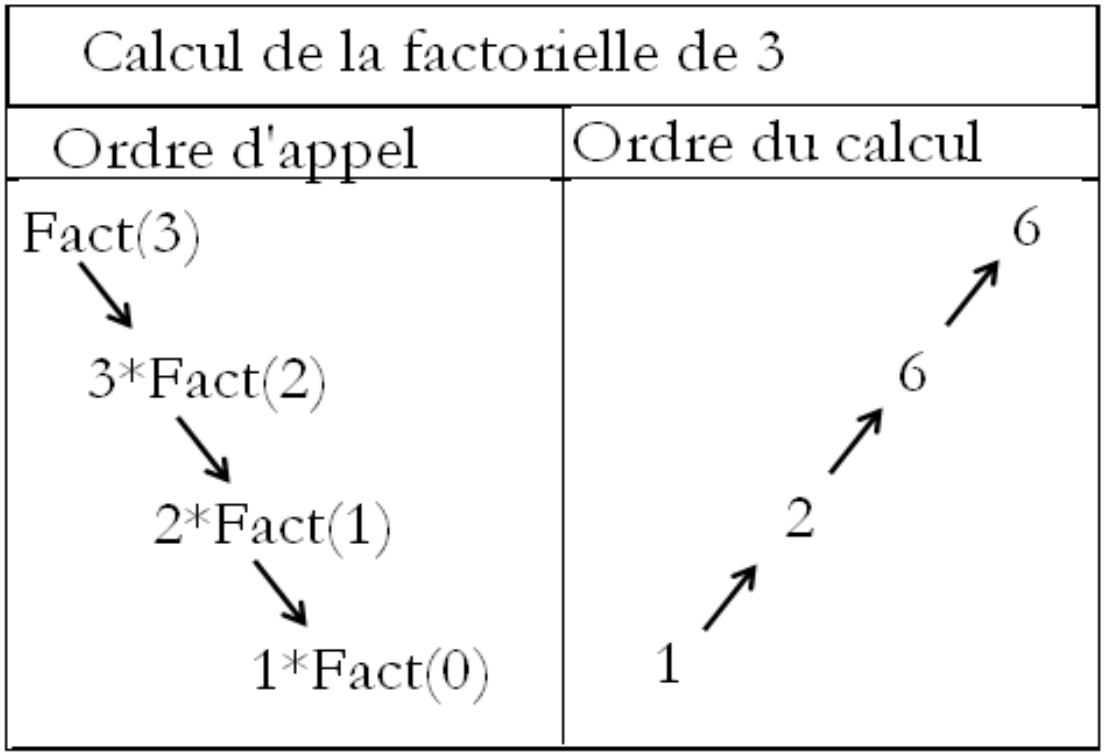
Fonctions récursives

Le langage C autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects:

- Récursivité directe.

- Récursivité croisé.

Voici un exemple classique d'une fonction calculant une factorielle de manière récursive.


Calcul de la factorielle de 3
Ordre d'appel
Ordre du calcul
Fact(3)
3*Fact(2)
2*Fact(1)
1*Fact(0)
6
6
2
1


C:\Documents and Settings\lamnii\Bureau\TPC\Fact.exe
donnez n
3
3! = 6

Faire un programme qui calcule C^i_n

Procédé pratique

Pour trouver une solution récursive d'un problème, on cherche à le décomposer en plusieurs sous problèmes de même type, mais de taille inférieurs. On procède de la manière suivante:

1. **Rechercher un cas trivial et sa solution (évaluation sans récursivité).**
2. **Décomposer le cas général en cas plus simples, eux aussi décomposables pour aboutir au cas trivial.**

Exemple de récursivité avec deux appels

Calcul du n^{ème} terme de la suite de Fibonacci

$$u_0 = 0$$
$$u_1 = 1$$
$$u_n = u_{n-1} + u_{n-2},\ n \geq 2$$

- Prototype et caractéristiques d'une fonction en C++

✓ **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
```

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=2;
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:

```
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 identificateur(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.