CHAPITRE 4

LES FONCTIONS EN C

I. INTRODUCTION

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée main. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est *récursive*).

II. DEFINITION D'UNE FONCTION

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme

```
type nom-fonction (type-1 arg-1,...,type-n arg-n)
{
            [déclarations de variables locales];
            liste d'instructions;
}
```

La première ligne de cette définition est **l'en-tête** de la fonction. Dans cet en-tête, *type* désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne (résultat). Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef void. Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef void.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, return, dont la syntaxe est **return**(*expression*);

La valeur de *expression* est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type void), sa définition s'achève par l'instruction **return**;

Plusieurs instructions return peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier return rencontré lors de l'exécution. Voici quelques exemples de définitions de fonctions :

```
int produit (int a, int b)
{
  return(a*b);
}
int puissance (int a, int n)
{
  if (n == 0)
```

```
return(1);
return(a * puissance(a, n-1));
}
void imprime_tab (int *tab, int nb_elements)
{
  int i;
  for (i = 0; i < nb_elements; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}</pre>
```

III. APPEL D'UNE FONCTION

L'appel d'une fonction se fait par l'expression *nom-fonction(para-1,para-2,...,para-n)*

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur *virgule*. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (++ ou --) dans les expressions définissant les paramètres effectifs.

IV. DECLARATION D'UNE FONCTION

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main. Toutefois, il est indispensable que le compilateur ``connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction main), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype*, qui donne le type de la fonction et celui de ses paramètres, sous la forme : type nomfonction(type-1,...,type-n);

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction main. Par exemple, on écrira

```
int puissance (int, int );
int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction main et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le prototype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux

types indiqués dans le prototype. Ainsi les fichiers d'extension **.h** de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier math.h le prototype de la fonction **pow** (élévation à la puissance) :

double pow(double , double);

La directive au préprocesseur **#include <math.h>** permet au préprocesseur d'inclure la déclaration de la fonction pow dans le fichier source. Ainsi, si cette fonction est appelée avec des paramètres de type int, ces paramètres seront convertis en double lors de la compilation. Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction pow avec des paramètres de type int peut produire un résultat faux !

V. VARIABLES GLOBALES

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, n est une variable globale :

```
int n;
void fonction();

void fonction()
{
    n++;
    printf("appel numero %d\n",n);
    return;
}

main()
{
    int i;
    for (i = 0; i < 5; i++)
        fonction();
}</pre>
```

La variable n est initialisé à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche.

VI. VARIABLES LOCALES

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et

donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant :

```
int n = 10;
void fonction();
void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}
main()
{
```

```
int i;
for (i = 0; i < 5; i++)
fonction();
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef static : **static** *type nom-de-variable*;

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, n est une variable locale à la fonction secondaire fonction, mais de classe statique.

```
int n = 10;
void fonction();

void fonction()
{
    static int n;
    n++;
    printf("appel numero %d\n",n);
    return;
}

main()
{
    int i;
    for (i = 0; i < 5; i++)
        fonction();
}</pre>
```

On voit que la variable locale n'est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.