LICENCE 2

MATHEMATIQUES APPLIQUEES ET INFORMATIQUE

LANGAGE C

LES FONCTIONS

Enseignant concepteur : M. Khalifa SYLLA

UVS
UNIVERSITÉ VIRTUELLE DU SÉNÉGAL

FONCTIONS

Il arrive fréquemment qu'une même suite d'instruction doive être exécutée en plusieurs points du programme. Une fonction permet d'associer un nom à une suite d'instruction et d'utiliser ce nom comme abréviation chaque fois que la suite apparaît dans le programme.

Pour alléger l'écriture du programme, il est conseillé de rédiger des fonctions qui sont appelées dans le corps du programme.

La modularité des programmes ainsi obtenue assure :

- une meilleure qualité de programmation
- un code plus court et par suite une diminution du risque d'erreurs de syntaxe,
- une mise au point aisée et rapide des programmes,
- une facilité de maintenance accrue,
- une facilité d'intégration de modules à d'autres programmes,
- une centralisation de la résolution d'un problème autour d'un programme principal.

Il est immédiat que la qualité et l'élégance de la solution finale dépendront fondamentalement de la qualité de la décomposition choisie.

Les modules crées sont appelés les sous programmes ; en C, on parle de fonctions.

L'intérêt des sous-programmes est :

- *de permettre d'écrire un programme en suivant l'analyse descendant ;*
- *d'éviter d'écrire plusieurs fois une séquence d'instructions destinée à être exécutée plusieurs fois ; il suffit alors d'en faire une fonction et de l'appeler autant de fois que nécessaire ;*
- *de réutiliser facilement des traitements déjà écrits sous forme de fonctions;*
- *de partager le travail entre programmeurs lors de la résolution d'un problème informatique complexe.*

I- Définition d'une fonction

Avant d'être utilisée, une fonction doit être définie car pour l'appeler dans le corps du programme il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions qu'elle contient.

La définition d'une fonction s'appelle "*déclaration*".

La déclaration d'une fonction se fait selon la syntaxe suivante:

```
type NomFonction(type1 argument1, type2 argument2, ...)
{
liste d'instructions
}
```

Remarques:

- La première ligne de cette définition est l'*en-tête* de la fonction. Dans cet en-tête, type_de_donnee représente le type de valeur que la fonction est sensée retourner (char, int, float,...)
- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé *void*
- Si aucun type de donnée n'est précisé (cela est très vilain!), le type *int* est pris par défaut
- le nom de la fonction suit les mêmes règles que les noms de variables:
  - le nom doit commencer par une lettre
  - un nom de fonction peut comporter des lettres, des chiffres et les caractères _ et & (les espaces ne sont pas autorisés!)
  - le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)
- 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 ***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;
}
```

II- Appel d'une fonction

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivie d'une parenthèse ouverte (éventuellement des arguments)puis d'une parenthèse fermée. 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.

Prototype d'une fonction

Le prototype d'une fonction est une description d'une fonction qui est définie plus loin dans le programme. On place donc le prototype en début de programme (avant la fonction principale *main()*).
Cette description permet au compilateur de "vérifier" la validité de la fonction à chaque fois qu'il la rencontre dans le programme, en lui indiquant:

- Le type de valeur renvoyée par la fonction
- Le nom de la fonction
- Les types d'arguments

Contrairement à la **définition** de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas le nom des paramètres (seulement leur type).

Syntaxe

Type NomFonction(type_argument1, type_argument2, ...);

Exemple

```
void Affiche_car(char, int);
int Somme(int, int);
```

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 protype.

III- Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même *durée de vie*. On distingue deux catégories de variables.

1. Les variables permanentes (ou statiques)

Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef ***`static`***.

2. Les variables temporaires

Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, `auto`, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type `register`. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur, il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

La durée de vie des variables est liée à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

3. 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();
}
```

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

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

4. 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 :

```
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();
}
```

Le programme affiche :

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

```
appel numero 1
```

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();
}
```

Ce programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

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.

IV- Arguments d'une fonction

Il est possible de passer des arguments à une fonction, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la fonction puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments. Le passage d'arguments à une fonction se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la fonction.

Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre au risque, sinon, de générer une erreur lors de la compilation...

Un argument peut être une constante, une variable, une expression ou une autre fonction.

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*.

Par exemple :

```
void echange (int, int );

void echange (int a, int b)
{
 int t;
 printf("debut fonction :\n a = %d \t b = %d\n",a,b);
 t = a;
 a = b;
 b = t;
 printf("fin fonction :\n a = %d \t b = %d\n",a,b);
 return;
}



main()
{
 int a = 2, b = 5;
 printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
 echange(a,b);
 printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Ce programme imprime :

```
debut programme principal :
 a = 2   b = 5
debut fonction :
 a = 2   b = 5
fin fonction :
 a = 5   b = 2
fin programme principal :
 a = 2   b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur (chapitre pointeurs).

Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);

void echange (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction.

Par exemple :

```
void init (int *, int );
void init (int *tab, int n)
{
  int i;
  for (i = 0; i < n; i++)
    tab[i] = i;
  return;
}
main()
{
  int i, n = 5;
  int *tab;
  tab = (int*)malloc(n * sizeof(int));
  init(tab,n);
}
```

Le programme initialise les éléments du tableau `tab`.