# Chapitre 10

## Ecriture, Appel, et Utilisation de fonctions

A présent, nous allons étudier les fonctions. Vous savez déjà comment en utiliser, puisque les ROM_CALLs ne sont rien de plus que des fonctions, mais nous n'avons pas encore réellement appris à en écrire, si ce n'est la fonction _main, qui est un cas particulier de fonctions.
Au cours de ce chapitre, nous verrons qu'elle est l'utilité d'écrire des fonctions, puis nous apprendrons à en écrire, et nous finirons par montrer comment on les appelle.

## I:\ Mais pourquoi écrire, et utiliser des fonctions ?

Lorsque l'on programme, en C comme dans bien d'autres langages, il est fréquent d'avoir à utiliser plusieurs fois une portion de code dans un programme, ou, même, d'utiliser la même portion de code dans plusieurs programmes. Cela dit, nous ne voulons pas avoir à réécrire ce code à chaque fois ; ce serait d'ailleurs une ridicule perte de temps, et d'espace mémoire !
C'est pour cela que les notions de "fonction", ou de "procédure", ont été créées.
Qu'est-ce qu'une fonction ? Pour faire bref, on peut dire que c'est une portion de code, qui peut travailler sur des données que l'on lui fourni, qui peut renvoyer un résultat, et qui est souvent destinée à être utilisée plusieurs fois, par son créateur ou par quelqu'un d'autre, sans avoir à être ré-écrite à chaque fois.

Certaines fonctions, couramment appelées ROM_CALLs sont inclues à la ROM ; nous avons déjà appris comment les utiliser. D'autres sont inclues dans les bibliothèques partagées de TIGCC. Toutes ces fonctions, vous pouvez les appeler sans avoir à les ré-écrire à chaque fois ; admettez que c'est bien pratique : imaginiez qu'il vous faille recopier plusieurs dizaines, voire centaines, de lignes de code C ou Assembleur, selon les cas, à chaque fois que vous voulez afficher un message, ou attendre qu'une touche soit pressée au clavier... rien qu'en pensant à cela, je suis certain que vous comprenez aisément l'utilité, et même la nécessité, des fonctions !
Vous pouvez aussi, et c'est le sujet de ce chapitre, écrire vos propres fonctions, que vous pourrez appeler tout à fait de la même manière que celles que nous avons jusqu'à présent utilisé. D'ailleurs, sans vraiment le savoir, vous en avez déjà écrit : `_main` est une fonction, obligatoire certes, mais une fonction tout de même !

Si vous lisez des documents traitants d'algorithmie, vous pourrez peut-être constater qu'ils font souvent, tout comme certains langages de programmation, une distinction entre ce qui est appelé "procédures", qui sont des portions de code effectuant un traitement à partir de une ou plusieurs données, et ne retournant aucun résultat et ce qui est appelé "fonctions", qui sont des portions de code effectuant elles aussi un traitement à partir de une ou plusieurs données, mais retournant un résultat. Le langage C ne fait pas de différence entre les notions de "fonctions" et de "procédures" : le terme de fonction est généralisé, et une fonction peut retourner quelque chose... ou rien. Lorsque l'on programme dans ce langage, le terme de procédure n'est donc que rarement employé.

Avant que l'on passe à l'écriture de fonctions, notez que ce qui est important pour une fonction, c'est de savoir ce qu'elle fait, à partir de quoi, et ce qu'elle renvoie : l'utilisateur de la fonction n'a pas à savoir comment elle effectue son traitement, ni par quel algorithme !
Si une fonction est bien pensée, il doit être possible de changer totalement son mode de fonctionnement sans avoir à changer ni ce qu'elle retourne, ni ce qu'elle attend comme données ;

autrement dit, il doit rester possible d'utiliser la fonction exactement de la même manière, sans même avoir à savoir que son mode de fonctionnement a été modifié. Ceci est d'autant plus vrai si vous avez l'intention de diffuser une fonction !

# II:\ Écrire une fonction

Maintenant que nous avons vu l'utilité des fonctions, voyons comment en écrire.

## A: Écriture générale de définition d'une fonction

En C, une fonction a toujours un type de retour, qui correspond au type du résultat qu'elle peut renvoyer et qui peut être n'importe lequel des types que nous avons précédemment étudié ; ce type de retour peut être `void` si on souhaite que la fonction ne renvoie rien. Elle a aussi un nom, qui respecte les conventions de nommage des variables (des lettres, chiffres, ou '_', en commençant par une lettre ou un underscore). Et, enfin, elle a une liste de zéro, un, ou plusieurs, paramètres.
Voici ce à quoi ressemble la définition d'une fonction :

```
type_de_retour nom_de_la_fonction(type_param_1 nom_param_1, type_param_2 nom_param_2)
{
    Contenu de la fonction
}
```

Comme je l'ai dit juste au-dessus, on peut n'avoir aucun paramètre, ou un, ou deux, ou autant qu'on veut. Ici, j'en ai mis deux, mais j'aurai pu en mettre un nombre différent.
Tant que nous en sommes à parler des paramètres, ils peuvent prendre pour type n'importe lequel de ceux que nous avons jusqu'à présent étudié et de ceux que nous étudieront plus tard.
Il est des gens qui utilisent le terme de "paramètre" lorsque l'on déclare la fonction et de "argument" lorsqu'on l'utilise ; d'autres personnes font exactement le contraire... pour simplifier, et n'ayant pas trouvé de norme à ce sujet, j'utilise les deux indifféremment.

## B: Cas d'une fonction retournant une valeur

Une fonction déclarée comme ayant un type de retour différent de `void` doit retourner quelque chose. Pour cela, nous utiliserons l'instruction `return`, dont nous avons brièvement parlé au chapitre précédent.
Cette instruction, utilisée seule, permet de quitter une fonction (ou le programme, si la fonction est `_main`) ; si on la fait suivre d'une donnée, elle permet de quitter la fonction, en faisant en sorte que celle-ci renvoie la valeur précisée. Notez que le type de la donnée utilisée avec l'instruction `return` doit être le même que le type de retour de la fonction !

Lorsque l'on veut simplement quitter la fonction, on utilise cette écriture :

```
return;
```

Si l'on veut quitter une fonction renvoyant un entier, et que l'on souhaite retourner à l'appelant la valeur 150, on utilisera cette écriture :

```
return 150;
```

Notez que, pour une fonction dont le type de retour est `void`, qui, donc, ne renvoie rien, l'instruction `return` est optionnelle : une fois arrivé à la fin de la fonction, on retournera à l'appelant, même s'il n'y a pas d'instruction `return`.
Par contre, pour une fonction dont le type de retour est différent de `void`, il est obligatoire d'utiliser l'instruction `return`, en lui précisant quoi retourner. Dans le cas contraire, le compilateur vous signifiera qu'il n'est pas d'accord avec ce que vous avez écrit.

## C: Quelques exemples de fonctions

Tout d'abord, commençons par un exemple de fonction ne prennant pas de paramètre (autrement dit, elle prend `void` en paramètre, c'est-à-dire néant), et qui ne retourne rien non plus. Cette fonction ne fait rien de plus qu'afficher un message.

```
void fonction1(void)
{ // Cette fonction ne prend pas de paramètre,
  // et ne renvoie rien
    printf("Ceci est la fonction 1");
}
```

Passons à une fonction qui prend deux entier, un `short` et un `long`, en paramètres, et qui affiche leurs valeurs... en appelant la fonction `printf`, fournie dans les bibliothèques partagées de TIGCC.
Comme nous pouvons le remarquer, nous pouvons, au sein de la fonction, utiliser les paramètres tout à fait comme nous utilisons des variables ; en fait, les paramètres ne sont rien de plus que des variables, auxquelles on affecte une valeur au moment de l'appel de la fonction.

```
void fonction2(short param1, long param2)
{ // Cette fonction prend deux paramètres,
  // et ne renvoie rien
    printf("Param1 = %d,\nParam2 = %ld", param1, param2);
}
```

Un peu plus utile, une fonction qui prend en paramètre un entier, signé, sur 32 bits, qui calcule son carré, le stocke dans une variable, et retourne la valeur de cette variable :

```
unsigned long fonction_carre1(long a)
{ // Cette fonction calcule le carré du nombre
  // qui lui est passé en paramètre
    unsigned long result = a*a;
    return result;
}
```

A présent, exactement la même chose, sauf que, cette fois, on n'utilise plus de variable pour stocker le résultat : on retourne directement le résultat du calcul :

```
unsigned long fonction_carre2(long a)
{ // Cette fonction calcule le carré du nombre
  // qui lui est passé en paramètre
  // (Ecriture plus courte)
    return a*a;
}
```

Il est possible d'écrire des fonctions qui, tout comme `printf`, admettent un nombre variable d'arguments. Cela dit, il est rare d'avoir à écrire de telles fonctions, et cela est assez complexe. Nous ne traiterons donc pas ce sujet ici.

Naturellement, les fonctions utilisées ici en exemples sont volontairement très courtes, et ne contiennent que le code nécessaire à leur bon fonctionnement. En réalité, il serait ridicule d'utiliser des fonctions pour réaliser des tâches aussi courtes et simples, ne serait-ce du fait que l'on passe un petit peu de temps à appeler la fonction ! Remplacer une seule instruction par une fonction n'est généralement guère utile !
Cela dit, des exemples plus long n'auraient pas mieux montré les méthodes d'écriture de fonctions, et auraient risqué d'attirer votre attention sur des points qui n'entrent pas dans le sujet de ce chapitre, ce qui explique le choix que j'ai fait de ne montrer que des exemples brefs.

Pour finir, notez que même si appeler une fonction prend un certain temps, ce temps est vraiment minime, en particulier dans un langage fonctionnel tel le C, surtout à partir du moment où la fonction fait plus de quelques lignes. De plus, plus on appelle la fonction un grand nombre de fois, plus le gain en espace mémoire est réel et sensible.
Autrement dit, n'hésitez pas à utiliser des fonctions... sans toutefois en abuser dans des cas tels ceux que j'ai ici pris en exemple.

# D: Notion de prototype d'une fonction

Pour en finir avec l'écriture des fonctions, et faire le lien avec leur utilisation, nous allons étudier la notion de prototype d'une fonction.
Un prototype de fonction permet au compilateur de savoir ce que la fonction attend en paramètre, et le type de ce qu'elle retournera, même si le code correspondant à la fonction est écrit ailleurs, ou plus loin dans le fichier source.

## *1: Pourquoi ?*

Lorsque nous voulons appeler une fonction, il faut que le compilateur sache ce qu'elle retourne, et ce qu'elle prend en paramètre.
Puisque le compilateur lit les fichiers source du haut vers le bas, une solution, à laquelle on pense souvent lorsque l'on débute, est de déclarer toutes les fonctions avant leur utilisation... et, donc, de placer la fonction `_main` tout en bas du fichier source, en supposant que les fonctions ne s'appellent pas les unes les autres. Cela dit, cette méthode est loin d'être optimale ; en effet, elle ne fonctionnera pas si on a plusieurs fichiers sources, ou si les fonctions s'appellent les unes les autres. De plus, placer la fonction `_main` tout en fin de fichier ne facilite pas la lecture, puisque, nous aussi, nous avons tendance à lire de haut en bas, du début vers la fin.

C'est pour cela que le langage C introduit la notion de prototype de fonction.

## *2: Comment ?*

Le prototype d'une fonction reprend exactement l'en-tête de la fonction, mais pas son corps, qui est remplacé par un point-virgule.
Un prototype a donc une écriture de la forme suivante :

```
type_de_retour nom_de_la_fonction(type_param_1 nom_param_1, type_param_2 nom_param_2);
```

Dans l'écriture d'un prototype de fonction, les noms de paramètres sont optionnels. Si vous donnez à vos paramètres des noms évocateurs (ce que je ne peux que vous inciter à faire !), il peut être bon de les conserver dans les prototypes, afin que vous sachiez à quoi correspond chaque paramètre... A vous de voir, encore une fois.

En règle générale, on place les prototypes de fonctions en début de fichier source, et les fonctions en fin... Ou même, on place les fonctions dans d'autres fichiers sources, lorsque l'on travaille sur un projet suffisamment important.

## *3: Exemples*

Pour en finir avec les prototypes de fonctions, voici les prototypes des fonctions que nous avons précédemment utilisées en exemples :

```
void fonction1(void);
void fonction2(short param1, long param2);
unsigned long fonction_carre1(long a);
unsigned long fonction_carre2(long a);
```

# III:\ Appel d'une fonction

Appeler une fonction est fort simple ; nous l'avons d'ailleurs déjà fait maintes fois pour des ROM_CALLs qui, même si elles n'ont pas été écrites par vous, ne sont rien d'autre que des fonctions. Cela explique pourquoi nous serons aussi bref dans cette partie.
Il vous faut écrire le nom de la fonction, suivi de, entre parenthèses, les différents paramètres, s'il y en a. Naturellement, on fait suivre d'un point-virgule si on est en fin d'instruction.

Juste pour la forme, puisque nous savons déjà comment faire, voici quelques exemples ; tout d'abord, des fonctions ne retournant rien :

```
fonction1();

short a = 12;
fonction2(a, 345);  // Il faut, naturellement, que le paramètre passé
                    // soit du type attendu par la fonction.
```

Et maintenant, utilisons des fonctions retournant quelque chose ; Si l'on souhaite récupérer le résultat renvoyé par une fonction, on peut tout à fait déclarer une variable du type de la donnée renvoyée par la fonction, et lui affecter le résultat renvoyé.
Ou alors, on peut utiliser directement le résultat renvoyé, que ce soit dans un calcul, dans un autre appel de fonction, ...
En somme, une fonction renvoyant un résultat de type X peut être utilisée partout où l'on peut utiliser directement une valeur, ou une variable, de type X.

```
unsigned long a;
a = fonction_carre1(2);

unsigned long b = fonction_carre1(fonction_carre1(4));
```

Utiliser une fonction n'est pas plus difficile que cela... Et cela nous a évité d'avoir à écrire deux fois le code contenu par la fonction.
(Même si, avec une fonction aussi courte que celle-ci, il aurait mieux valu se passer de fonction, et utiliser directement le code qu'elle contient... du moins, si l'on n'était pas dans un cas d'école)

# IV:\ Quelques mots au sujet de la récursivité

Le langage C permet aux fonctions d'être récursives ; cela signifie qu'il est possible, en C, pour une fonction, de s'appeler elle-même.
Cela dit, il faut penser, à un moment ou à un autre, à inclure une condition permettant de stopper la récursion, afin de ne pas entrer dans une récursion infinie.

La récursion est parfois utilisée là où employer des boucles serait extrêmement compliqué.
En général, la récursion consiste à partir d'un état d'origine, à effectuer un traitement sur cet état, et de recommencer sur l'état obtenu.
En somme, la récursion sert souvent à illustrer la façon dont l'homme raisonne.

Si vous avez déjà suivi des cours d'algorithmie concernant les tris de données, vous avez fort probablement utilisé la récursion lorsque vous avez étudié le QuickSort (algorithme permettant de trier rapidement un tableau, dans la plupart des cas, en se basant sur le principe du "Diviser pour Régner"). Si vous avez déjà suivi des cours d'algorithmie avancée concernant les arbres, vous aurez utilisé la récursion pour parcourir un arbre (notez qu'il est probablement possible de parcourir un arbre avec des boucles... Mais je n'ose même pas imaginer à quel point cela doit être complexe par rapport à l'utilisation de la récursivité !).

Pour la forme, voici un bref exemple utilisant une fonction permettant de calculer la factorielle d'un nombre, de manière récursive

```
#include <tigcclib.h>

long factorielle(long a);

void _main(void)
{
    clrscr();
    printf("%ld", factorielle(5));
    ngetchx();
}

long factorielle(long a)
{
    if(a<=1) return 1;
    return a*factorielle(a-1);
}
```

Exemple Complet

Pour bien comprendre cet exemple, il faut se rappeler que la factorielle d'un nombre se défini comme étant le produit de ce nombre avec tous ceux qui lui sont inférieurs. Ainsi, 5! = 5*4*3*2*1.
Ceci pourrait se programmer avec une boucle de type `while`.
Cela dit, on peut aussi procéder ainsi :

5! = 5*4!
4! = 4*3!
3! = 3*2!
2! = 2*1

Avec ce raisonnement, on suit une logique récursive : pour calculer la factorielle d'un nombre, on calcule la factorielle de celui qui lui est inférieur, et on recommence... Tout en sachant que l'on doit s'arrêter à 1. C'est exactement ce fonctionnement qui est utilisé par notre second exemple :
On commence par appeler la fonction factorielle pour calculer la factorielle de 5, et celle-ci s'auto-appelle jusqu'à ce qu'on arrive à 1.

Étudiez bien cet exemple, qui est une application idéale de la notion de récursivité.

# V:\ Passage de paramètres par valeurs

En langage C, les paramètres de fonctions sont passés à celle-ci par valeur (on dit aussi parfois qu'ils sont passés par copie), ce qui signifie que la fonction appelée ne travaille pas sur les données mêmes que la fonction appelante lui a passé en paramètre, mais sur une copie de ces données, qui est propre à la fonction.
Cela implique qu'une fonction ne peut pas modifier des variables déclarées dans la fonction appelante.

Pour illustrer mes propos, prenons une fonction simple, qui prend en paramètre deux entiers, et essaye d'échanger leurs valeurs. Voici le coude-source correspondant :

```
#include <tigcclib.h>

void echange(short a, short b);

void _main(void)
{
    short a = 10;
    short b = 20;

    clrscr();

    printf("AVANT: a=%d ; b=%d\n", a, b);

    echange(a, b);

    printf("APRES: a=%d ; b=%d\n", a, b);

    ngetchx();
}


void echange(short a, short b)
{
    short temp;

    printf("DEBUT: a=%d ; b=%d\n", a, b);

    temp = a;
    a = b;
    b = temp;

    printf("FIN  : a=%d ; b=%d\n", a, b);
} // Fin echange
```

Exemple Complet

Si nous exécutons ce programme, nous aurons comme affichage :

```
AVANT: a=10 ; b=20
DEBUT: a=10 ; b=20
FIN  : a=20 ; b=10
APRES: a=10 ; b=20
```

Cela nous montre que, dans la fonction echange, les valeurs des deux variables a et b ont bien été échangées, mais que cela n'a eu absolument aucune influence sur les deux variables a et b de la fonction _main, qui avait appelé la fonction echange.
En effet, comme nous l'avons dit, les variables a et b de la fonction _main ont été passées par copie à la fonction echange, ce qui signifie que celle-ci n'a pas travaillé sur a et b de _main, mais sur ses propres variables a et b... que nous aurions d'ailleurs tout à fait pu appeler autrement sans que cela n'ait la moindre influence sur le déroulement du programme.

Au chapitre suivant, nous allons étudier ce que sont les pointeurs, et nous verrons qu'ils nous permettent de régler ce problème, en nous permettant de passer des paramètres à une fonction autrement que par valeur.