Introduction au langage C

Copyleft © GUINKO Tonguim Ferdinand - IBAM - Université de Ouagadougou

13 janvier 2010

Table des matières

Chapitre 1

Les fonctions

1.1 Introduction

Exemple 1 :

```
#include <stdio.h>

void main(void)
{  /* le corps de la fonction principale */
    int n=4;
    long temp=1;

    while(n>1)
       temp*=n--; /* On calcul fac(4) */
    printf("La factorielle de 4 est %d\n", temp);

    int n=5;
    temp=1;
    while(n>1)
        temp*=n--; /* On calcul fac(5) */
    printf("La factorielle de 5 est %d\n", temp);

    int n=6;
    temp=1;
    while(n>1)
        temp*=n--; /* On calcul fac(6) */
    printf("La factorielle de 5 est %d\n", temp);

    int n=7;
    temp=1;
    while(n>1)
        temp*=n--; /* On calcul fac(7) */
```

```
    printf("La factorielle de 5 est %d\n", temp);

    int n=8;
    temp=1;
    while(n>1)
        temp*=n--; /* On calcul fac(8) */
    printf("La factorielle de 5 est %d\n", temp);
}
```

Que constatez vous ? Le même code est utilisé 2 fois, ce qui pose un problème. En effet, on peut trouver plusieurs inconvénients au fait de dupliquer une portion de code au sein d'un même programme (plus généralement, au sein de différents programmes). Que se passe t'il si vous avez besoin de cette section de code dans bien plus que deux endroits d'un programme (par exemple 50) ? Vous pensez peut-être que copier et coller fera l'affaire : mais dans ce cas, que faites vous si une erreur s'est glissée dans la section de code dupliquée ? Alors une question se pose : que faire ?

L'idée consiste à définir cette section de code (jusqu'alors dupliquée) dans une sorte de containeur que l'on pourrait alors accéder de divers endroits. Ce mécanisme d'accession serait alors bien plus concis, diminuant ainsi considérablement la taille de votre programme. Et cela n'est pas négligeable, car développer un programme (en langage C ou en tout autre langage) de 1000 lignes est sans aucun doute plus aisé que d'en maintenir un de 10000 lignes. Ce mécanisme qui opère ainsi une factorisation de code se nomme une fonction. La notion de fonction en langage C est en un sens extrêmement proche de la notion de fonctions mathématiques.

Exemple 2 :

```
#include <stdio.h>

long fac(int n)
{ /* La fonction qui calcule la factorielle */
     long temp=1;
     while(n>1)
         temp*=n--;
     return temp;
}

void main(void)
{  /* le corps de la fonction principale */
     printf("La factorielle de 4 est %d\n", fac(4));
     printf("La factorielle de 5 est %d\n", fac(5));
     printf("La factorielle de 6 est %d\n", fac(5));
     printf("La factorielle de 7 est %d\n", fac(5));
     printf("La factorielle de 8 est %d\n", fac(5));
}
```

1.2 La modularité et ses avantages

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts. A l'aide de ces structures nous pouvons modulariser nos programmes pour obtenir des solutions plus élégantes et plus efficientes.

1.2.1 Modules

Dans ce contexte, un module désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

Dans ce manuel, les modules (en langage C : les fonctions) sont mis en évidence par une bordure dans la marge gauche.

1.2.2 Avantages

Voici quelques avantages d'un programme modulaire :

1. meilleure lisibilité ;
2. diminution du risque d'erreurs ;
3. possibilité de tests sélectifs ;
4. dissimulation des méthodes ;
5. *Réutilisation de modules déjà existants* : il est facile d'utiliser des modules qu'on a écrits soi-même ou qui ont été développés par d'autres personnes ;
6. *simplicité de l'entretien* : un module peut être changé ou remplacé sans devoir toucher aux autres modules du programme ;
7. *favorisation du travail en équipe* : un programme peut être développé en équipe par délégation de la programmation des modules à différentes personnes ou groupes de personnes. Une fois développés, les modules peuvent constituer une base de travail commune ;
8. *hiérarchisation des modules* : un programme peut d'abord être résolu globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous modules et ainsi de suite. De cette façon, nous obtenons une hiérarchie de modules. Les modules peuvent être développés en passant du haut vers le bas dans la hiérarchie *top-down development* - méthode du raffinement progressif) ou bien en passant du bas vers le haut *bottom-up development*).

En principe, la méthode du raffinement progressif est à préférer, mais en pratique, on aboutit souvent à une méthode hybride, qui construit un programme en considérant aussi bien le problème posé que les possibilités de l'ordinateur ciblé (*méthode du jo-jo*).

1.3 Déclaration et Définition d'une fonction

On appelle fonction un sous programme qui permet d'effectuer un ensemble d'instructions par simple appel de la fonction dans le corps du programme principal. Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).

1.3.1 La déclaration d'une fonction

La déclaration d'une fonction, ou son prototype, est une brève description de la dite fonction. Cette fonction sera décrite plus en détail, ou définie, plus loin dans le programme. On place donc le prototype en début de programme (avant la fonction principale main()).

Cette brève 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.

l

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). Un prototype de fonction ressemble donc à ceci :

```
Type_de_donnee_renvoyee Nom_De_La_Fonction(type_argument1, type_argument2, ...);
```

Le prototype est une instruction, il est donc suivi d'un point-virgule !
Exemples :

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

1.3.2 Définition

La fonction doit être ensuite définie avant d'être utilisée. La définition consiste en la description détaillée d'une fonction. La syntaxe est la suivante :

```
type_de_donnee Nom_De_La_Fonction(type1 argument1, type2 argument2, ...)
{
     liste d'instructions
}
```

Remarques :

1. type_de_donnee représente le type de valeur que la fonction est sensée retourner (`char`, `int`, `float` ...)
2. si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot clé `void`.
3. si aucun type de donnée n'est précisé (cela est très vilain !), le type `int` est pris par défaut.
4. le nom de la fonction suit les mêmes règles que les noms de variables :
   a) le nom doit commencer par une lettre ;
   b) un nom de fonction peut comporter des lettres, des chiffres et les caractères _ et & (les espaces ne sont pas autorisés !) ;
   c) le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules).
5. les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes
6. il ne faut pas oublier de refermer les accolades.
7. le nombre d'accolades ouvertes (fonction, boucles et autres structures) doit être égal au nombre d'accolades fermées !
8. la même chose s'applique pour les parenthèses, les crochets ou les guillemets !

Une fois cette étape franchie, votre fonction ne s'exécutera pas tant que l'on ne fait pas appel à elle quelque part dans la page !

1.4 Appel de fonction

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

```
Nom_De_La_Fonction();
```

Remarques :

– le point virgule signifie la fin d'une instruction et permet au navigateur de distinguer les différents blocs d'instructions ;
– si jamais vous avez défini des arguments dans la déclaration de la fonction, il faudra veiller à les inclure lors de l'appel de la fonction (le même nombre d'arguments séparés par des virgules !)
`Nom_De_La_Fonction(argument1, argument2);`

1.5 Mode de passage des paramètres

Le langage C propose 2 méthodes de passage des paramètres :

1.5.1 Le passage par valeur

Ce mode de passage fonctionne en établissant une copie exacte de la valeur qui doit être passée en paramètre. Ceci implique un résultat important : le contenu d'une fonction ne peut en aucun cas changer la valeur d'une variable passée en paramètre. Cela est logique car l'on en réalise une copie.

```
void essai(int value)
{
    value++;
}

void main(void)
{
    int a=10;

    essai(a);
    printf("La valeur de a vaut : %d\n",a);
}
```

Attention, ce mode de transfert de paramètres présente des avantages, mais aussi des inconvénients. Pour vous en persuader, que pensez vous qu'il se passe lors de l'appel d'une fonction comme la suivante ?

```
void essai(int value[10000])
{
        /* Le reste du code */
}
```

Le passage par valeur assure que le tableau de `10000 entrées` (ce qui représente un bon paquet d'octets) est recopié dans son intégralité. Outre le fait que cette copie coûte du temps, il faut aussi disposer de suffisamment d'espace libre en mémoire. Autre détail, ne pensez surtout pas à écrire une fonction récursive (ce terme sera expliqué plus loin dans ce chapitre) prenant des gros tableaux en paramètres : vous satureriez la mémoire en moins de temps qu'il ne faut pour le dire.

1.5.2 Le passage par référence

Contrairement au passage par valeur, le passage par référence, n'effectue pas de copie du paramètre : il passe une référence sur celui-ci. En conséquence, une fonction

passant des paramètres par référence ne pourra jamais prendre (en paramètre) une constante, mais uniquement une variable. Ce mode de passage peut donc être intéressant, surtout si un paramètre est important en terme de taille (un tableau, une structure, ...). Voir le chapitre sur les pointeurs pour plus de détails.