

Chapitre 9:

Les fonctions en C

Définition et exemples

- Comme tous les langages, C permet de découper un programme en plusieurs parties nommées souvent « modules ».
- Cette programmation dite « modulaire » se justifie pour de multiples raisons :
 - ✓ Le code d'un programme devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte.
 - ✓ La programmation modulaire permet d'éviter des séquences d'instructions répétitives.
 - ✓ La programmation modulaire permet le partage des outils communs. Il suffit d'avoir la mise au point une seule fois de ces outils.

Définition et exemples

- 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.

Définition et exemples

- Pour élaborer une fonction 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.

Définition et exemples

- 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

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

Définition et exemples

Exemple

```
#include <stdio.h>
#include <conio.h>

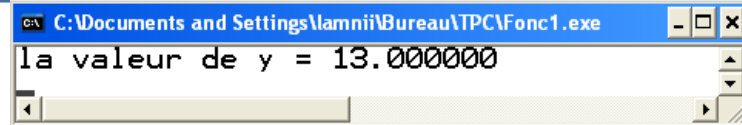
/* Programme principal (fonction main) */
int main(int argc, char *argv[])
{
    float Fpoly(float, int, int); /* Déclaration de la fonction Fpoly */
    float x=2.5;
    float y;
    int m=4, n=3;

    /* Appel de la fonction Fpoly avec les arguments x, n et p */

    y=Fpoly(x,m,n);

    printf("la valeur de y = %f\n",y);
    getch();
}

/* La fonction Fpoly */
float Fpoly(float x, int a, int b)
{
    float val;
    val=a*x+b;
    return(val);
}
```

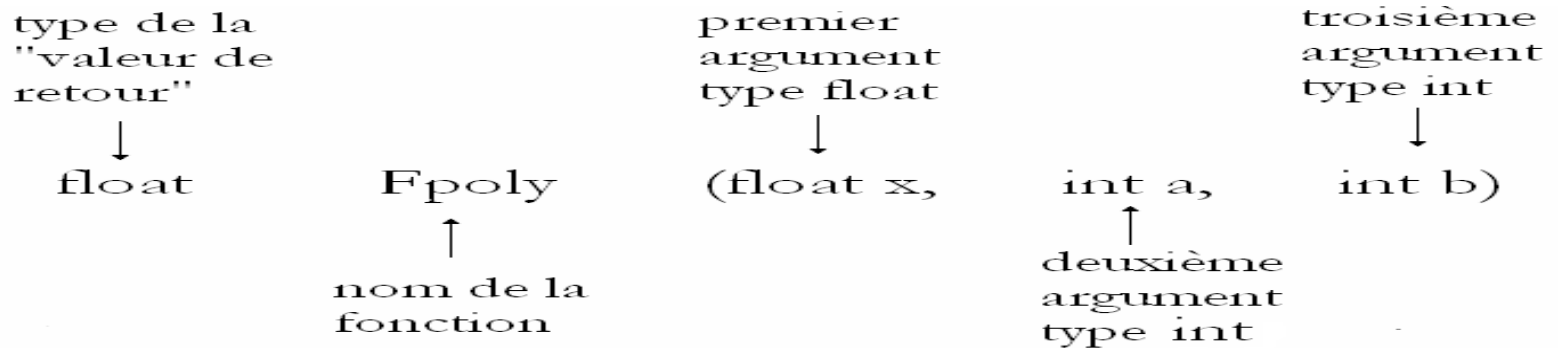


Définition et exemples

- Nous y trouvons face à un programme formé de deux blocs :
 - ✓ Programme principal (la fonction main).
 - ✓ La fonction **Fpoly**.
- En général la définition d'une fonction à une structure voisine de la fonction « main », à savoir un en-tête et un corps délimité par des accolades({ et }).
- Détaillons notre exemple :

Définition et exemples

- l'en-tête de la fonction **Fpoly** est plus élaboré que celui de la fonction main:



Définition et exemples

- Les noms des arguments n'ont d'importance **qu'au sein** du corps de la fonction.
- **float val** ; simple déclaration ; mais, on dit que val est une « variable locale » à la fonction Fpoly, de même que les variables m, n et y sont des variable locales à la fonction main.
- L'instruction **val=a*x+b;** est une simple affectation.
- Enfin, l'instruction **return(val);** précise la valeur que fournira la fonction à la fin de son travail.

Définition et exemples

- En définitive, on peut dire que **Fpoly** est une fonction telle que **Fpoly(x,a,b)** fournisse la valeur du polynôme $p(x)=a*x+b$.
- Notez bien l'aspect arbitraire du nom des arguments.
- On obtient la même définition de la fonction **Fpoly**, avec, par exemple :

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

Définition et exemples

- Examinons maintenant la fonction main. Vous constatez qu'on y trouve une déclaration:
`float Fpoly (float ,int ,int);`
Elle sert à prévenir le compilateur que `Fpoly` est une fonction donnée de 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

- **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.

- **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 reprises 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 instructions de conversion.

Exercices

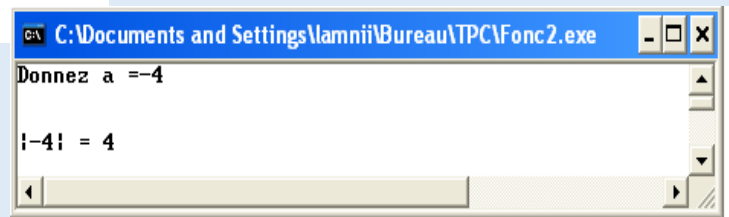
1. Déclarez et appelez une fonction qui calcul la valeur absolue d'entier.
2. On dispose de 4 entiers. Calculez l' entier le plus grand. Pour cela : Déclarer et appeler plusieurs fois une fonction qui compare deux entiers et qui renvoie le plus grand.

Solution-1

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
    int Absolu1(int);
    int a,b;
    printf("Donnez a =");
    scanf("%d",&a);
    b=Absolu1(a);

    printf("\n\n| %d | = %d\n",a,b);
    getch();
}

int Absolu1(int x)
{
    if (x>0)
        return(x);
    else
        return(-x);
}
```



Solution-2

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
    int Copar1(int, int);
    int a,b,c,d;
    int e,f,G;
    printf("Donnez a =");
    scanf("%d", &a);
    printf("\nDonnez b =");
    scanf("%d", &b);
    printf("\nDonnez c =");
    scanf("%d", &c);
    printf("\nDonnez d =");
    scanf("%d", &d);
    e=Copar1(a,b);
    f=Copar1(c,d);
    G=Copar1(e,f);
    printf("\n\nLa plus grande valeur de %d, %d, %d, %d egal a %d\n",a,b,c,d,G);
    getch();
}

int Copar1(int x, int y)
{
    if (x>y)
        return(x);
    else
        return(y);
}
```


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
 - ✓ 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.
 - ✓ Déclaration : `void FoncSansArRt(void);`
 - ✓ En-tête: `void FoncSansArRt(void);`

Définition

- 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.

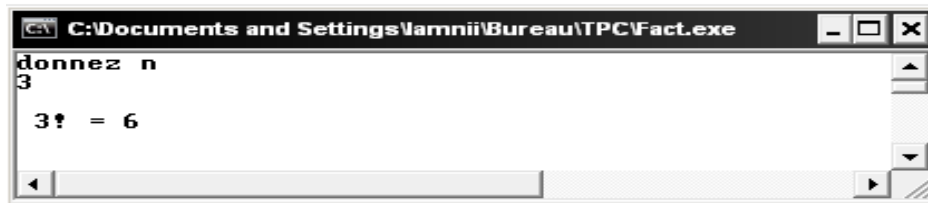
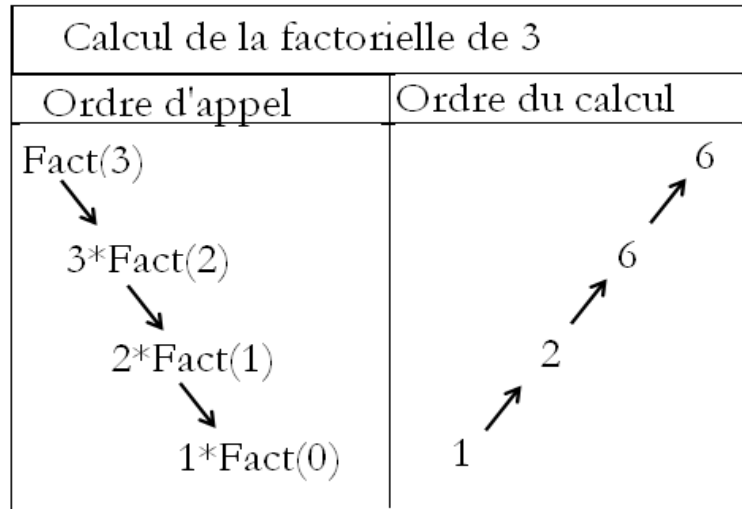
Exemple

Exemple

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main(int argc, char *argv[])
{
    int Fact(int);
    int n;
    printf("donnez n\n");
    scanf("%d", &n);
    printf("\n %d! = %d\n", n, Fact(n));
    getch();
}

int Fact(int n)
{ if (n==0)
    return(1);
  else
    return(n*Fact(n-1));
}
```

Exemple



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

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

Exemple

- Calcul du $n^{\text{ème}}$ terme de la suite de **Fibonacci**

$$u_0 = 0$$

$$u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2}, \quad n \geq 2$$

Exemple

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main(int argc, char *argv[])
{
    int Fib(int);
    int n;
    printf("Le terme d'indice n=");
    scanf("%d", &n);
    printf("vaut %d\n", Fib(n));
    getch();
}

int Fib(int i)
{
    if(i<2)
        return(i);
    else
        return(Fib(i-1)+Fib(i-2));
}
```

Problème

Problème

```
#include <stdio.h>
main()
{ void echange (int a, int b) ;
  int n=10, p=20 ;
  printf ("avant appel    : %d %d\n", n, p) ;
  echange (n, p) ;
  printf ("après appel    : %d %d", n, p)
}
void echange (int a, int b)
{
  int c ;
  printf ("début echange : %d %d\n", a, b) ;
  c = a ;
  a = b ;
  b = c ;
  printf ("fin echange   : %d %d\n", a, b) ;
}
```

```
avant appel :      10  20
début echange :    10  20
fin echange :      20  10
après echange :    10  20
```

Solutions

Ce problème possède plusieurs solutions, à savoir :

- Transmettre en argument la valeur de **l'adresse d'une variable**. La fonction pourra éventuellement agir sur le contenu de cette adresse. C'est précisément ce que nous faisons lorsque nous utilisons la fonction **scanf**.
- Utiliser des **variables globales**.

transmission par un pointeur

Solution-1

```
#include <stdio.h>
main()
{
    void echange (int * ad1, int * ad2) ;
    int a=10, b=20 ;
    printf ("avant appel %d %d\n", a, b) ;
    echange (&a, &b) ;
    printf ("après appel %d %d", a, b) ;
}

void echange (int * ad1, int * ad2)
{
    int x ;
    x = * ad1 ;
    * ad1 = * ad2 ;
    * ad2 = x ;
}
```

avant l'appel 10 20

après appel 20 10

Variables globales

- Une **variable globale** est une variable dont le nom et la valeur seront reconnus dans l'ensemble du programme et pendant toute la durée de vie de celui-ci.
- Une **variable globale** est déclarée en dehors de toute fonction et avant la fonction **main**.
- Les **variables globales** sont accessibles à toutes les fonctions depuis leurs déclarations jusqu'à la fin du programme.

Variables globales

Exercice

Proposer une solution au problème d'échange vu précédemment en utilisant des variables globales

Définition et exemple

- Les **variables globales** peuvent être initialisées, et leurs initialisation se fait avant le début de la fonction main.
- Si une variable n'est pas initialisée par le programmeur alors celle-ci sera initialisée automatiquement à la valeur **0**.

Exemple

```
#include <stdio.h>
#include <conio.h>
int n;
float p=3.5;
int main(int argc, char *argv[])
{
    int F1(void);
    printf("n=%d, p=%f\n",n,p);
    printf("F1()=%d",F1());
    getch();
}

int F1(void)
{
    int p=55;
    printf(" p de F1 est local =%d\n",p);
    return(p);
}
```

Variables locales

- Une **variable locale** est une variable dont le nom et la valeur ne seront reconnus qu'à l'intérieur d'une fonction ou à l'intérieur d'un bloc d'instructions.
- Une **variable locale** est visible à partir du point où elle est déclarée et jusqu'à la fin du bloc de la fonction dans laquelle elle est déclarée.
- Une **variable locale** est réservée et initialisée dès qu' on rentre dans son espace de visibilité, et elle se détruit dès que l'on quitte.

Exemple

Exemple

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int x=2;
int f(int);
int g(int);
int main(int argc, char *argv[])
{ printf("x=%d\n", x);
  { int x=6;
    printf("x=%d\n", x);
  }
  printf("f(%d)=%d\n", x, f(x));
  printf("g(%d)=%d\n", x, g(x));
  getch();
}
int f(int a)
{ int x=9;
  return(a+x);
}
int g(int a)
{
  return(a*x);
}
```



A screenshot of a Windows command prompt window titled "C:\Documents and Sett...". The window displays the output of the C program:
x=2
x=6
f(2)=11
g(2)=4

Définition et exemple

- Une **variable statique** est une variable persistante : elle garde sa valeur à la sortie d'une fonction ou d'un bloc.
- Elle est déclarée en utilisant le mot-clé **static**. Elle n'est pas allouée qu'une seule fois et au début du programme. Cependant, elle n'est accessible qu'à l'intérieur de son espace de visibilité (un bloc ou une fonction).
- Une **variable statique** non initialisée prend par défaut la valeur 0.

Définition et exemple

Exemple

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int f(void);
int g(void);
static int a=5;
int main(int argc, char *argv[])
{
    printf("a=%d\n",a);
    printf("a=%d\n",g());
    printf("b=%d\n",f());
    printf("a=%d\n",g());
    printf("b=%d\n",f());
    printf("a=%d\n",g());
    printf("a=%d\n",a);
    getch();
}
int f(void)
{
    static int b;
    b++;
    a++;
    return(b);
}
int g(void)
{
    int b=3;
    a=a+b;
    return(a);
}
```


Définition

- Syntaxe de la définition d'un pointeur de fonction :
`Type (*pointeur)(liste-des-paramètres-ou-leur-type);`
- Exemple:
`int f(void); /*fonction renvoyant un int*/`
`int *f(); /*fonction qui retourne un pointeur sur un int*/`
`int (*pf) (void); /*pointeur de fonction sans argument renvoyant un int */`
- Le nom d'une fonction est traduit par le compilateur en adresse d'une fonction.
- Les pointeurs de fonction permettent de passer des fonctions en arguments à une fonction.

Exemple

Exemple

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
    int x,z;
    int f1(int);
    int f2(int);
    int f3(int);
    int (*pf)(int);
    printf("Donnez un entier (1,2 ou 3) x=");
    scanf("%d",&x);
    if (x==1)
        pf=f1;
    else
        if (x==2)
            pf=f2;
        else
            pf=f3;
    z=(*pf)(x); /* appel et affectation */
    printf("pour x=%d on a z=%d",x,z);
    getch();
}
int f1(int n)
{
    return(2*n);
}
int f2(int n)
{
    return(n*n);
}
int f3(int n)
{
    return(n*n+2*n+3);
}
```

Cas unidimensionnel

- Lorsque l'on place le nom d'un tableau en argument effectif de l'appel d'une fonction, on transmet finalement l'adresse du tableau à la fonction, ce qui lui permet d'effectuer toutes les manipulations voulues sur ses éléments, qu'il s'agisse d'utiliser leur valeur ou de la modifier. Voyons quelques exemples pratiques.
- Voici un exemple de fonction qui met la valeur 1 dans tous les éléments d'un tableau de 10 éléments,

```
void fct (int t[10])  
{  
    int i ;  
    for (i=0 ; i<10 ; i++) t[i] =1 ;  
}
```

Cas unidimensionnel

- Voici deux exemples d'appels possibles de cette fonction :

```
int t1[10], t2[10] :
```

```
fct(t1) ;
```

```
fct(t2) ;
```

- L'en-tête de fct peut être indifféremment écrit de l'une des manières suivantes :

```
✓ void fct (int t[10])
```

```
✓ void fct (int * t)
```

```
✓ void fct (int t[])
```

- La connaissance de la taille exacte du tableau n'est pas indispensable au compilateur.

Cas multidimensionnel

- Voici un exemple de fonction qui met la valeur 1 dans un tableau de dimension 10 et 15 :

```
void f2(int t[10][15])
{ int i, j ;
  for (i=0 ; i<10 ; i++)
    for (j=0 ; j<15 ; j++)
      t[i][j] = 1 ;
}
```

- Appel:

```
int t1[10][15] ;
f2(t1) ;
```

- L'en-tête de f2 aurait pu être:

```
void f2 (int t[][15]) mais pas void fct (int t[][]).
```