Programmation C: Fonction

Ibrahim ALAME

ESIEE

27/11/2023

Fonctions

Définition

Une fonction est une suite d'instructions, mais vue du programme principal **main()**, elle représente une seule action. Deux types de fonction:

- Les fonctions prédéfinies **scanf, printf, strcpy**...
- les fonctions personnelles

Utilité des fonctions:

- Supprimer les répétitions de code
- Structurer des blocs de code indépendants (maintenance)
- Création de bibliothèques de fonctions

Quelques exemples

Une définition de fonction spécifie :

- Le type de la valeur renvoyée par la fonction
- Le nom de la fonction
- Les paramètres (argument) qui sont passés à la fonction
- Les variables locales et externes utilisées par la fonction
- D'autre fonctions appelées éventuellement par la fonction
- Les instructions que doit exécuter la fonction

Syntaxe d'une fonction:

```
<type_iden_sortie> iden_Fct( <type> ien_1, ... ,<type> iden_N){
        /* déclaration  des variables locales */
        <type>  iden_2,...,iden_Z;
        //
        // Traitement
        //
        /* renvoi le résultat dans le paramètre de sortie*/
        return(valeur);
    }
```

Quelques exemples

Exemple d'une fonction:

```
double carre (double C ) {
    double resultat;   /* déclaration  d'une variable locale */
    resultat = C * C;
    /* renvoi le résultat dans le paramètre de sortie*/
    return (resultat);
}
```

Autres caractéristiques des fonctions :

- Une fonction peut en appeler une autre. La fonction appelée doit être déclarée avant celle qui appelle.
- On ne peut pas déclarer une fonction à l'intérieur d'une autre fonction.
- Une fonction possède un seul point d'entrée, mais éventuellement plusieurs de sortie.

Où déclarer une fonction? : avant le main()

```c
#include <stdio.h>

double carre (double C ) {
    double resultat; // déclaration  d'une variable locale
    resultat = C * C;
    return( resultat ); // renvoi le résultat à la sortie
    }

int main(){
        double A, B=2;
         A = carre (B );
        printf ("Le carré de %f est %f",B,A);
    return 0;
}
```

Où déclarer une fonction? : après le main()

```c
#include <stdio.h>

double carre (double C ) ; // PROTOTYPE DE LA FONCTION

int main(){
        double A, B=2;
         A = carre (B );
        printf ("Le carré de %f est %f",B,A);
    return 0;
}

double carre (double C ) {
    double resultat; // déclaration  d'une  variable locale
    resultat = C * C;
    return( resultat ); // renvoi le résultat à la sortie
}
```

Déclaration PROTOTYPE

Le prototype de la fonction définie l'interface de la fonction. Lors de la compilation, le compilateur accepte l'appel à une fonction seulement s'il à rencontré au moins le prototype de la fonction. Dans la déclaration prototype il est possible d'omettre les noms des variables mais pas leur type.

```c
#include <stdio.h>
double carre (double) ; // PROTOTYPE SANS NOM DE VARIABLE
int main(){
        double A, B=2;
         A = carre (B );
        printf ("Le carré de %f est %f",B,A);
    return 0;
}
double carre (double C ) {
    double resultat; // déclaration  d'une  variable locale
    resultat = C * C;
    return( resultat ); // renvoi le résultat à la sortie
}
```

Les fonctions et les échanges de variables

Deux types:

- Les fonctions sans passage de paramètres et ne renvoyant rien au programme :

```
void UneFonction ( void );
```

- Les fonctions avec passage de paramètres et renvoyant un objet au programme :

```
int UneFonction ( int A ):
```

Les fonctions et les échanges de paramètres : Par des variables globales

Une variable globale est déclarée au début du programme. Cette variable sera visible dans toutes les fonctions

Exemple

```
#include <stdio.h>

int X; /* déclaration de X en variable globale */

void main() {
...

}
```

Les fonctions et les échanges de paramètres : Par des variables locales

- Une variable locale est déclarée au début d'une fonction.
- Cette variable sera visible uniquement dans la fonction.
- Une variable globale portant le même nom sera invisible dans la fonction.
- Les variables locales ne sont pas initialisées par défaut et perdent leur valeur à chaque nouvel appel de la fonction

```
#include <stdio.h>
int X,A;  /* déclaration de X et A en variable globale */
int MaFonction( int A) {  /* déclaration de A en variable locale */
    static int X;         /* déclaration de X en variable locale */
        . . .
    return (X);
}
void main(void) {
        . . .
}
```

Échange entre les fonctions par variables globales

```c
#include <stdio.h>

int A,B;
void cube(void);

int main(void) {
    A=2; // initialisation de A à 2
    int B; /* déclaration locale de B */
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=0  */
    cube();
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=0   */
}

void cube() {
    B = A*A*A;
    return;
}
```

Échange entre les fonctions par variables locales

```c
#include <stdio.h>

int cube(int);

int main(void) {
    int A=2; //déclaration locale et initialisation de A à 2
    int B; /* déclaration locale de B */
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=??   */
    B = cube(A);
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=8    */
}

int cube(int X) { /* X est un paramètre de la fonction cube */
                  /* X récupère la valeur de A du main */
        int resultat;
    resultat = X*X*X;
    return (resultat);
}
```

Échange entre les fonctions par variables locales

```c
#include <stdio.h>
void cube(int);
int main(void) {
    int A=2; //déclaration locale et initialisation de A à 2
    int B; /* déclaration locale de B */
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=??   */
    cube(A);
    printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=??   */
}

void cube(int A) { /* A est un paramètre de la fonction cube */
                   /* A récupère la valeur de A du main */
        int B;
        B = A*A*A;
        printf("A=%d, B=%d\n",A,B);          /* affiche : A=4, B=8   */
        return ;
}
```

Échange entre les fonctions par variables locales

```c
#include <stdio.h>
void cube(int, int*);
int main(void) {
    int A=2; /* déclaration locale de A et B */
    int B;
    printf("A=%d, B=%d\n",A,B);/* affiche : A=4, B=??   */
    cube( A , &B);//On donne à la fonction cube le contenu de A
                  // et l'adresse de de la variable B
    printf("A=%d, B=%d\n",A,B);/* affiche : A=4, B=??   */
}

void cube(int A, int *B) {/* A est un paramètre de la fonction*/
                /* B récupère un pointeur sur un entier */
    printf("> A=%d, B=%d\n",A,*B);/* affiche : A=4, B=??   */
    *B = A*A*A;
    printf("> A=%d, B=%d\n",A,*B);/* affiche : A=4, B=8   */
    return;
}
```

Fonction récursive

- Une fonction récursive est une fonction qui s'appelle elle-même.
  - Directement (si la fonction P appelle directement P , on dit que la récursivité est directe).

```
int f1(...){
    x=f1(...);
    return ...;
}
```

  - Indirectement à travers une ou plusieurs fonctions relais (si P appelle une fonction P_1 , qui appelle une fonction P_2 , ... , qui appelle une fonction P_n et qui enfin appelle P , on dit qu'il s'agit d'une récursivité indirecte).

```
int f1(...){
    x=f2(...);
    return ...;
}
```

```
int f2(...){
    x=f1(...);
    return ...;
}
```

Fonction récursive

- La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques.
- Elle permet:
  - d'écrire des programmes beaucoup plus lisibles;
  - d'écrire d'une manière très rapide (par rapport d'une manière itérative);
  - d'utiliser le principe diviser-pour-résoudre.
- La récursivité utilise toujours la pile du programme en cours.
- Dans une fonction récursive, toutes les variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs.
- La pile se remplit progressivement, et si on ne fait pas attention on arrive à un "débordement de pile". Ensuite, les variables sont désempilées.
- Toute fonction récursive comporte une instruction (ou un bloc d'instructions) nommée "point terminal" ou "point d'appui" ou "point d'arrêt", qui indique que le reste des instructions ne doit plus être exécuté.

Fonction récursive : Syntaxe

- La fonction récursive est composée de deux partie: une partie strictement récursive et une partie non récursive (base) servant de point de départ à l'utilisation de la définition récursive.
- Structure générale d'une fonction récursive :

```
if(/* !condition de convergence */)
    exit(1);
if(/*condition d'arrêt*/)
    return(/*Ce qu'elle doit retourner*/);
else
/*Traitement*/
/*appel récursif */
/*Traitement*/
```

Fonction récursive : Exemple

- Soit le factoriel $n! = 1 \times 2 \times 3 \cdots \times n$ pour n entier naturel. Par convention $0! = 1$.
- Par récurrence le factoriel est calculé à l'aide de la suite récurrente:

$$\begin{cases} u_n = n \times u_{n-1} & \text{pour } n \geq 1 \\ u_0 = 1 \end{cases}$$

```c
#include <stdio.h>
#include <stdlib.h>
int u(int n){
    if(n<0) exit(EXIT_FAILURE);
    else
        if(n==0) return 1;
        else
            return n*u(n-1);
}
int main(void) {
    printf("n!=%d\n",u(5));
}
```

Fonction récursive : Variables locales, arguments de fonctions

- Lorsqu'une fonction récursive définit des variables locales, un exemplaire de chacune d'entre elles est crée à chaque appel récursif de la fonction.
- Il en est de même des arguments des fonctions.
- *Exemple* - considérons une fonction **void miroir()** qui lit caractère par caractère une chaîne terminée par '?' et l'affiche dans l'ordre inverse de celui de la lecture.

Fonction itérative : Variables locales, arguments de fonctions

```c
#include <stdio.h>
void miroir() {
    char s[20],c;
    int i=0;
    while( (c=getchar())!='?')
        s[i++]=c;
    s[i]='\0';
    while (--i>=0)
        putchar (s[i]);
}
void main() {
    printf("Entrer les caracteres avec ? a la fin.\n");
    miroir();

}
```

Fonction récursive : Variables locales, arguments de fonctions

```c
#include <stdio.h>
void miroir() {
     int c = getchar();
     if (c != '?') {
          miroir();
          printf("%c", c);
     }
 }
 void main() {
     printf("Entrer les caracteres avec ? a la fin.\n");
     miroir();

 }
```

Fonction récursive : Variables locales, arguments de fonctions

`miroir() c='1'`			
	`miroir() c='2'`		
		`miroir() c='3'`	
			`miroir() c='?`
			destruction de c retour
		`printf("%c",c)`	
		destruction de c retour	
	`printf("%c",c)`		
	destruction de c retour		
`printf("%c",c)`			
destruction de c retour			

Fonction récursive : Dangers et précautions

- Débordement de pile (Stack Overflow)
- Dans la pile sont non seulement stockés les valeurs des variables de retour mais aussi les adresses des fonctions entre autres choses, les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque sans conteste une sortie anormale du programme.
- Si vous êtes presque sûr de dépasser ce genre de limites, préférez alors une approche itérative plutôt qu'une approche récursive du problème.

Fonction récursive ; Un piège subtil: la suite de Fibonacci

- Écrire une fonction qui calcule le n-ième terme de la suite définie par

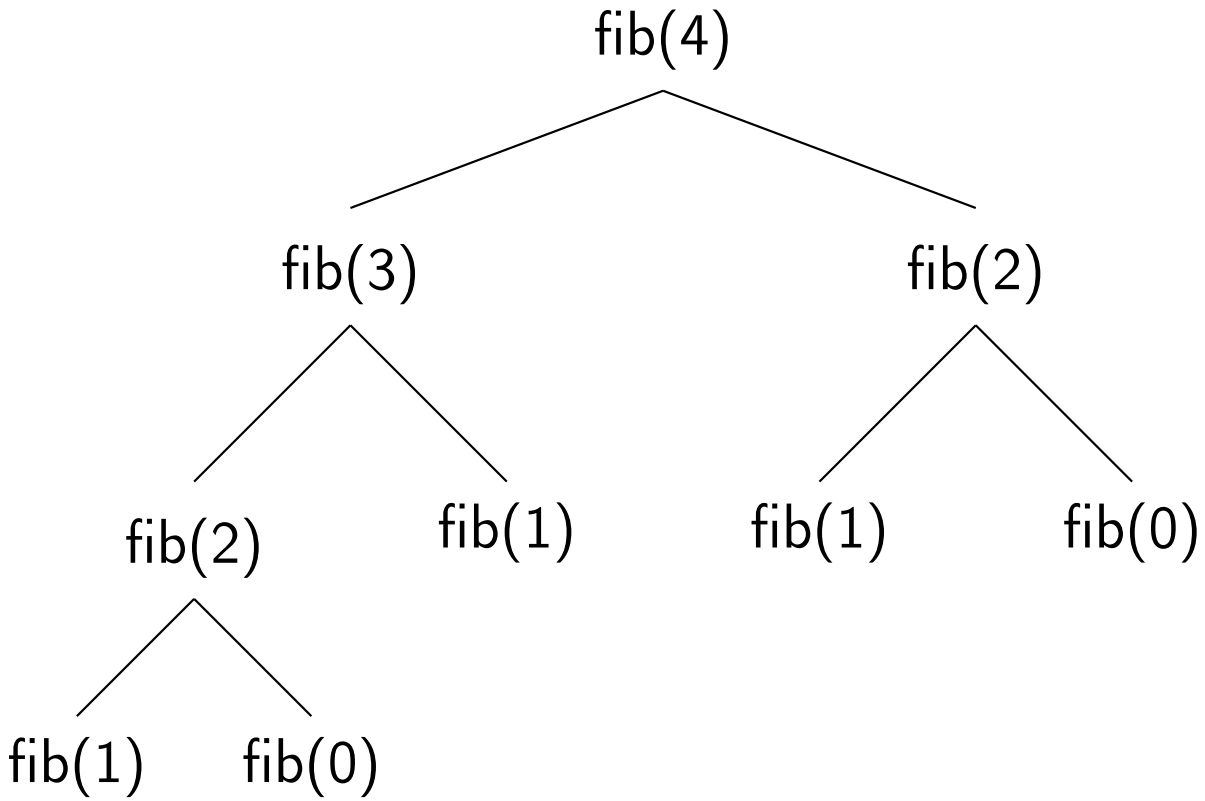
$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 2 \end{cases}$$

- Le programme marche, il termine. Le problème se situe dans le nombre exponentiel d'appels à la fonction.

```
#include <stdio.h>
long fib(int n){
    if(n <= 1)
        return n; // cas de base
    else
        return fib(n-1)+fib(n-2);  // cas général
 }
 void main() {
     int n=40;
     printf("fib(%d)=%ld\n",n,fib(n));
 }
```

Fonction récursive ; Un piège subtil: la suite de Fibonacci

- Le programme marche, il termine. Le problème se situe dans le nombre exponentiel d'appels à la fonction.

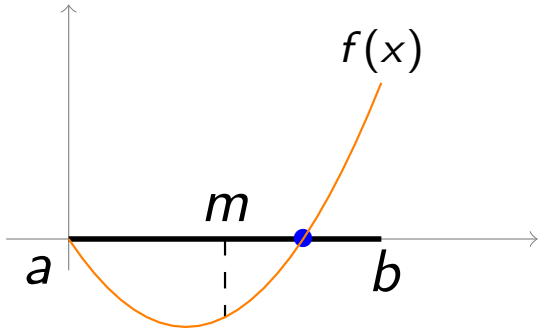


Fonction récursive ; la suite de Fibonacci: version itérative

```c
#include <stdio.h>
long fib(int n){
    int i;
    long u, v, w;
    u = 0; v = 1; // u = F(0); v = F(1)
    for(i = 2; i <= n; i++){
        w = u+v; // u = F(i-2); v = F(i-1)
        u = v;
        v = w;
    }
    return v;
}
void main() {
     int n=50;
     printf("fib(%d)=%ld\n",n,fib(n));
 }
```

Fonction récursive ; Diviser pour résoudre

- Quand on ne sait pas résoudre un problème, on essaie de le couper en morceaux qui seraient plus faciles à traiter.
- Exemple - Recherche d'une racine par dichotomie
- On suppose que $f : [a, b] \to \mathbb{R}$ est continue et telle que $f(a) < 0, f(b) > 0$. Il existe une racine x_0 de f dans l'intervalle $[a, b]$, qu'on veut déterminer de sorte que $|f(x_0)| \leq \varepsilon$ pour ε donné. On calcule $f(m)$ où $m = (a + b)/2$. En fonction de son signe, on explore $[a, m]$ ou $[m, b]$.



Fonction récursive ; La méthode de dichotomie

```c
#include <stdio.h>
#include <math.h>
double f(double x){
    return x*x-2;
}
double racineDicho(double a, double b, double eps){
    double m = (a+b)/2, fm = f(m);
    if(fabs(fm) <= eps)
        return m;
    if(fm < 0) // la racine est dans [m, b]
        return racineDicho(m, b, eps);
    else // la racine est dans [a, m]
        return racineDicho(a, m, eps);
    }
void main(){
    double a=0,b=2,eps=0.0001;
    printf("La racine=%lf\n",racineDicho(a,b,eps));
}
```

Fonction récursive ; Quand utiliser la récursivité

- Quand il existe une définition récursive claire.
- Quand la récursivité est plus simple que la version itérative.
- Quand on a besoin d'un gain en performance possible grâce à une formulation récursive habile.
  - Généralement rapide et simple pour le développement.
  - Temps de mise au point ("débogage") peut être long.
  - Elle est lourde à l'exécution

Fonction récursive ; Exemple

$$S_n = \underbrace{1 + 2 + \cdots + (n-1)}_{S_{n-1}} + n$$

D'où la définition

$$\begin{cases} S_n = S_{n-1} + n & \text{si } n \geq 1 \\ S_0 = 0 \end{cases}$$

```c
#include <stdio.h>
int S(int n){
    if(n==0) return 0;
    return n+S(n-1);
}
void main(){
    int n=10;
    printf("S(%d)=%d\n",n,S(n));
}
```

Fonction récursive ; Exemple

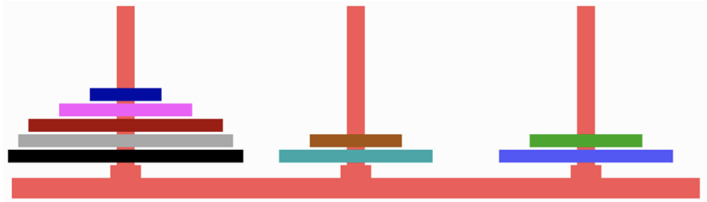
$$ListerJusqua(n) = \underbrace{0, 1, 2, \cdots, (n-1)}_{ListerJusqua(n-1)}, n$$

$$\begin{cases} ListerJusqua(n) = ListerJusqua(n-1), afficher(n) & \text{si } n \geq 1 \\ ListerJusqua(0) = afficher(0) \end{cases}$$

```c
#include <stdio.h>
int L(int n){
    if(n==0) printf("%d ",0);
    else{
        L(n-1);
        printf("%d ",n);
    }
}
void main(){
    int n=10;
    L(n); }
```

Fonction récursive ; Tour-de-Hanoï

```c
#include <stdio.h>
void hanoi(int n,
           char D, char F, char I){
    if(n>0){
        hanoi(n-1,D,I,F);
        printf("%c --> %c\n",D,F);
        hanoi(n-1,I,F,D);
    }
}
int main(){
    hanoi(3,'A','B','C');
    return 0;
}
```



```
A ----> B
A ----> C
B ----> C
A ----> B
C ----> A
C ----> B
A ----> B


...Program finished with exit code 0
Press ENTER to exit console.
```

Fonction récursive ;coût de Tour-de-Hanoï

```c
#include <stdio.h>
unsigned long c(int n){
    if(n==1) return 1;
    else return 2*c(n-1)+1;
}
int main(){
    int i;
    for(i=1;i<30;i++)
        printf("%ld ",c(i));
    return 0;
}
```

```
1 3 7 15 31 63 127 255 511 1023 2047 4095 819
1 16383 32767 65535 131071 262143 524287 1048
575 2097151 4194303 8388607 16777215 33554431
 67108863 134217727 268435455 536870911

...Program finished with exit code 0
Press ENTER to exit console.
```