Introduction à la programmation (C)

Cours 3 : fonctions

Fonction

Regrouper des instructions effectuant un calcul

Plusieurs arguments (entrées)

Un résultat

```
type_retour nom_fonction (type1 arg1, type2 arg2 …)
{
    // code de la fonction
    return result ;
}
```

Fonction

```
type_retour nom_fonction (type1 arg1, type2 arg2 …)
{
    // code de la fonction
    return result ;
}
```

```
float puissance (float x, int n)
{
    int i ;
    float res=1 ;
    for(i=0 ; i < n ; i=i+1)
    {
        res = res * x ;
    }
    return res ;
}
```

} Déclarations de variables locales

} Code

} Retour du résultat

Type des arguments

Types simples : int, float, double, char ...

```
int ma_fct (float x, char c)
```

Tableaux : type arg[]

```
int ma_fct (float T[ ], char c)
```

Rien/vide : void

```
void ma_fct (float T[ ], char c)
double ma_fct2(void)
```

Arguments tableaux bi-dimensionnels

Matrice : type arg[][NC]

int ma_fct (float T[ ][NC], char c)

Techniquement les tableaux bi-dimensionnels sont « vectorisés » (codés comme des tableaux avec tous les coefficients à la suite).
Donc il est nécessaire de passer le nombre de colonnes aux fonctions !
Pour l'éviter : pointeurs (cf. §4) ...

Passage des arguments (par valeur)

Déclaration / implémentation

```
void ma_fct (float x)
{
    x = x+1 ;
}
```

Cette fonction ne fait rien
car seule la valeur de a est passée
↓
pointeurs (§4)

Appel

```
int main ()
{
    float a = 1 ;
    ma_fct(a) ;
}
```

En C

Passage par valeur
On passe la valeur contenue dans a

Passage par référence
On passe la « localisation » de a en mémoire

Passage des arguments (par valeur)

Règle sur les arguments (v1)

Une fonction

* ne peut pas modifier la valeur d'une variable passée en argument
* sauf si c'est un tableau (cf. §4).

→ Pour cela il faudra des pointeurs

Variables locales

Chaque fonction a des variables locales

```
float ma_fct (float x, float y)
{
    float z = x * x, t = y * y ;
    return z/t ;
}
```

Arguments : variables locales initialisées par la valeur reçue

Variables déclarées en tête de fonction

Les variables locales sont détruites en fin de fonction

Passage des arguments (par valeur)

Déclaration / implémentation

```
void ma_fct (float x)
{
    x = x+1 ;
}
```

Variable locale x initialisée avec la valeur de a (soit 1)

Destruction de x à la fin de la fonction

Appel

```
int main ()
{
    float a = 1 ;
    ma_fct(a) ;
}
```

Cette fonction ne fait rien
car seule la valeur de a est passée
↓
pointeurs (§4)

Déclaration / implémentation des fonctions

Le C permet de distinguer :

* Déclaration de la fonction (simplement son prototype)
  - A la fin de l'entête
* Implémentation de la fonction (son code)
  - L'ordre n'a alors pas d'importance

```c
#include <stdio.h>
#define N 10

void print_tab (float T[], int n) ;

int main ()
{
    float tab[N] ;
    for (i=0;i<N;i=i+1)
    {
        tab[i] = (2*i+5)%11 ;
    }
    print_tab (tab,N) ;
    return 0 ;
}

void print_tab (float T[], int n)
{
    int i ;
    for(i=0 ; i < n ; i=i+1)
    {
        printf("%d" ,tab[i]) ;
    }
    printf("\n") ;
}
```

Fonctions récursives

Le C supporte les fonctions récursives :
fonctions s'appelant elles-mêmes

Cas terminaux

Cas général (récursif)

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

Tout le problème est de les concevoir ... méthodologie

Récursive : pas de while/for, mais appel de la fonction par elle-même !

Méthodologie pour les fonctions récursives

1. Brouillon :
   * Exprimer la relation récursive dans le cas général
   * En déduire le/les cas terminal/terminaux

2. Implémentation :
   * Commencer par les cas terminaux
   * Puis le cas général

Exigé en examen

Fonction puissance

Calcul de x^n pour $n \geqslant 0$

1. Brouillon

Cas général

$$x^n = x \cdot x^{n-1}$$

Cas terminaux

* Pour $n = 0$ renvoyer 1
* Pour $n < 0$ erreur

2. Implémentation

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

float puissance (float x, int n)
{
    // Cas terminaux
    assert (n>=0) ; // erreur si n < 0
    if (n == 0)
        { return 1 ; }
    // Cas général
    else
        { return x * puissance(x,n-1) ; }
}
```

On peut fait bien plus efficace (cf. TD)