

X. Les fonctions

1. Généralités
2. Ecrire une fct
3. Portée d'une variable
4. Communication entre fcts
5. Tableau en paramètre
6. Bibliothèques standard
7. Fonctions récursives

1. Généralités

Une fonction ou procédure :

- Porte un nom
- Accepte 0, 1 ou plusieurs paramètres
- Exécute une suite d'instructions
- Une fonction retourne un résultat (i.e. elle peut apparaître dans une expression)
- Une procédure ne retourne pas de résultat

$y = \underline{\sin(\theta)} * \text{rayon}$

Nom
Paramètre(s)
Résultat

Intérêt des fonctions et procédures :

- ➔ Ecrire une seule fois un algorithme utilisé plusieurs fois
- ➔ Disposer d'un algorithme paramétré (valeurs fixées au moment de l'appel)
- ➔ Structurer un programme pour en faciliter la conception (1 tâche -> 1 procédure)

Définir une fonction : **écrire** le code sans connaître la valeur des paramètres

Appeler une fonction : **utiliser** ce code avec des paramètres/valeurs particuliers

2. Ecrire une fonction

Peut se faire en 3 étapes : concevoir l'algorithme, écrire le code, en faire une fonction.

a) Concevoir l'algorithme

- Quel objectif ? *Exemple : calculer la racine carrée d'un nombre*
- Quelles sont les données ? *un réel positif*
- Quel résultat ? *un réel positif*

```
soient x, min, max, milieu, racine réels
lire(x) ← Récupérer la donnée
min ← 0
si x<1 alors max ← 1
sinon max ← x
tant que (max-min)>0.001 ← On réduit l'intervalle jusqu'à
    milieu ← (min+max)/2          encadrer la solution à 10-3
    si milieu*milieu > x alors max ← milieu
    sinon min ← milieu
fin tant que
racine ← (min+max)/2
ecrire(racine) ← Communiquer le résultat
```

b) Écrire le code

```
double x, min, max, milieu, racine ;
scanf("%lf", &x) ; ← Lecture de la donnée
min = 0 ;
if (x<1) max=1 ;
else max=x ;
While((max-min)>0.001)
{ milieu = (min+max)/2 ;
  if (milieu*milieu > x) max = milieu ;
  else min = milieu ;
}
racine = (min+max)/2 ;
printf("%lf", racine) ; ← Communiquer le résultat
```

c) En faire une fonction

```

      Type du résultat
      /
     /
    /
   /
  /
 /
/
double racineCarree(double x)
{ double x, min, max, milieu, racine ;
  scanf("%lf", &x) ;
  min = 0 ;
  if (x<1) max=1 ;
  else max=x ;
  While((max-min)>0.001)
  { milieu = (min+max)/2 ;
    if (milieu*milieu > x) max = milieu ;
    else min = milieu ;
  }
  racine = (min+max)/2 ;
  printf("%lf", racine) ;
  return racine ; ← quitte la fct et transmet le résultat
}

```

Syntaxe : *type fct(liste de paramètres formels) Entête*
 { [définition]
 [instruction] Bloc de la fonction
 }

Syntaxe d'une fonction

Syntaxe pour définir une fonction sans paramètre :

fonction ->

type identificateur (void) *Entête*

instruction-bloc *Bloc de la fonction*

Syntaxe pour définir une fonction avec paramètres :

fonction ->

type identificateur (type paramètreFormel [, type paramètreFormel])

instruction-bloc

Remarques :

- Une fonction doit être déclarée avant son utilisation
- Elle peut être utilisée dans une expression

d) Appeler la fonction

Exemple : calcul de l'hypoténuse d'un triangle rectangle

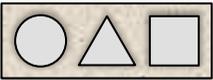
```
void main(void)
{ double larg, haut, H2, H ;
  printf("donnez la largeur :\n");
  scanf("%lf", &larg);
  printf("donnez la hauteur :\n");
  scanf("%lf", &haut);
  H2 = larg*larg + haut*haut ;
  H = racineCarree(H2) ; ← Appel de la fonction
  printf("Hypotenuse = %f\n", H) ;
}
```

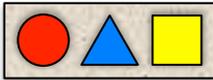
Paramètre effectif

Remarque 1 : on aurait pu écrire directement

```
printf("Hypotenuse = %f\n", racineCarree(H2)) ;
```

Remarque 2 : Les paramètres effectifs (appel) doivent-êtré « compatibles » en nombre et en type avec les paramètres formels (définition de la fct)

Définition : **fct**()

Appel: **fct**()

e) Résultats d'une fonction

L'instruction **return** *résultat* ;

- Permet de quitter immédiatement une fonction en retournant un unique résultat
- Le résultat doit être d'un type compatible avec le type déclaré de la fct
- Ce résultat se substitue à l'appel de la fonction pour la suite des calculs

Remarque : La fonction **scanf** retourne le nbre d'éléments effectivement lus :

```
if (scanf("%d",&i)==0) printf("err: ce n'est pas un nbre\n")
```

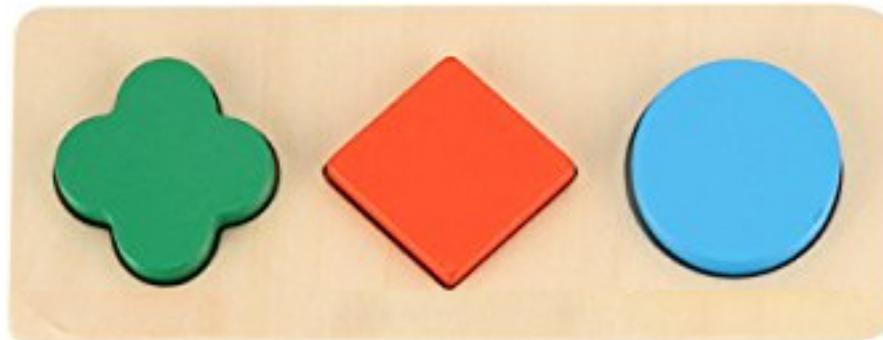
Fonction ou procédure :

- Une procédure est une fonction sans résultat
- Elle est déclarée de type **void** (comme **main** par exemple)
- L'instruction **return;** permet de quitter la procédure à tout moment
- L'instruction **return;** est facultative et l'accolade de fin de procédure est un **return** implicite

A comprendre absolument !!!

Paramètres formels et paramètres effectifs :

- Un **paramètre formel** est une variable locale à la fct qui permet d'écrire le programme de la fct pour une donnée encore inconnue.
- Le **paramètre formel** et le **paramètre effectif** (la donnée) sont deux zones mémoires distinctes
- Les paramètres **paramètres effectifs** (ou paramètres d'appels) sont affectés aux **paramètres formels** à l'appel de la fonction (communication des données)
- Les **paramètres effectifs** doivent être compatibles en nombre et en type avec les **paramètres formels** (les règles de conversion sont implicitement appliquées)



3. Portée d' une variable

Rappel sur *instruction-bloc* :

```
{ [ définition ]  
  [ instruction ]  
}
```

variable globale :

- définie en dehors de toute fonction (de préférence en tête de prog.)
- accessible et partagée par toutes les fonctions

variable locale :

- définie à l'intérieur d'un bloc (en général d' une fonction)
- durée de vie limitée à l' exécution du bloc (de fonction)
- visible uniquement dans ce bloc, inconnue en dehors
- occulte une variable globale de même nom

➔ *Il faut privilégier les variables locales*

```

#include <stdio.h>
int note[10], m ; // globales

void lireNotes(void)
{ int i; // locale
  for (i=0 ; i<10 ; i++)
    scanf("%d", &note[i]);
}

void moyenne(void)
{ int i;
  m = 0;
  for (i=0 ; i<10 ; i++)
    m = m+note[i];
  m = m/10;
}

int max(void)
{ int i, m; /* m occulte m */
  m = 0;
  m = note[0];
  for (i=1 ; i<10 ; i++)
    if (m<note[i]) m=note[i];
  return m ;
}

```

```

void main(void)
{ int nMax;
  lireNotes();
  moyenne();
  nMax = max();
  printf("moyenne=%d ", m);
  printf("max=%d", nMax);
}

```

4. Communication entre fonctions

- variables globales
- instruction **return**
- Paramètres :
 - données
 - **données/résultats**

a) Paramètre de type « donnée »

Objectif : transmettre une valeur à une fct

- **paramètre formel** : variable locale particulière
- qui sera affectée du **paramètre effectif** à l'appel de la fct
(la valeur d'une variable ou le résultat d'une expression)

➔ *règles de conversion pour l'affectation*

Remarque :

- la fct travaille avec une **copie** des paramètres effectifs
- les paramètres effectifs **ne peuvent pas être modifiés** par la fonction

Exemple : x^n avec *décrément* de n

b) Paramètre de type « donnée/résultat »

- On souhaite modifier un paramètre effectif
- exemple : **scanf** modifie une variable par lecture au clavier

Technique :

- **transmettre l'adresse de la variable** à modifier (et non la valeur)
- cette adresse ne pourra pas être modifiée !
- le paramètre formel est un pointeur sur la variable à modifier
- qui permettra d'accéder à sa zone mémoire

Exemple 1 : *permuter(a, b)*

```
void permuter(float *Pa, float *Pb)
// Pa et Pb sont 2 pointeurs sur les variables a et b à permuter
{ float tmp ;
  tmp = *Pa ;   *Pa = *Pb ;   *Pb = tmp ;
}
```

Appel : **float** a=1, b=2;
 permuter(&a, &b);

Exemple 2 : *résolution d'une équation du second degré*

c) Petite excursion en C++ : paramètres données/résultats

Passage par **référence** en C++ :

```
void permuter(int& a, int& b)
{ int tmp ;
  tmp = a ;
  a = b ;
  b = tmp ;
}
```

Appel :

```
int x=1, y=2 ;
Permuter(x, y) ;
```

Résumé

Résultat, paramètre par valeur et paramètre par adresse

```
int maFct(int a, float *Px)
{ int r ; // même type que la fonction
  ...
  *Px = ... // accès à la variable pointée par Px
  ...
  return r ;
  // fin de la fct : a, Px et r sont supprimés
}
```

Appel de la fonction :

```
int i, n;
float x;
...
n = maFct(i+1, &x); // affectation des paramètres a et Px avec i+1 et &x
                    // x pourra être modifié par la fct
```

Les erreurs classiques

- Paramètres d'appel incompatibles avec les paramètres formels (nombre, positions et types)
- Renvoyer un résultat ne respectant pas le type de la fct
- Renvoyer l'adresse d'une variable locale

5. Les tableaux et les fonctions

a) Tableau en paramètre

```
int T1[10];
```

Déclaration d'un tableau en paramètre d'une fct :

```
void initTab(int t[10])  
{corps de la fonction  
}
```

Exemple d'appels de la fct avec un tableau en paramètre **effectif** :

```
initTab(T1); /* on donne l'adresse de T1 */
```

➔ Le contenu d'un tableau est modifiable par une fct car on passe son adresse

b) Tableau de dimension inconnue à la compilation

```
int *T2;  
T2 = malloc(10*sizeof(int)) ;
```

Deux écritures possibles pour déclarer un tableau en paramètre :

```
void initTab(int t[])  
{corps de la fonction }
```

```
void initTab(int *t)  
{corps de la fonction }
```

➔ Il faut aussi transmettre la dimension

```
void initTab(int t[], int nbElements)
```

Rappel sur les matrices : $m[i][j]$ accès à la cellule d'adr $(m + i*Nbcol + j)$

➔ seule la première dimension est facultative :

```
void initMat(int mat[][Nbcol], int nbLig)
```

c) Tableau local à une fonction

- Gros tableau : *allocation dynamique conseillée !*

```
type fct(...)
```

```
{ int *t;
```

```
  t = malloc(...) ;
```

```
  un certain traitement avec t
```

```
  free(t); //ne pas oublier de liberer l'espace
```

```
}
```

Résumé

- Tableau en paramètre

```
void bidouille(double t[], int n) } au choix
void bidouille(double *t , int n) }
{ int i;
  ...
  for (i=0 ; i<n ; i++)
  {
    traiter t[i]
  }
  ...
}
```

Appel de la fonction :

```
double tab[100];
...
bidouille(tab, 100);
```

6. Bibliothèque standard C-ANSI

a) Prototype de fonction

- Il faut **définir** les fonctions avant utilisation
- Correction : il faut **déclarer** les fonctions avant utilisation
 - son nom
 - son type
 - ses paramètres formels

➔ Prototype d'une fonction : *entête ;*

➔ La fonction sera définie plus loin dans le programme

Exemples de prototype : **`void initTab(int t[], int nb);`**
`double sin(double x) ; // voir math.h`

c) Bibliothèques standard

<code>#include <stdio.h></code>	<i>entrée/sortie</i>
<code>#include <math.h></code>	<i>fcts mathématiques (compiler avec l'option <code>-lm</code>)</i>
<code>#include <string.h></code>	<i>traitement des chaînes de caractères</i>
<code>#include <ctype.h></code>	<i>classification des caractères</i>
<code>#include <stdlib.h></code>	<i>utilitaires (malloc,abs,...)</i>
<code>#include <limits.h></code>	<i>limites propres à l'implantation des entiers (INT_MAX, ...)</i>
<code>#include <float.h></code>	<i>limites propres à l'implantation des réels</i>
<code>#include <assert.h></code>	<i>aide à la mise au point</i>

7. Fonctions récursives

- une fonction peut s'appeler elle-même
- une fonction A peut appeler une fonction B qui appelle A

Rappels :

- appel d'une fonction : création des variables locales
 - fin de la fonction : destruction des variables locales
- ➔ les fonctions récursives utilisent une " pile " de variables locales

Ex : factoriel(n)

- $0!$ vaut 1
- $n!$ vaut $n * (n-1) !$

Ex : PGCD(A, B)

- $\text{PGCD}(A, A) = A$
- si $A < B$ alors $\text{PGCD}(A, B) = \text{PGCD}(A, B-A)$
- sinon $\text{PGCD}(A, B) = \text{PGCD}(A-B, B)$

A quoi sert la récursivité ?

- Algorithme parfois plus facile à concevoir
- C'est la seule solution à certains problèmes

Exemple 1 : somme des n premiers entiers

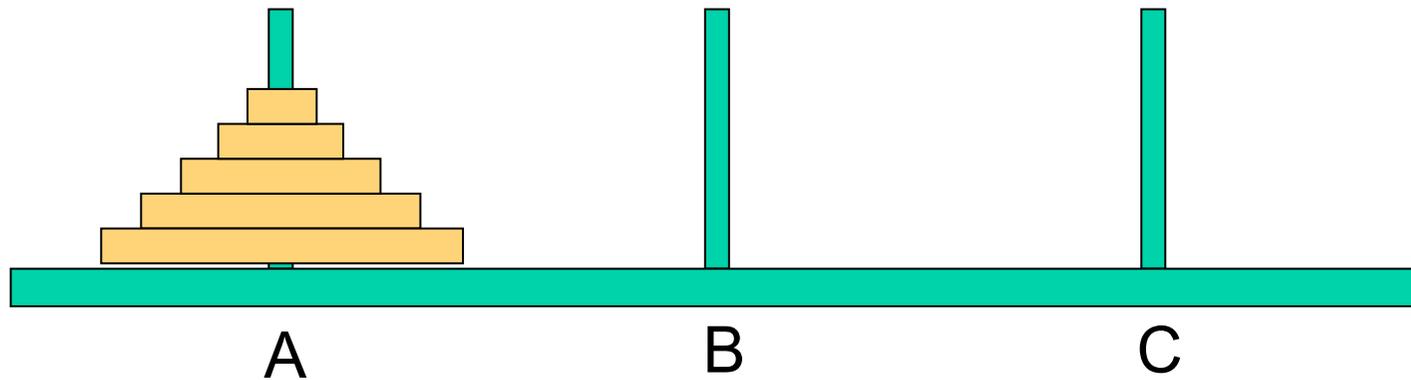
- formule : $\text{somme}(n) = n * (n+1) / 2$
- itératif : $\text{somme}(n) = \sum_{i=0, n} i$
- récursif : $\text{somme}(0) = 0$ et $\text{somme}(n) = n + \text{somme}(n-1)$

Exemple 2 : Il n'y a pas de formule pour calculer n!

Exemples sans formule et sans solution itérative :

- fin de partie d'échec, plus court chemin dans un graphe, ...
- parcours « d'arbre »

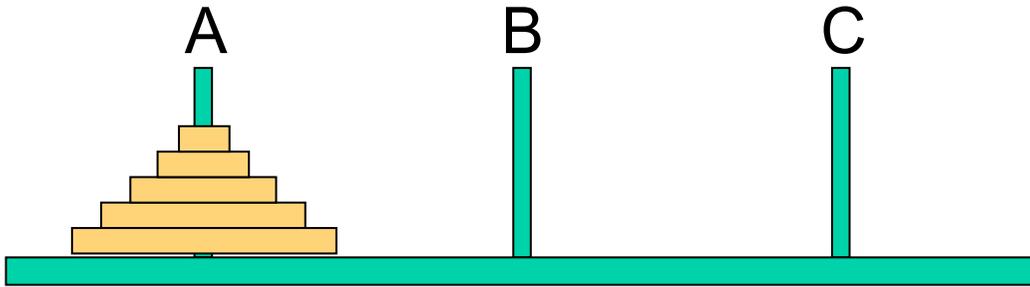
Les tours de Hanoï



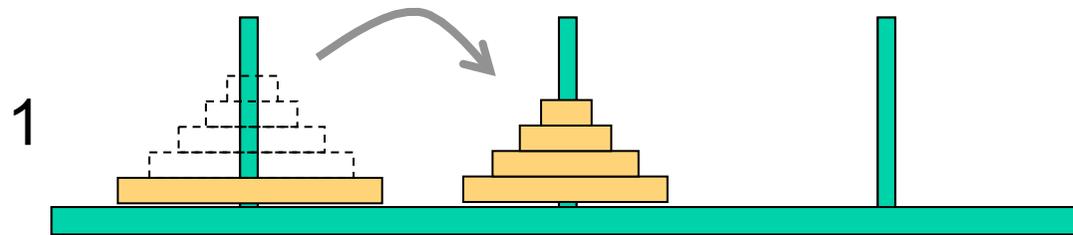
objectif : déplacer la pyramide de A en C

règles :

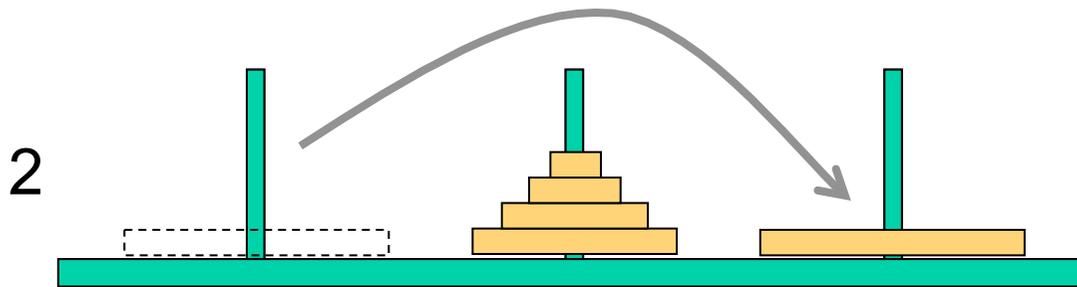
- on ne déplace qu' un disque à la fois
- un disque ne peut être posé que sur un disque plus grand



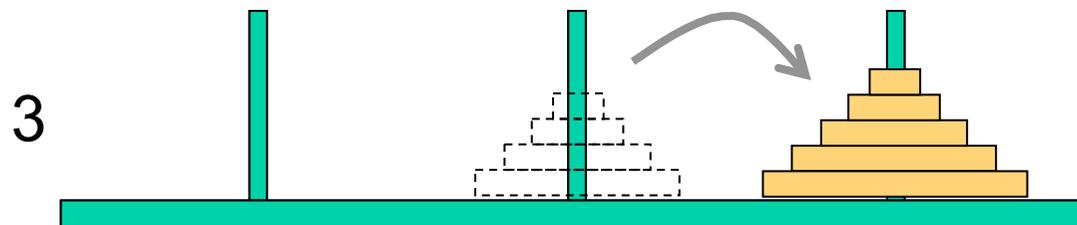
déplacer(n, A, C, B) :



déplacer($n-1, A, B, C$)



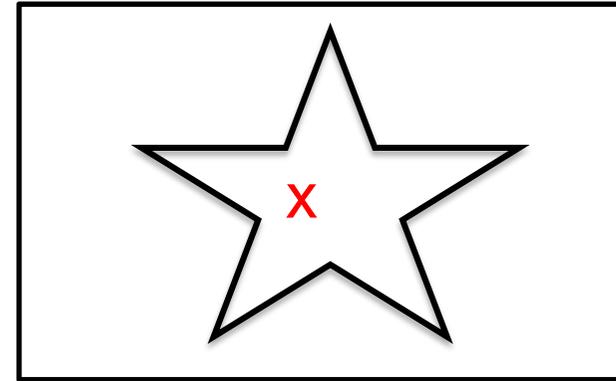
A -> C



déplacer($n-1, B, C, A$)

Coloriage d' une forme quelconque :

- Matrice image I [HAUT][LARGE]
- Forme délimitée par des 0 (couleur noir)
- Pixel d' entrée en (i_0, j_0)



Solution récursive pour colorier en noir :

- si le pixel courant est sur le contour ($=0$) \rightarrow Fin
- si le pixel courant est déjà colorié ($= 0$) \rightarrow Fin
- colorier le pixel (affecter 0)
- relancer le coloriage sur les 4 pixels voisins

