# Algorithmique et Programmation 2

Dr Ouédraogo T. Frédéric

UFR-ST

# Information

UE Informatique 2 (5 crédits) :

- **Algorithmique et Programmation (poids 3)**
- Architecture des ordinateurs (poids 2)

**EC Algorithme et Programmation:**

- **Volume Horaire de 36h dont**
- **Cours : 24h**
- **TD/TP : 12h**

# Contenu

1. **Pointeurs et allocation dynamique de mémoire (en C)**
2. **Types structurés (en C)**
3. **Sous programmes (Procédures et Fonctions)**
4. **Récursivité**
5. **Algorithmes de recherche**

# 1 - Pointeurs et allocation dynamique de mémoire (en C)

# 1 - Pointeurs et allocation dynamique de mémoire

## 1-1 Pointeurs

Un pointeur est un type simple  qui contient l'adresse d'un espace de la mémoire.

le pointeur peut pointer sur une variable déclarée.

# 1-1 Pointeurs

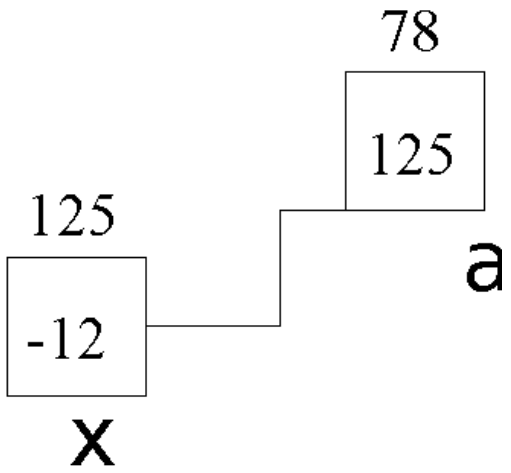
Les pointeurs servent à:

- Réaliser les structures dynamiques (liste chainées)
- Passage des paramètres par référence (fonctions)
- Gestion efficace de structure complexe (arbre, graphe)

# 1.1 Pointeurs

Exemple: Pointeur sur variable.

1) int x;
2) int * a;




3) x=-12
4) a = &x
5) *a = 7

*a signifie la valeur qui se trouve à l'adresse pointée par a

&x signifie adresse de la variable x

# 1.1 Pointeurs

## Déclaration

La déclaration d'un pointeur est comme suit :

**<Type> * <NomPointeur>**

déclare un pointeur **<NomPointeur>** qui peut recevoir des adresses de variables du type **<Type>**

Une déclaration comme

**int *p;**

peut être interprétée comme suit:

"*p est du type int" ou " p est un pointeur sur int" ou

" p peut contenir l'adresse d'une variable du type int"

# 1.1 Pointeurs

NULL est la valeur d'un pointeur qui pointe sur aucune adresse.

Int *a ;

a = NULL;

Une variable pointeur non initialisé contient par défaut la valeur NULL.

# 1-1 Pointeurs

## Opérateurs de base

Lorsqu'on utilise des pointeurs en C, nous avons besoin:

- d'un opérateur « adresse de » **&** pour obtenir l'adresse d'une variable.

**&<NomVariable>** fournit l'adresse de la variable **<NomVariable>**

**Exemple:**

- Soit **P** un pointeur et **X** une variable (du même type) contenant la valeur **20** . Alors l'instruction **P = &X;** affecte l'adresse de la variable **X** à la variable **P**.

# 1-1 Pointeurs

## Opérateurs de base

- d'un opérateur « contenu de » * pour accéder au contenu d'une adresse.
- ***<NomPointeur>** désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

# 1-1 Pointeurs

## Exemple

Soit **X** une variable contenant la valeur **20**, **Y** une variable contenant la valeur **40** et **P** un pointeur

Après les instructions,

1. `P = &X;`
2. `Y = *P;`
3. `*P = 100;`

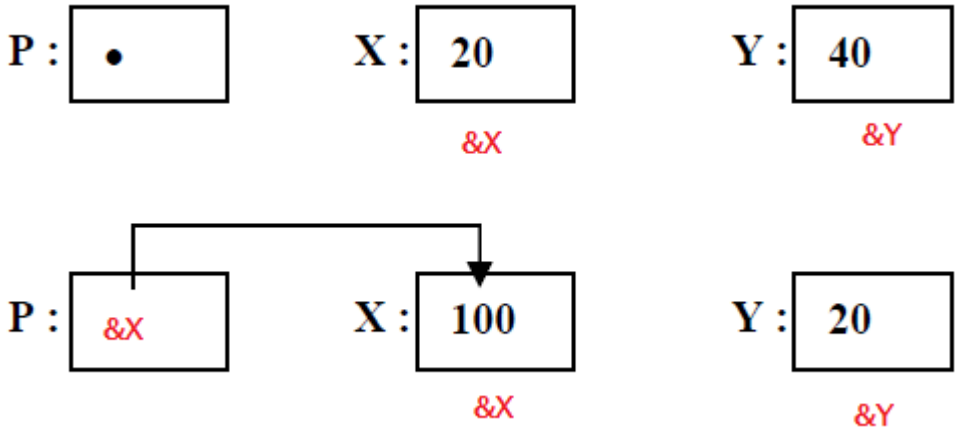
P pointe sur X,

le contenu de X (référencé par *P) est affecté à Y, et

le contenu de X (référencé par *P) est mis à 100.

# 1-1 Pointeurs

## Exemple



# 1-1 Pointeurs

## Priorité des opérateurs

Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrémentation ++, la décrémentation --).

Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

# 1.2 Adressage des composantes d'un tableau

## Tableau à 1 dimension

Le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes:

**&tableau[0]  et  tableau** sont une seule et même adresse.

## Exemple

En déclarant un tableau **T** de type *int* et un pointeur **P** sur *int*,

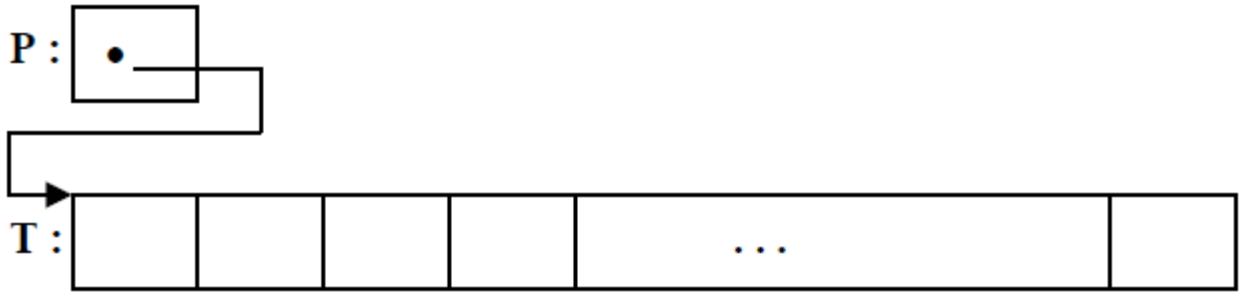
- int T[20];
- Int *P;

# 1.2 Adressage des composantes d'un tableau

## Tableau à 1 dimension

L'instruction:

P = T;  est équivalente à P = &T[0];

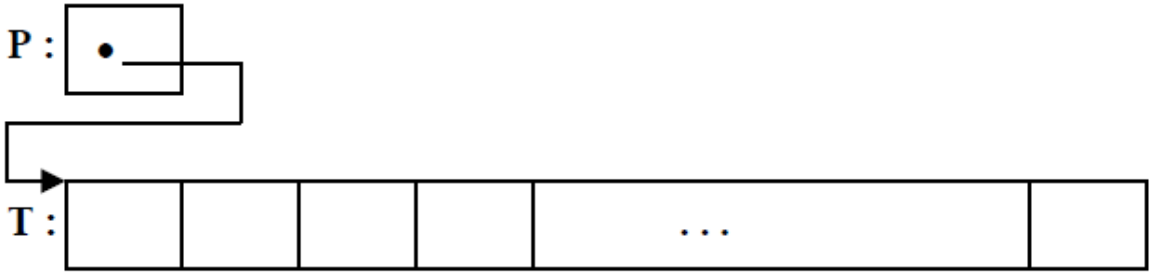


# 1.2 Adressage des composantes d'un tableau

## Tableau à 1 dimension

l'instruction:

P = T;  est équivalente à P = &T[0];




- Si **P** pointe sur une composante quelconque d'un tableau, alors **P+1** pointe sur la composante **suivante**. Plus généralement,

**P+i** pointe sur la **i-ième** composante **derrière P** et

**P-i** pointe sur la **i-ième** composante **devant P**.

# 1.2 Adressage des composantes d'un tableau

## Tableau à 1 dimension

Ainsi, après l'instruction, P = T; le pointeur P pointe sur T[0], et

*(P+1) désigne le contenu de T[1]

*(P+2) désigne le contenu de T[2]

...

*(P+i) désigne le contenu de T[i]

# 1.2 Adressage des composantes d'un tableau

## Tableau à 2 dimensions

Un tableau à 2 dimensions est un tableau dont les éléments sont eux-mêmes des tableaux.

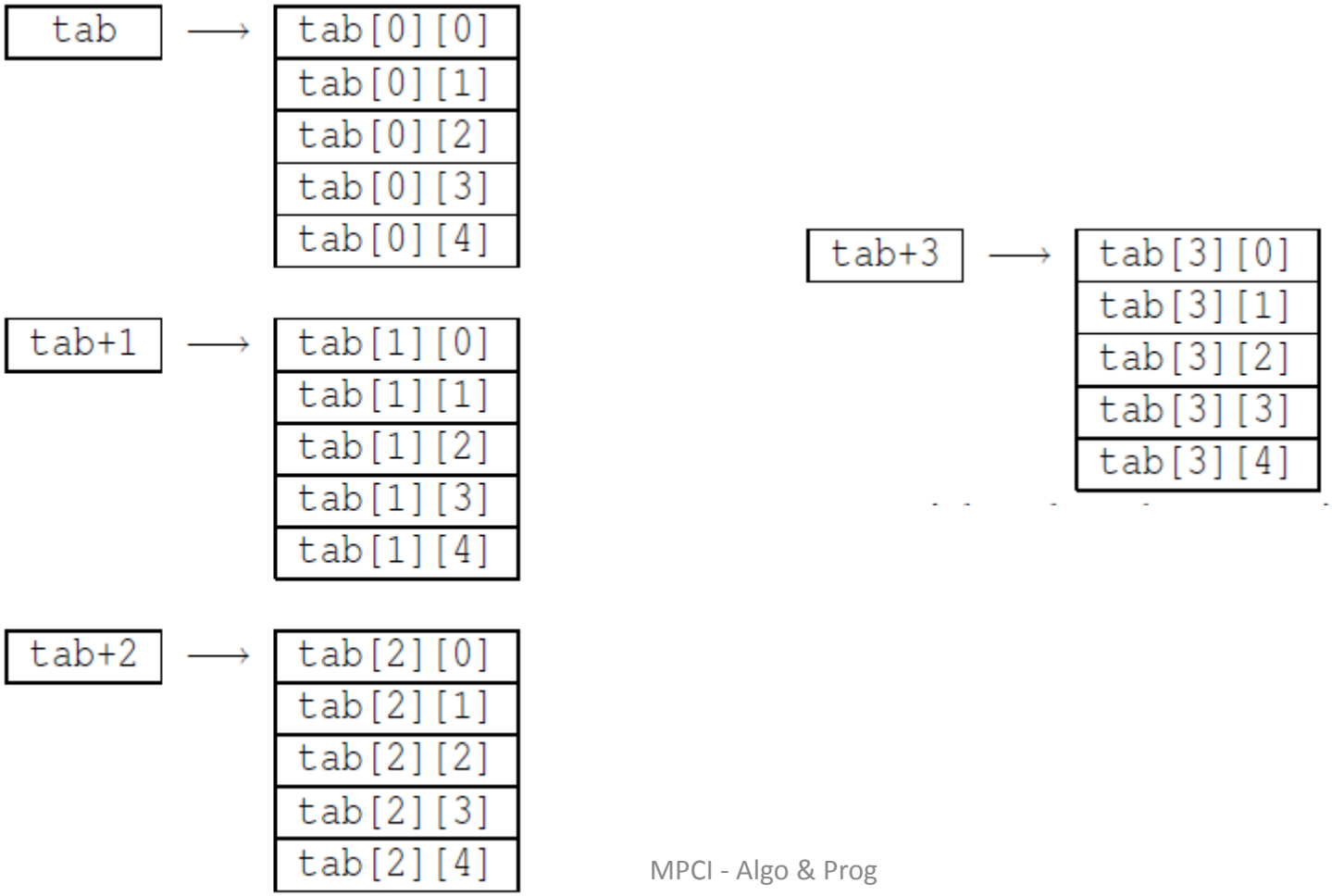
Ainsi, le tableau défini par $\mathrm{int}\ \mathrm{tab}[4][5]$ ; contient 4 tableaux de 5 entiers chacun.

$\mathrm{tab}$ donne l'adresse du 1*er* sous-tableau,

tab+1 celle du 2*ème* sous-tableau et ainsi de suite :

# 1.2 Adressage des composantes d'un tableau

## Tableau à 2 dimensions



# Exercice d’application

# 1.3 Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur *int* peut contenir l'adresse d'un nombre isolé ou d'un élément d'un tableau,

un pointeur sur char peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères.

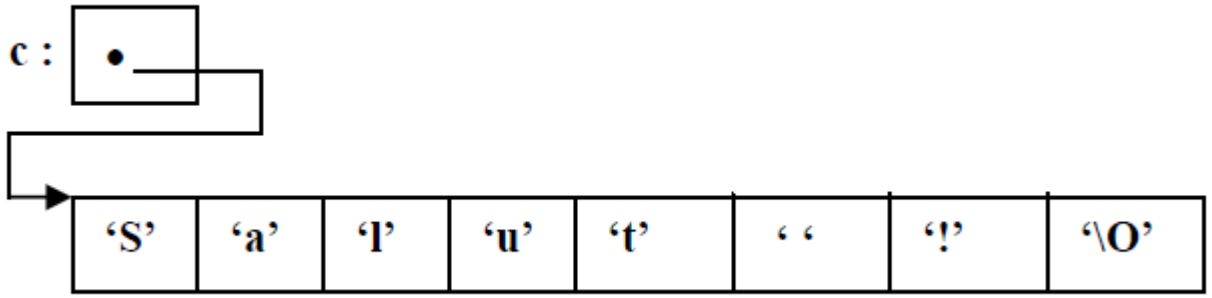
Un chaîne de caractères est représentée en langage C par un tableau de caractères.

# 1.3 Pointeurs et chaînes de caractères

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur char :

Exemple

- *char *c;*
- *c = "Salut !";*



# 1.3 Pointeurs et chaînes de caractères

Un pointeur sur char peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

*char *C = "Salut !";*

***Attention !** : Il existe une différence importante entre les deux déclarations:*

***char T[] = "Salut !";** /* un tableau */*

***char *C = "Salut !";** /* un pointeur */*

# 1.3 Pointeurs et chaînes de caractères

- **T *est un tableau*** qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '**\0**'. Les caractères de la chaîne peuvent être changés, mais le nom **T** va toujours pointer sur la même adresse en mémoire.
- **C *est un pointeur*** qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.

# Exercices

- Ecrire un programme qui lit une chaîne de caractères CH et détermine la longueur de la chaîne à l'aide d'un pointeur P.

- Ecrire un programme qui vérifie sans utiliser une fonction de <*string*>, si une chaîne CH introduite au clavier est un palindrome:

Rappel: Un palindrome est un mot qui reste le même qu'on le lise de gauche à droite ou de droite à gauche:

Exemples:

PIERRE  n'est pas un palindrome

OTTO est un palindrome

# 1.4 Allocation dynamique de mémoire

L'allocation dynamique de mémoire permet en **cours d'exécution** d'un programme, de **réserver un espace mémoire dont la taille n'est pas nécessairement connue lors de la compilation**.

Ceci est à mettre en opposition avec l'allocation statique de mémoire. (??)

L'allocation dynamique est réalisée par les fonctions **malloc**() et **calloc ()**.

# 1.4 Allocation dynamique de mémoire

## Fonction malloc()

- La fonction malloc( N ) renvoie l'adresse d'un bloc de mémoires de N octets libres ou NULL s'il n'y a pas assez de mémoire libre.
- Pour l'utiliser il faut inclure la bibliothèque <**stdlib.h**> en début de programme.

# 1.4 Allocation dynamique de mémoire

Exemple

```
int *a, n;
1. n=14;
2. a=(int*) malloc(sizeof(int));
3. *a=25;
```

La fonction malloc alloue (si possible) un bloc mémoire pour 1 int et retourne l'adresse (non typée, void *) du bloc. On effectue alors une conversion de type : void * vers int *.

# 1.4 Allocation dynamique de mémoire

## Prototype

void* **malloc**( size_t  t) ;

- permet de réserver un bloc de taille t (en octets) et renvoie un pointeur vers l'adresse du bloc alloué s'il y a suffisamment de mémoire disponible;
- renvoie la valeur NULL en cas d'erreur.

# 1.4 Allocation dynamique de mémoire

## L'operateur sizeof

D'une machine à une autre, la taille réservée pour un int, un float, peut varier. Si nous voulons réserver de la mémoire pour des données d'un type sur machine, l'operateur sizeof nous fournit ce renseignement.

- **sizeof** (type) fournit la taille de « type » sur la machine

Exemple

- printf("char : %d octets\n", sizeof(char));
- printf("int : %d octets\n", sizeof(int));

# 1.4 Allocation dynamique de mémoire

## Fonction free();

- void **free**(void* p)

free permet de libérer de la mémoire préalablement allouée par les fonctions malloc(). En paramètre, on passe l'adresse du bloc mémoire à libérer.

# 1.4 Allocation dynamique de mémoire

```
#include<stdio.h>
#include<stdlib.h>
# define PI  3,14
int main() {
 int* memoireAllouee ;
 memoireAllouee =(int*) malloc(sizeof(int)); // Allocation de la mémoire

if (memoireAllouee == NULL) {
            exit(0);   // Erreur : on arrête tout !
 }
// Utilisation de la mémoire
printf("Quel age avez-vous ? ");
scanf("%d", memoireAllouee);
printf("Vous avez %d ans\n", *memoireAllouee);
free(memoireAllouee);  // Libération de mémoire
return 0;
}
```

# 1.5 Tableaux dynamiques

Un des principaux intérêts de l'**allocation dynamique** est de permettre **à** un programme de réserver la place nécessaire au stockage d'un tableau en **mémoire** dont il ne connaissait pas la taille avant.

En effet, jusqu'ici, la taille de nos tableaux était fixée dans le code source. Nous allons voir comment créer des tableaux dont la taille peut varier à l'exécution.

# 1.5 Tableaux dynamiques

Lors de la déclaration d'un tableau en langage C, il fallait préciser les dimensions, soit de:

- façon explicite :

```
int tab[5] ;
float tam[3][2];
```

- façon implicite (par initialisation) :

```
int tab[ ]  = {4, 6, 2, 8, 1};
float tam[ ][ ] = { {0 , 1 } , {2 , 3 } , {4 , 5 } };
```

# 1.5 Tableaux dynamiques

Dans les 2 cas, on a déclaré des tableaux de tailles fixes.

Mais si l'on veut que le tableau change de taille d'une exécution à une autre, cela nous oblige à modifier le programme et à le recompiler à chaque fois, ou bien à déclarer un tableau de taille nettement  plus grand mais ce serait du gâchis.

# 1.5 Tableaux dynamiques

Pour éviter cela, on fait appel à l'**allocation dynamique de mémoire** : au lieu de réserver de la place lors de la compilation, on la réserve pendant l'exécution du programme, de façon interactive.

# 1.5 Tableaux dynamiques

## Allocation dynamique pour un tableau à 1 dimension

On veut réserver de la place pour un tableau de n éléments(entiers), où n est lu au clavier.

Exemple:

```
int n ; //variable pour la taille du tableau
int *tab ;
printf("taille du tableau :\n") ;
scanf("%d", &n) ;
tab = (int *)malloc(n*sizeof(int)) ;
```

# 1.5 Tableaux dynamiques

## Allocation dynamique pour un tableau à 1 dimension

Les (int *) devant le malloc s'appelle un **cast**. Un cast est un opérateur qui convertit ce qui suit(ici void*) selon le type précisé entre parenthèses.

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
    int nombreDAmis = 0,  i = 0 ;
    int* ageAmis = NULL; // Ce pointeur va servir de tableau après l'appel du malloc

    // On demande le nombre d'amis à l'utilisateur
    printf("Combien d'amis avez-vous ? ");
    scanf("%d", &nombreDAmis);

    if (nombreDAmis > 0) {  // Il faut qu'il ait au moins un ami
        ageAmis =  (int *)malloc(nombreDAmis * sizeof(int));  // On alloue de la mémoire pour le tableau
        if (ageAmis == NULL) {   // On vérifie si l'allocation a marché ou non
            exit(0); // et on quitte le programme si l'allocation a échoué
        }
```

```
 // On demande l'âge des amis un à un
        for (i = 0 ; i < nombreDAmis ; i++){
            printf("Quel age a l'ami numero %d ? ", i + 1);
            scanf("%d", &ageAmis[i]);  // on pouvait aussi
ecrire  scanf("%d", (ageAmis+i));
        }
        // On affiche les âges stockés un à un
        printf("\n\nVos amis ont les ages suivants :\n");
        for (i = 0 ; i < nombreDAmis ; i++) {
            printf("%d ans\n", ageAmis[i]);
        }
        // A la fin, on libère la mémoire allouée avec
malloc, on n'en a plus besoin
        free(ageAmis);
    }
    return 0;
}
```

# 1.5 Tableaux dynamiques

Exercices d'application

1. Modifier pour afficher la moyenne d'âge de vos amis, le plus vieux, le plus jeune et ceux qui ont l'âge supérieur à la moyenne.
2. Ecrire un programme qui affiche dans l'ordre inverse des entiers entrés au clavier par l'utilisateur. Le nombre d'entiers est précisé par l 'utilisateur.

3. Ecrire un programme qui lit un entier X et un tableau A du type int au clavier et élimine toutes les occurrences de X dans A en tassant les éléments restants. Le programme utilisera les pointeurs P1 et P2 pour parcourir le tableau.

# 1.5 Tableaux dynamiques

## Allocation dynamique pour un tableau à 2 dimensions

On veut réserver de la place pour un tableau de n fois m entiers, où les dimensions n et m sont lus au clavier :

# Allocation dynamique pour un tableau à 2 dimensions

```
int i,j,n,m ;
float **tab ;  /* (1) */
scanf("%d %d", n, m) ;
tab = (float **) malloc(n*sizeof(float *)) ; /* (2) */
for (i=0 ; i<n ; i++){
        tab[i] = (float *)malloc(m*sizeof(float)) ; /* (3) */
}
for (i=0 ; i<n ; i++){
     for (j=0 ; j<m ; j++){
         tab[i][j] = 10*i+j ; /* (4) */
      }

}
// libérer tab à la fin
```

# Allocation dynamique pour un tableau à 2 dimensions

1. Un tableau de pointeurs étant un pointeur de pointeurs, on déclare tab donc un pointeur de pointeurs **tab.
2. tab étant en fait un tableau de pointeurs de réels(flottants), on réserve un bloc pouvant contenir n pointeurs de réels (float). tab contient alors l'adresse de ce bloc.

# Allocation dynamique pour un tableau à 2 dimensions

/* (1) */ Un tableau de pointeurs étant un pointeur de pointeurs, on déclare tab donc un pointeur de pointeurs **tab.

/* (2) */ tab étant en fait un tableau de pointeurs de réels(flottants), on réserve un bloc pouvant contenir n pointeurs de réels (float). tab contient alors l'adresse de ce bloc.

/* (3) */ Les tab[i] sont des sous-tableaux de tab. On réserve alors pour chacun d'eux de la place pour m flottants. Au total, on a bien réservé de la place pour n fois m flottants.

# Allocation dynamique pour un tableau à 2 dimensions

/* (4) */ On manipule les éléments de tab comme ceux d'un tableau "normal".

# 2 - Types structurés

Langage C

# 2 - Types structurés (en C)

Le type tableau est une structure composée de plusieurs éléments **de même type.**

Mais si nous voulons regrouper les informations de différents types, un autre structure composée appelée **type structuré(ou enregistrement)** est nécessaire.

# 2 - Types structurés (en C)

## Déclaration

La déclaration d'une structure dont l'identificateur est *nomStructure* suit la syntaxe suivante :

```
struct nomStructure {
    type1 nomChamp1;
    type2 nomChamp2;
    ...
    typen nomChampn;
};
```

# 2 - Types structurés (en C)

## Exemple introduction

Les structures permettent de rassembler des valeurs de types différents.

Par exemple, pour une adresse, on a besoin du numéro de porte (int), du nom de la rue (char) et du numéro du secteur.

# 2 - Types structurés (en C)

## Exemple introductif

On déclare la structure adresse ainsi :

```
struct adresse {
   int numero ;
   char rue[50] ;
   int secteur ;
} ;
```

- Chaque élément déclaré à l'intérieur de la structure est appelé un **champ**.
- on a  fait que déclarer un type de structure, pas une variable. On déclare les variables associées à une structure de cette manière :

```
struct adresse  chezFati, chezJean ;
```

# 2 - Types structurés (en C)

## Accès aux champs

- On accède aux données contenues dans les champs d'une structure et faisant suivre le nom de la structure par un point "." et le nom du champ voulu :

```
chezFati.numero=19 ;
printf("%s",chezFati.rue);
```

- On peut initialiser une structure lors de sa déclaration :

```
struct adresse chezFati= { 15 , "rue  M. Yaméogo",  8 } ;
```

# 2 - Types structurés (en C)

## Affectation

Si deux variables sont de même structure alors on peut effectuer une affectation:

```
chezJean = chezFati ;
/* Jean a emménagé chez Fati ! */
```

Attention : on ne peut pas comparer 2 variables de même structure (avec == ou !=).

# 2 - Types structurés (en C)

## Structure de structure

On peut utiliser une structure comme champ d'une autre structure.

Dans la lignée des exemples précédents, on peut définir une **structure adresse** qui pourrait être utilisée dans la **structure repertoire** ou toute autre structure.

# Structure de structure

```
struct adresse {
        int numero ;
        char rue[50] ;
        int secteur;
} ;
struct repertoire {
        char nom[20] ;
        char prenom[20] ;
        struct adresse  maison ;
};
```

Déclaration d'un champ structure de type 'adresse' appelé 'maison'.

# 2 - Types structurés (en C)

## Tableau de structure

On déclare et utilise un tableau de structure en C de la même façon qu'un tableau de variables simples. Le nombre d'éléments est précisé entre crochets.

```
struct adresse pers[100] ;
```

Cette déclaration nécessite que la structure adresse ait déjà été déclarée avant.

pers est alors un tableau dont chaque élément est de structure de type adresse.

pers[i].rue fait référence au champ "rue" du *i ème* élément du tableau pers.

# Exercices

1. Déclarer un type qui permettent de stocker un étudiant caractérisé par son nom, prénom, sa date de naissance, son matricule et son sexe, puis une liste de 50 étudiants
2. Ecrire une fonction en C LireEtudiant() qui permet de lire au clavier les informations d'un étudiant.
3. De même écrire une fonction afficherEtudiant() qui permet d'afficher les informations sur chaque étudiant de la liste.

# 3 : Les sous-programmes

# Introduction

- les algorithmes selon une approche simple :
  - Un seul algorithme exprime une fonction
  - qui à partir de données initiales produit un résultat

- Cas plus général :
  - Le calcul d'une fonction est le résultat de plusieurs algorithmes disjoints
  - À chaque algorithme va donc correspondre un sous - programme

# Exemple simple d'appels de sous-programmes

```
#include <stdio.h>

    main(){
        float X, Y;
        printf("Entrez la valeur  : ");
        scanf(‘’%f’’,&X);
        Y = cos(X);
        printf(" %f’’ ,Y);
    }
```

*appel du sous-programme d'affichage*

*appel du sous-programme de lecture*

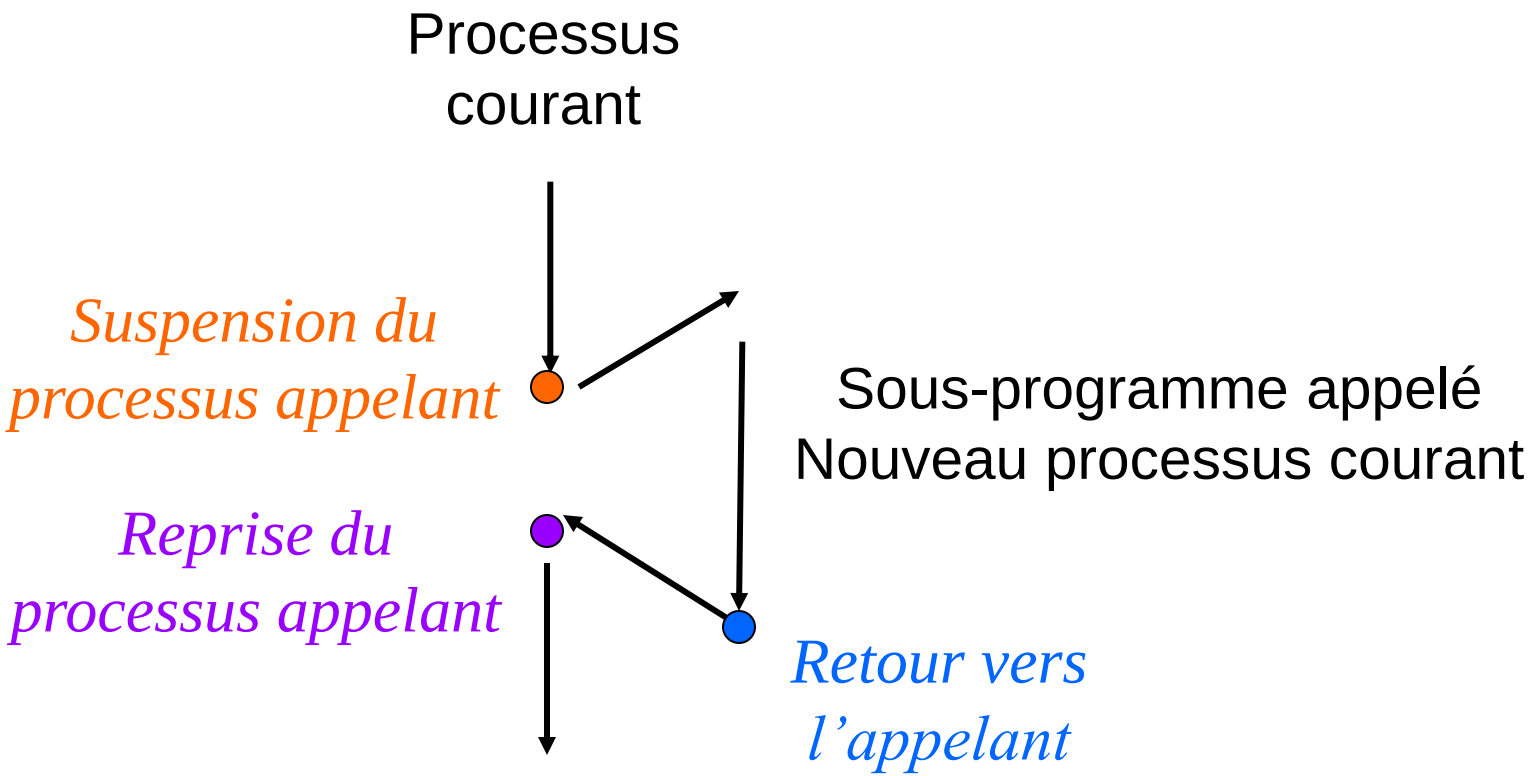
*appel du sous-programme cos*

*appel du sous-programme d'affichage*

# 3.1 Fonctionnement

- le programme principal (main)  lance successivement un séquence de sous-programmes, chacun d'eux pouvant d'ailleurs lancer ses propres sous-programmes.
- Cela revient à suspendre l'exécution du processus du programme principal (main) et à lancer le processus associé au sous-programme.

# 3.1 Fonctionnement



# 3.2 Intérêts

- Diviser pour mieux régner
- Méthode modulaire de conception utilisant le découpage d'un problème en sous-problèmes distincts
- Permet de réutiliser des programmes (sous-programmes) déjà développés et surtout validés

# Problématique de l'usage des sous-programmes

- Quel est le rôle du sous-programme qui est exécuté en premier ?
- Peut-on définir des variables communes à plusieurs sous-programmes et comment sont-elles définies et/ou modifiées ?
- Comment peut-on transmettre des informations spécifiques à un sous-programme lors de son lancement ?
- Comment un sous-programme peut renvoyer des informations au programme appelant ?

# 3.3 Environnement d'un sous-programme

- L'environnement d'un sous-programme est l'ensemble des variables accessibles et des valeurs disponibles dans ce sous-programme, en particulier celles issues du programme appelant.
- Trois sortes de variables sont accessibles dans le sous-programme :
  - Les variables dites globales
  - Les variables dites locales
  - Les paramètres formels

# 3.3 Environnement d'un sous-programme

## Les variables dites globales :

- Elles sont définies dans le programme appelant et considérées comme disponibles dans le sous-programme
- Elles peuvent donc être référencées partout dans le programme appelant et dans le sous-programme appelé
- Leur portée est globale

# 3.3 Environnement d'un sous-programme

## Les variables dites locales

- Celles définie dans le corps du sous-programme, utiles pour son exécution propre et accessibles seulement dans ce sous-programme
- Elles sont donc invisibles pour le programme appelant
- Leur portée est locale

# 3.3 Environnement d'un sous-programme

## Les paramètres formels

- Les variables identifiées dans le sous-programme qui servent à l'échange d'information entre les programmes appelant et appelé
- Leur portée est également locale.

# Paramètres formels & paramètres effectifs

Dans la déclaration de la procédure « Saisir », Les paramètres « X » et « dim » sont appelés paramètres ***formels*** dans la mesure où ils ne possèdent pas encore de valeurs.

Lors de l'utilisation de la procédure dans l'algorithme principal, « A », « B » et « n » sont des paramètres ***effectifs*** (ou réels).

**Remarque importante:**

- Dans une procédure, le nombre de paramètres formels est exactement égal au nombre paramètres effectifs.
- De même à chaque paramètre formel doit correspondre un paramètre effectif de même type.

```
Procédure carre (Données x,y :entiers)
  Entier: a, b
  Début
    a ← x
    b ←  y
    x ←  a*a
    y ←  b*b
  Fin
entiers : m1,m2 ;
DÉBUT
  lire m1, m2
  carre(m1,m2)
  écrire  ”m1= “,m1,  ”m2= “,m2
FIN
```

Paramètres formels

Variables locales du sous-programme

Variables locales Du programme

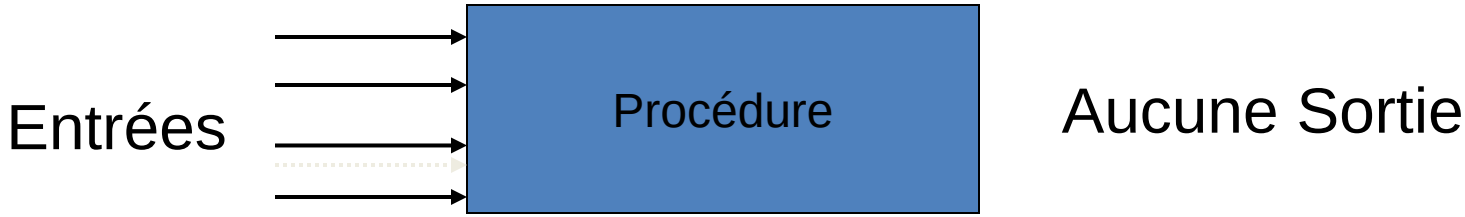
Paramètres effectifs

# 3.4 Les procédures

Une procédure est un sous-programme ***nommé*** qui représente une ou plusieurs actions, et qui ne retourne aucune valeur, **sauf en paramètre.**

Une procédure « P » doit être définie une seule fois dans un algorithme « A » et peut être appelée plusieurs fois par « A » ou par une autre procédure de « A ».

# 3.4 Les procédures



# 3.4 Les procédures

La définition d'une procédure comprend trois parties:

- Partie 1 : L'entête

  **Procédure** *Nom_procédure* **(***Liste de paramètres formels***);**

- Partie 2 : Déclaration des variables locales

  *type_variable_i  Nom_variable_i*

- Partie 3 : Corps de la procédure

  **Début**

  *Instructions*

  **Fin**

# 3.4 Les procédures

**Exemple :**

Calcul de la somme de deux matrices carrées A et B.

- Pour calculer la somme de deux matrices il faut lire d'abord ces deux matrices.
- Ainsi, au lieu d'écrire deux fois le sous-programme de lecture d'une matrice, il est possible d'écrire plutôt une procédure et de l'utiliser pour saisir A et B.

# 3.4.1 Passage des paramètres

Lors de la définition d'une procédure il est possible de choisir entre trois modes de transmission de paramètres:

- **Mode donnée** *(En C on parle de Passage par valeur)*
- **Mode Résultat** *(En C on parle de Passage par adresse*).
- **Mode Donnée/Résultat** (*En C on parle de Passage par adresse*):

# Mode données

Lors de l'appel d'une procédure, un emplacement mémoire est réservé pour chaque **paramètre formel.**

De même, un emplacement mémoire est aussi réservé pour chaque **paramètre effectif** lors de sa déclaration.

Cependant, lors de l'appel de la procédure, les valeurs des paramètres effectifs sont copiées dans les paramètres formels.

# Mode donnée

L'exécution des instructions de la procédure se fait avec les valeurs des paramètres formels et toute modification de ces derniers ne peut affecter en aucun cas celles des paramètres effectifs.

Dans ce type de passage de paramètres, les valeurs des paramètres effectifs sont connues avant le début de l'exécution de la procédure et jouent le rôle uniquement d'entrées de la procédure.

# Mode données

En C, il n'existe que des fonctions. Une fonction qui ne retourne aucun résultat (type void) peut traduire la notion de procédure en algorithmique.

**Procédure** Afficher(**Données** Y : Vecteur, **Données** dim: entier)

Se traduit en C par :

```
void  Afficher(Vecteur Y, int dim);
```

# Mode données

**procédure** carre(**données** x,y :**entiers**)

**entiers** a,b

**début**

a ← x

b ← y

x ← a∗a

y ← b∗b

**fin**

```
entiers m1,m2
Début
  lire m1, m2
  carre(m1,m2)
  écrire " m1= ",m1, " m2= ",m2
Fin
```

Traduire en Langage C

```
Void carre(int x, int y){
Int a,b;
    a = x ;
    b = y ;
    x =  a*a ;
    y =  b*b ;
}
```

```
Int main(){
Int m1,m2;
Scanf("%d  %d", &m1,&m2);
carre(m1,m2);
 printf(" m1= %d m2= %d",m1, m2);
Return 0;
}
```

# Mode résultat

- Dans ce mode de passage de paramètres, les valeurs des paramètres effectifs sont inconnues au début de l'exécution de la procédure.
- Un paramètre formel utilisant ce type de passage ne peut être que le résultat de la procédure. D'où le nom du mode « Résultat ».
- Pour spécifier dans une procédure qu'il s'agit du mode « Résultat », il suffit d'ajouter dans l'algorithme la mention « **Résultat** » avant le paramètre formel

# Mode résultats

**Procédure** Somme(**Données** X,Y:entier, **Résultat** Z:entier)

Se traduit en C par :

```
void  Somme(int X, int Y,   int *Z);
```

Contrairement au passage par valeur, dans ce cas à l'appel de la procédure les valeurs des paramètres effectifs ne sont pas copiées dans les paramètres formels.  Ces derniers utilisent directement l'emplacement mémoire (ou l'adresse) des paramètres effectifs.

# Mode résultats

La principale différence entre le passage par valeur et le passage par adresse  est que dans ce dernier **un seul emplacement mémoire** est réservé pour le paramètre formel et le paramètre effectif correspondant.

- Dans ce cas chaque paramètre formel de la procédure utilise **directement** l'emplacement mémoire du paramètre effectif correspondant.
- Toute modification du paramètre formel entraîne la même modification du paramètre effectif correspondant.

# Mode Données/Résultats

- Les paramètres formels jouent simultanément le rôle de donnée (i.e. d'entrée) et de résultat (sortie) de la procédure: d'où l'appellation « Donnée/Résultat ».
- A l'appel de la procédure, les valeurs des paramètres effectifs sont connues au début de l'exécution de la procédure mais elles seront modifiées à la fin de cette exécution: d'ou la seconde appellation « Donnée Modifiée ».

# Mode Données/Résultats

- Ces deux derniers modes (i.e. *Résultat* et *Donnée Modifiée*) se distinguent uniquement au niveau de l'algorithme.
- En C ces deux derniers modes sont exactement identiques : il s'agit dans les deux cas de *passage par adresse*.

# Mode Données/Résultats

- Pour spécifier dans une procédure qu'il s'agit du mode Donnée/Résultat (ou donnée modifiée), il suffit d'ajouter dans l'algorithme la mention
  - « **Donnée/Résultat** » ou « **Donnée Modifiée** »

  devant le paramètre formel
- Dans le langage C le paramètre formel doit être un pointeur

# Mode Données/Résultats

- Exemple :

  Algorithme d'une procédure permettant de remplacer par 0 toutes les valeurs inférieures à 10 d'un vecteur.

  Le vecteur qu'il faut fournir à cette procédure doit être une donnée ( i.e. qui contient déjà des valeurs).

  Le résultat de cette procédure est ce même vecteur modifié. Ainsi, dans ce cas le mode que l'on doit utiliser est « **Donnée/Résultat** » (ou **Donnée modifiée**).

```
Procédure Remplacer (Donnée Modifiée V:vecteur; Donnée
    dim: entier)
  i entier
Début
  Pour i←1 à dim faire
     Si V[i]<10 alors
        V[i] ← 0
     Fsi
    Fpour
Fin;
```

Exo : Traduire en Langage C

# 3.5 Les fonctions

Une fonction est un sous-programme qui calcule, à partir de paramètres éventuels, une valeur d'un certain type utilisable dans une expression du programme appelant.

Les langages de programmation proposent de nombreuses fonctions prédéfinies : partie entière d'un nombre, racine carrée, fonctions trigonométriques, etc.

# 3.5 Les fonctions

- Les fonctions sont comparables aux procédures à deux exceptions près :
  - Leur appel ne constitue pas à lui seul une instruction, mais figure dans une expression ;
  - Leur exécution produit un résultat qui prend la place de la fonction lors de l'évaluation de l'expression.

# 3.5 Les fonctions

La définition d'une fonction comprend aussi trois parties:

- Partie 1 : L'entête

**Fonction** *Nom_fonction*(*Liste de paramètres formels*):
Type_retourné_fonction

- Partie 2 : Déclaration des variables locales

*type_variable_i  Nom_variable_i*

- Partie 3 : Corps de la fonction

**Début**

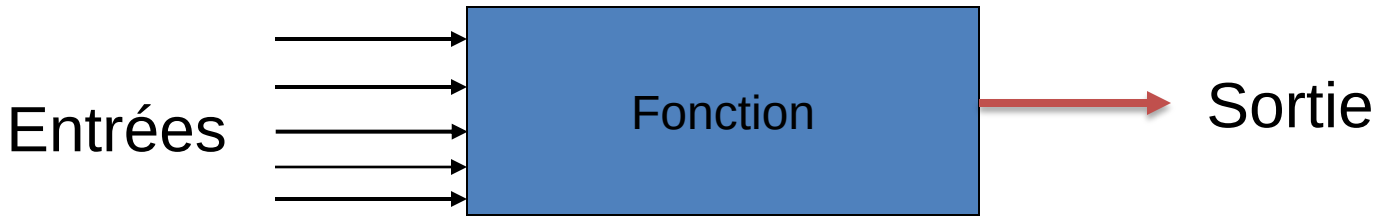
*Instructions*

**Retour** ***valeur retournée***

**Fin**

# 3.5 Les fonctions

- Il  s'agit donc de sous-programmes *nommés* et *typés* qui calculent et retournent une et une seule valeur.
- Le type de cette valeur est celui de la fonction.



# Passage des paramètres

Lors de la définition d'une fonction en LDA il ne doit, normalement, être utilisé qu'un seul mode de transmission de paramètres :

- **Mode donnée** *(En C on parle de Passage par valeur)*

**Cependant les langages de programmation permettent d'utiliser les autres modes.**

# Exemple

Fonction permettant de calculer $e^x$ pour x donnée avec une précision $10^{-3}$ et sachant que :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

```
Constante utilisée
E=0.001
Variables
Réel x
fonction  expo  (Donnée x: réel): réel
Entier i
réel f, ex
Début
  f ←1    ex ←1     i ←1
  Tant que f ≥ E faire
    f ← f * ( x / i )
    ex ← ex + f
    i ← i + 1
  ftq
  Retour   ex;
Fin
```

```
{ programme principal }

Début
  Écrire " Donner la valeur de x: "
  Lire x
  Écrire   "L'exponentielle de x= ", x,
     "est : ", expo(x)
Fin
```

# Traduction en C

```
#include <stdio.h>
#define E  0.001  //définition d'une constante en C

float expo(float x) {
  int i;
  float f, ex;
  f=1;
  ex=1;
  i=1;
  while (f>=E) {
     f=f*(x/i);
     ex=ex+f;
     i++;
  }
  return ex;
}
....
```

# Traduction en C

```
/*Programme principal */
…
Int main() {
  float x;
  printf("Donner la valeur de x : ");
  scanf("%f",&x);
  printf("L'exponentiel de x=%f est  %f\n", x, expo(x));
  return 0;
}
```

# Exercice

- Ecrire un programme se servant d'une fonction MOYENNE du type float pour afficher la moyenne arithmétique de deux nombres réels entrés au clavier.
- Ecrire une procédure en LDA qui permet de permuter deux entiers passées en paramètres, puis traduire en langage C
- Ecrire une procédure en LDA qui permet d’inverser un vecteur passé en paramètre puis traduire en langage C
- Ecrire une fonction Combinaison() qui se sert de la fonction factorielle pour recalculer et retourner la combinaison de deux entiers passées en paramètres

# 4 Récursivité

# 4 Récursivité

## Introduction

Récursivité (informatique) ←→ récurrence (maths)

Une démonstration par récurrence d'une propriété p(n) avec n un entier >0:

- Propriété vraie pour une valeur initiale $n_0$;
- Montrer que si la Propriété est vraie pour n alors vraie pour n+1

# 4 Récursivité

## Introduction

- Même principe pour le  programme récursif mais en sens inverse.  Ainsi, pour réaliser une tâche sur des données de taille n, on va devoir d'abord la réaliser sur des données de taille n-1.
- Appeler de façon récursive le même  programme en diminuant la taille des arguments.
- L'équivalent de l'initialisation d'une preuve par récurrence est une condition d'arrêt de l'appel récursif.
- Lorsque la taille est suffisamment petite alors s’exécute la partie non récursive du programme.

# 4 Récursivité

## Définition

Une fonction ou un algorithme est dit récursif lorsqu'il est **auto-référent** : elle fait référence à elle-même (directement ou indirectement)dans sa définition.

**Direct.**

```
Int  fonct(…)   {
     x = fonct(…);
}
```

**Indirect.**

```
Int f1(…){
x=f2(…);
}
```

```
Int f2(…){
x=f1(…);
}
```

# 4 Récursivité

## Motivation

Cela simplifie la résolution du problème.

**Diviser pour régner** : réappliquer un même traitement sur un échantillon de données d'une taille de plus en plus petite.

- La Récursivité sert aussi pour définir des structures de données : arbres, expressions arithmétiques, grammaires…
- La Récursivité est utilisable dans tous les langages de programmation modernes. En particulier, les langages **fonctionnels**.

# 4 Récursivité

## Propriétés

Trois éléments à considérer lors de l'élaboration d'une procédure récursive :

- **Expression récursive du problème** :

  L' « équation » de la récursivité.

- **Condition d'arrêt :**

  Quand est-ce qu'on arrête les appels récursifs?

- **Convergence** (vers la condition d'arrêt):

  Une petite « preuve» et les conditions qui nous assurent qu'un jour on va atteindre la condition d'arrêt.

# 4 Récursivité

## Propriétés

Une fonction récursive doit posséder les deux propriétés suivantes:

- il doit exister certains critères, appelés critères d'arrêt ou conditions d'arrêt, pour lesquels la fonction ne s’appelle pas elle-même;

- chaque fois que la procédure s’appelle elle-même (directement ou indirectement), elle doit converger vers ses conditions d'arrêt.

**Une fonction récursive possédant ces deux propriétés est dite bien définie.**

# 4 Récursivité

## Exemple La factorielle (!)

- **Expression récursive du problème :**

  $n! = n(n-1)\ldots(2)(1) = n(n-1)!$

- **Condition d'arrêt:**

  $n = 1$ ou $n = 0$

- **Convergence (vers la condition d'arrêt):**
  - Si $n = 1$ ou $n = 0$, alors on a «convergé »!
  - Si $n > 1$, alors la soustraction à l'étape suivante nous approche de $n = 1$
  - si n est une entier non négatif ça converge!

# 4 Récursivité

## Exemple La factorielle (n!)

```
Int factorielle(int n)
{

        if (n == 0 || n == 1)
                return 1; /* condition d'arrêt */

        else /* appel récursif */
                return n * factorielle(n - 1);
}
```

Exercice d'application: proposer un algorithme récursif pour calculer le pgcd(m,n).

# 4 Récursivité

## Itératif vs récursif

- Exemple : suite de *Fibonacci*
  - $U_0 = U_1 = 1$
  - $Un = U_{n-1} + U_{n-2}$
- Solution intuitive (récursive) :

```
fonction fibo(n : entier) : entier
début
  si (n = 0) ou (n = 1)
  alors retourne 1
  sinon retourne fibo(n-1) + fibo(n-2)
fin
```

# 4 Récursivité

## Itératif ou récursif

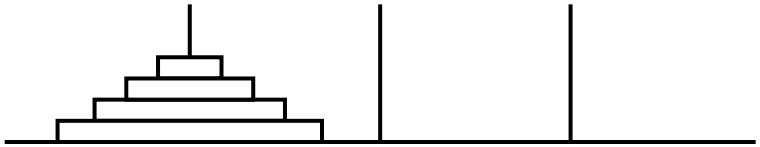
### Solution itérative

```
fonction fibo(n : entier) : entier
var a, b, c, i : entier
début
  b = c = 1
  pour i = 2 jusqu'à i = n faire
    a = b + c
    c = b
    b = a
  fait
  retourne a
fin
```

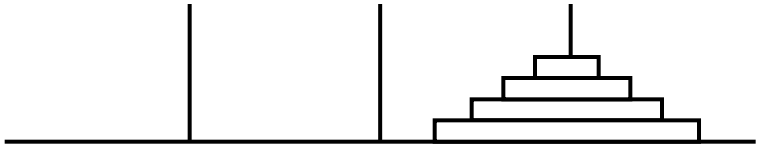
# 4 Récursivité

## Itératif ou récursif

- Les tours de Hanoï
  - Situation initiale :



  - Situation finale :



  - Déplacements autorisés
    - Un disque à la fois
    - Toujours sur un disque de dimensions supérieures (ou sur le plateau)

# 4 Récursivité
## Itératif ou récursif

- Solution itérative : très complexe
- Solution récursive :

```
procédure déplacer(n, pile1, pile2, pile3 )
début
  si (n > 1) alors
    déplacer(n-1, pile1, pile3, pile2)
    déplacer(1, pile1, pile2, pile3)
    déplacer(n-1, pile2, pile1, pile3)
  sinon
    afficher("déplacement d'1 disque de " + pile  1 +
             "vers" + pile3
  fsi
fin
```

# Exercice

1. Écrivez une fonction récursive qui compte le nombre de caractères numériques (0 à 9) dans une chaîne de caractères. Proposez une version itérative de cette fonction.
2. Écrivez une fonction récursive qui compare deux chaînes de caractères ch1 et ch2, et qui retourne TRUE si ch1 = ch2, et FALSE sinon.

# 5 Algorithmes de recherche

À de très nombreuses occasions, un programmeur est amené à soumettre l'exécution d'un calcul ou d'une instruction à la condition qu'une certaine valeur appartienne ou pas à un ensemble donné: **c'est la recherche**.

Un ensemble définit le regroupement d'éléments sans présupposer quoi que soit quant à la structure de ce regroupement

# 5 Algorithmes de recherche

## Rechercher une information dans un tableau

### Recherche séquentielle

Le principe est simple, on va parcourir les cases du tableau dans l'ordre croissant des indices jusqu'à ce qu'on trouve l'élément x, ou bien jusqu'à ce qu'on arrive à la fin, sans avoir trouver l'élément x.

# 5 Algorithmes de recherche

## Rechercher une information dans un tableau

Recherche séquentielle

Méthode :

- Soit T un tableau de N éléments et val l'élément cherché
- parcours du tableau à partir du premier élément (T[0])
- Arrêt quand élément trouvé ou si fin de tableau (T[n-1])

```
Algorithme recherche_sequentielle {Recherche le premier indice où se trouve la valeur val parmi les N données du tableau tab;}
variables : T [0, N-1], val  : entier
                    n, val, indice : entier
DÉBUT
indice ←0
tant que ( val <> T[indice] && indice < N-1) faire
        indice ←indice + 1  //<> signifie la différence
 ftq
 si T[indice] = val  alors
        afficher( " l'élément se trouve en : " indice);
 sinon
         afficher(" Elément non présent " );
fsi
FIN
```

# 5 Algorithmes de recherche

## Rechercher une information dans un tableau

Recherche dichotomique

Il s'agit de rechercher une information dans un **tableau trié**

Méthode :

- Soit T un tableau de N éléments et val l'élément cherché
- T est trié
- Sinon effectuer un prétraitement de tri.
- Comparer val avec l'élément u milieu du tableau T.
  - Si c'est le même => trouvé
  - sinon on recommence sur la première moitié ou la seconde selon que val est < à valmid ou val > valmid
- Arrêt quand élément trouvé ou si fin de tableau (T[indice-1])

```
Algorithme recherche_dichotomique
variables T [0, N-1] , val  :entier
                lnf, Sup, N, Mi : entier
DÉBUT
        Saisir (val)
        Inf ← 0
        Sup ← N-1
        Mi ← (Inf + Sup)/2
        tant que ( val <> T[Mi] && Inf <= Sup) faire
                    si val < T[Mi] alors
                            Sup = Mid - 1
                    sinon
                                    Inf = Mid + 1
                   fsi
                   Mi ← (Inf + Sup)/2
         ftq
         si T[Mi] = val  alors
                      afficher( " l'élément se trouve en : " Mi);
         sinon
                      afficher(" Elément non présent " );
          fsi
FIN
```