



# Chapitre 1

- ✓ **Les variables**
- ✓ **LA STRUCTURE ALTERNATIVE**
- ✓ **LA STRUCTURE REPETITIVE**

# QU'EST CE QUE L'ALGORITHMIQUE ?

## DEFINITION

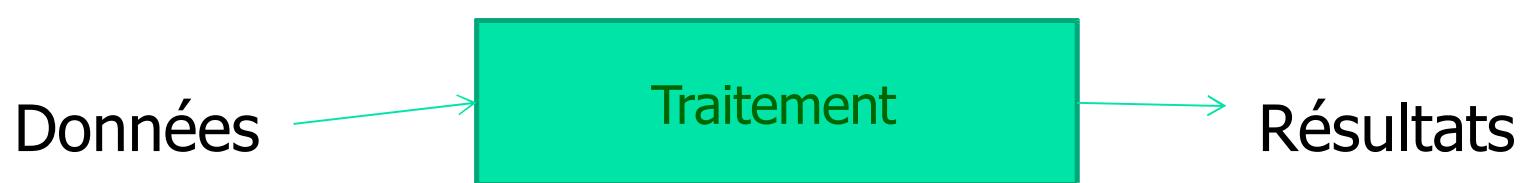
**L'algorithme** est un terme d'origine arabe. C'est la description d'un traitement automatisé de données destiné à être réalisé sur un ordinateur, après être traduit dans un langage de programmation.

**L'algorithlique** consiste , après analyse du problème à résoudre, à définir cette description du traitement, et ce, de manière totalement indépendante du langage qui sera choisi pour la programmation.

# QU'EST CE QUE L'ALGORITHMIQUE ?



De manière générale , un traitement automatisé consiste à effectuer des opérations sur des informations appelées données d'entrée et à fournir d'autres informations appelées résultats ou données de sortie.



# QU'EST CE QUE L'ALGORITHMIQUE ?



Calcul de la surface d'un cercle de rayon R

- \* Saisir le rayon du cercle
- \* Affecter à Surface le résultat de l'expression  $\pi \times \text{rayon}^2$
- \* Afficher le résultat

Les ordinateurs quelqu'ils soient, ne sont fondamentalement capables d'exécuter que quatre opérateurs logiques. Ces opérateurs sont :

- \* L'affectation des variables
- \* La lecture/ écriture
- \* Les tests
- \* Les boucles

# QU'EST CE QUE L'ALGORITHMIQUE ?

## ALGORITHME

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre notions de base. Si l'algorithme est juste, le résultat est celui voulu ; si l'algorithme est faux, le résultat n'est pas celui escompté.

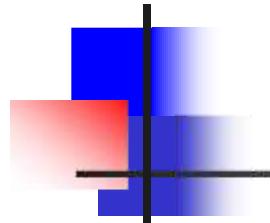
# QU'EST CE QUE L'ALGORITHMIQUE ?

## Caractéristiques d'un algorithme

Il doit donc être :

- **lisible:** l'algorithme doit être compréhensible même par un non-informaticien
- **de haut niveau:** l'algorithme doit pouvoir être traduit en n'importe quel langage de programmation, il ne doit donc pas faire appel à des notions techniques relatives à un programme particulier ou bien à un système d'exploitation donné
- **précis:** chaque élément de l'algorithme ne doit pas porter à confusion, il est donc important de lever toute ambiguïté

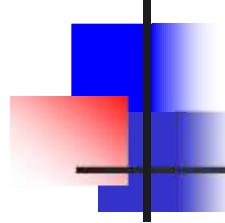
# QU'EST-CE QUE L'ALGORITHMIQUE ?



## Caractéristiques d'un algorithme

- **concis:** un algorithme ne doit pas dépasser une page. Si c'est le cas, il faut décomposer le problème en plusieurs sous-problèmes
- être le plus général possible afin de répondre au plus grand nombre de cas possibles
- être conçu de manière à limiter le nombre d'opérations à effectuer et la place occupée en mémoire
- **structuré:** un algorithme doit être composé de différentes parties facilement identifiables

# **QU'EST-CE QUE L'ALGORITHMIQUE ?**



## **Programmation structurée**

- Afin de répondre à ces critères, il faut utiliser la programmation structurée qui utilise la notion, de programmation descendante (top down).
  
- Elle consiste à décomposer le problème en plusieurs sous-problèmes plus simples qui seront traités séparément et éventuellement décomposés eux-mêmes de manière plus fine.

# **QU'EST CE QUE L'ALGORITHMIQUE**

## **Programme**

Les étapes de résolution d'un problème sont :

Enoncé du problème et données



Spécification



Cahier des charges



Analyse



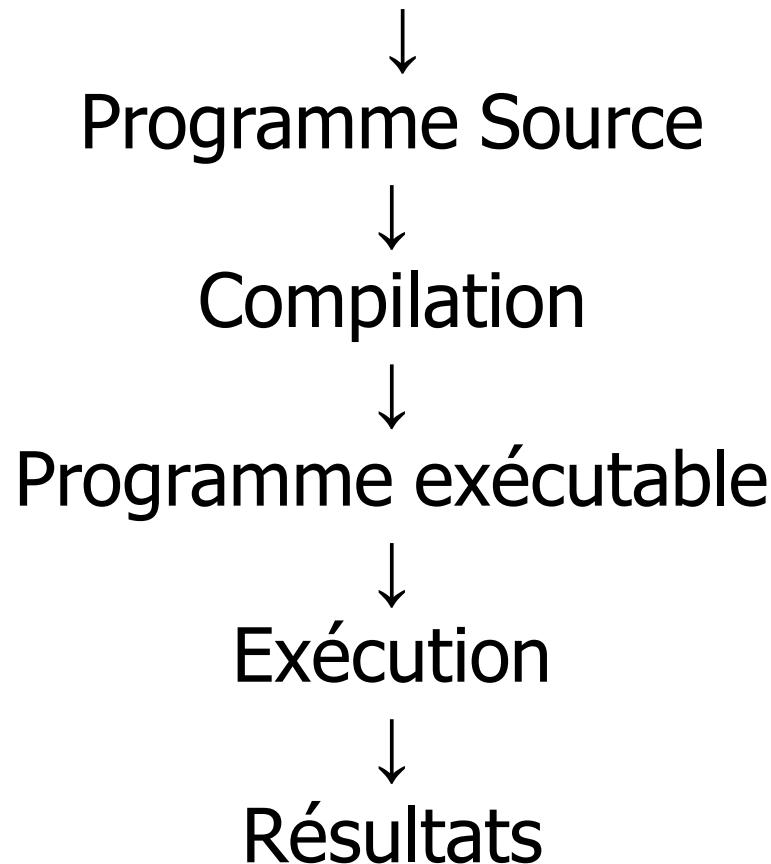
Algorithme



Traduction en langage et édition

# QU'EST CE QUE L'ALGORITHMIQUE

## Programme



# QU'EST CE QUE L'ALGORITHMIQUE

## Programme

Une fois l'algorithme est élaboré, il est traduit dans un langage de programmation

Le langage de programmation est l'intermédiaire entre l'humain et la machine, il permet d'écrire dans un langage proche de la machine mais intelligible par l'humain les opérations que l'ordinateur doit effectuer.

# QU'EST-CE QUE L'ALGORITHMIQUE ?

## Programme

Durant l'écriture d'un programme, on peut être confronté à deux types d'erreur :

- Les erreurs syntaxiques : elles se remarquent à la compilation et sont le résultat d'une mauvaise écriture dans le langage de programmation.
- les erreurs sémantiques : elles se remarquent à l'exécution et sont le résultat d'une mauvaise analyse. Ces erreurs sont beaucoup plus graves car elles peuvent se déclencher en cours d'exploitation du programme.

# QU'EST CE QUE L'ALGORITHMIQUE ?

## Programme

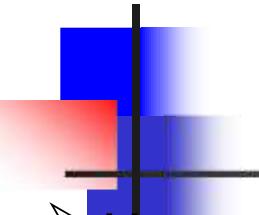
- Le programme est ensuite transformé en langage machine lors d'une étape appelée **compilation**. La *compilation* est une phase réalisée par l'ordinateur lui-même grâce à un autre programme appelé **compilateur**.
- La phase suivante s'appelle l'édition de liens, elle consiste à lier le programme avec tous les éléments externes (généralement des librairies auxquelles il fait référence).

# Exemple de programme en C

```
#include <stdio.h>

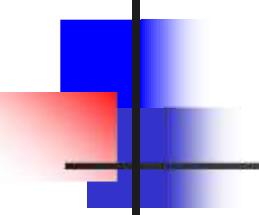
void main( )
{
    int i, borne, somme=0;
    /* Saisie*/Printf(<< somme=%d \n >>,somme);

    scanf(<< %d >>,&borne);
    for(i=1;i<=borne;i++) somme=somme+i;
    // Affichage Printf(<< somme=%d \n >>,somme);
    Printf(<< somme=%d \n >>,somme);
}
```



## FORME GENERALE D'UN PROGRAMME EN C

- Un programme source C se présente sous forme d'une collection d'objets externes (variables et fonctions), dont la définition est éventuellement donnée dans des fichiers séparés
- Tout programme C est composé d'un programme principal i.e. la fonction qui porte le nom main
- Une variable est un objet manipulé par le programme possédant un nom et un type qui définit l'ensemble des valeurs possibles pour l'objet
- En C, tout module (ss-programme) porte le nom de fonction. Toutes les fonctions, dont le programme principal, sont constituées d'un bloc qui est une suite d'instructions à l'intérieur de « { } »



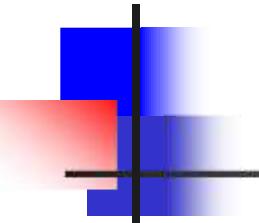
# Programme en C

## Les variables

### Qu'est ce qu'une variable ?

Une variable est un emplacement mémoire qui sert à stocker une valeur qui peut changer pendant l'exécution d'un programme. Elle est définie par cinq éléments :

- **L'identificateur:** c'est le nom que l'on donne à la variable.
- **Le type:** il détermine la nature de l'information (nombre entier, nombre réel, caractère, ...).
- **La taille:** c'est le nombre d'octets occupés en mémoire, elle est en fonction du type.
- **La valeur:** c'est la valeur que l'on attribue à la variable.
- **L'adresse:** c'est l'emplacement où est stockée la valeur de la variable.



# Programme en C

## Les variables

### Identificateur d'une variable

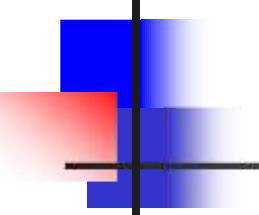
Il existe un certain nombre de limites pour choisir l'identificateur

- Il peut contenir des lettres minuscules ou majuscules, des chiffres, ou le caractère spécial de soulignement « \_ ».
- il ne doit pas commencer par un chiffre ou posséder des lettres accentuées
- les espaces ne sont pas admis dans l'identificateur
- Il ne doit pas comporter plus de 32 caractères
- les majuscules sont distinguées des minuscules,
- il ne peut pas être un mot réservé du langage

### Déclaration d'une variable

Une variable peut être initialisée lors de sa déclaration.

**Syntaxe : Type identificateur [= valeur\_initiale];**



# Programme en C

## Constantes symboliques : **const**

Une constante est une variable dont l'initialisation est obligatoire et dont la valeur ne pourra pas être modifiée en cours d'exécution. Elle est déclarée avec le mot clé : **const** qui doit précéder le type.

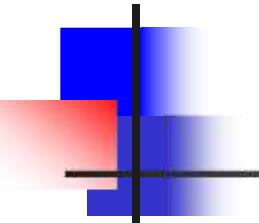
**Syntaxe :** **const** **type** identificateur = valeur\_initiale;

### Exemples :

```
const float PI = 3.14;
```

```
const float x = 0.2;
```

```
x = 0.4; Erreur !
```



## Types de base

**Entier : int** (2 octets ou 4 octets)

Format d'affichage : %d

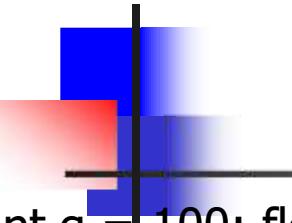
**Flottant ou réel : float** (4 octets) ou **double** (8 octets)

Format d'affichage : %f (flottant) %lf (double)

**Caractère : char** (1 octet)

Format d'affichage : %c pour les caractères et %s pour

les chaînes de caractères



## Exemples

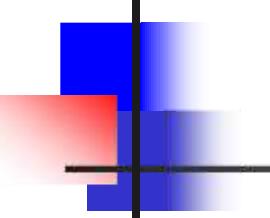
```
int q = 100; float p = 60.50; char t ='A';
printf("Q = %d\n",q); printf("P = %f\n", p); printf("T = %c\n",t) ;printf("R = %f", q*p);
ou bien
printf("Q = %d\nP = %f\nT = %c\nR = %f",q,p,t,q*p);
```



Affiche à l'écran

```
Q = 100
P = 60.500000
T = A
R = 6050.00000
```

```
float val = 25.1234;
printf("%f",val); /* affiche : 25.123400 */
printf("%.0f",val); /* affiche : 25 */
printf("%.1f",val); /* affiche : 25.1*/
printf("%.2f",val); /* affiche : 25.12*/
```



# Programme en C

## Expressions

Une *expression* peut être une variable, une constante, un appel d'une fonction (avec retour de valeur) ou d'une combinaison de chacun de ces éléments par des opérateurs. Toute expression a un type et une valeur.

### Exemples :

- $b^2 - 4ac$
- $(-b + \sqrt{b^2 - 4ac}) / (2a)$

# Programme en C

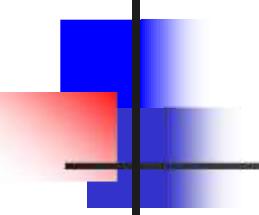
## Opérateurs

### Opérateurs élémentaires

- Arithmétiques : +, -, \*, /, %
- Relation : == , < , >, <=, >=, !=
- Logique : ! (négation ) , && ( et) , || (ou)
- Affectation : =
- Opérateur conditionnel : <exp1> ? <exp2> : <exp3>

### Combinaisons des opérateurs

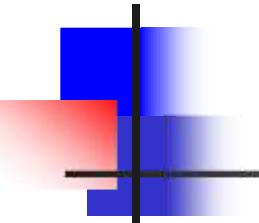
- i++ ou ++i
- i- - ou - -i
- i+=j
- i-=j
- /= , \*= , %=



# Programme en C

## Les instructions

- Une instruction simple est soit une expression terminée par *un point virgule, soit une instruction de contrôle*. Exemple : `x=x +1;`
  - Les instructions composées (ou blocs) permettent de considérer une succession d'instructions comme étant une seule instruction :
    - elles commencent par "{" et finissent par "}"
    - l'accolade ouvrante est l'équivalente du début en algorithme
    - l'accolade fermante est l'équivalente de la fin en algorithme
- ```
{ float a=6.5, b;  int c=50;
    b= c * a;
}
```

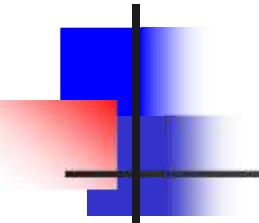


# Programme en C

## **Inclusion des fichiers :**

La première ligne contient la directive #include suivi d'un nom de fichier à pour effet d'insérer le fichier spécifié entre < et >, par exemple stdio.h, dans le fichier source à l'endroit où la directive est placée.

Le fichier d'en-tête stdio.h contenant les déclarations nécessaires à l'utilisation des fonctions d'entrées-sorties standard. Le compilateur dispose ainsi des informations nécessaires pour vérifier si l'appel de la fonction (en l'occurrence printf et scanf) est correct.



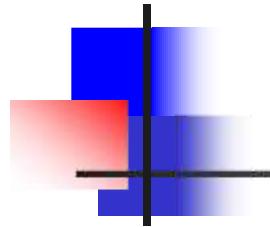
# Programme en C

**Fonction main()**: Tout programme C doit contenir au moins une fonction appelée main. Le code qu'il contient est encadré par deux accolades, { et }. L'exécution du programme débute immédiatement après l'accolade ouvrante et se termine lorsque l'accolade fermante correspondante est rencontrée.

**Commentaires** : Les commentaires permettent de documenter les programmes sources. Ils sont encadrés par " /\*" et par " \*/".

On peut utiliser des « commentaires de fin de ligne » en introduisant les deux caractères : //. Dans ce cas, tout ce qui est situé entre // et la fin de la ligne est un commentaire. Les commentaires sont alors ignorés par le compilateur.

# **LA STRUCTURE ALTERNATIVE**



## **INTRODUCTION**

Rares sont les algorithmes qui peuvent se décrire uniquement par un enchaînement séquentiel d'opérations élémentaires : la plupart du temps, les traitements informatiques font appel à des concepts de ruptures de séquence comme les tests, (ou opérations conditionnelles) et les boucles.

# LA STRUCTURE ALTERNATIVE

## TESTS SIMPLES

La syntaxe est :

```
si condition alors bloc d'instructions1  
sinon         bloc d'instructions2  
fsi
```

Cette instruction est composée de trois parties distinctes:

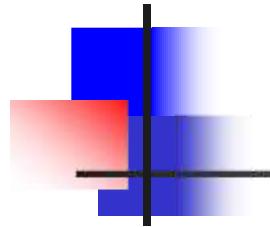
- ➥ la condition introduite par ***si***
- ➥ la clause ***alors***
- ➥ et la clause ***sinon***

# LA STRUCTURE ALTERNATIVE

## DEFINITION

- La condition est une expression dont la valeur est de type booléen. Elle s'exprime sous la forme d'une expression logique simple (condition unique) ou combinée (plusieurs conditions composées avec des opérateurs logiques ET, OU et NON)
- Chaque bloc d'instructions est composé d'une série d'instructions qui, au sein du bloc, sont exécutées en séquence. Dans le cas contraire, les instructions de la clause *sinon* sont exécutées.

# LA STRUCTURE ALTERNATIVE



## TESTS SIMPLES

- La clause **sinon** n'est pas obligatoire. Ainsi, si la condition est remplie, on exécute le bloc d'instructions1, dans le cas contraire, on ne fait rien de spécial.
- La syntaxe est alors :  
**si** condition   **alors**   bloc d'instructions  
**fsi**

Chaque bloc d'instructions peut, évidemment, se limiter à une seule instruction.

# LA STRUCTURE ALTERNATIVE



## DEFINITION

En C les 'conditions' peuvent être des expressions quelconques qui fournissent un résultat numérique. La valeur zéro correspond à la valeur logique « **faux** » et toute valeur différente de zéro est considérée comme « **vrai** ».

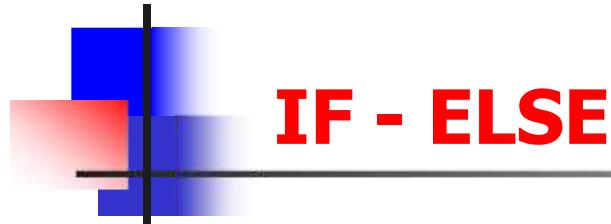
# LA STRUCTURE ALTERNATIVE

## IF - ELSE

```
if (<expression>)
    <bloc d'instructions1>
else
    <bloc d'instructions2>
```

- Si <l'expression > fournit une valeur différente de 0 alors le <bloc d'instructions1> est exécuté
- Si <l'expression > fournit la valeur 0 alors le <bloc d'instructions2> est exécuté

# LA STRUCTURE ALTERNATIVE



## IF - ELSE

- La partie <**expression**> peut désigner :
  - \* Une variable d'un type numérique
  - \* Une expression fournissant un résultat numérique
  
- ➥ La partie <**bloc d'instructions**> peut désigner :
  - \* Plusieurs instructions comprises entre accolades
  - \* Une seule instruction délimitée par un point-virgule

# LA STRUCTURE ALTERNATIVE

## Exemples

```
int qte_cmd;
float prix_unitaire, remise=0 ;
printf("La quantité commandée : ");
scanf("%d",&qte_cmd) ;
printf("Prix unitaire : ") ;scanf("%f",&prix_unitaire) ;
if(qte_cmd > 100) remise = 0.2;
printf("Prix à payer : %f",prix_unitaire*qte_cmd*(1 - remise) ;

char car ;
printf("Tapez une lettre minuscule : ");
scanf("%c",&car);
if (car == 'a' || car == 'e' || car == 'i' || car == 'o' || car == 'u' || car == 'y')
    printf("la lettre %c est une voyelle",car);
else
    printf("la lettre %c est une consonne",car);
```

# LA STRUCTURE ALTERNATIVE

## IF sans ELSE

**If** (**<expression>**)      <bloc d'instructions>

**Attention :** Comme la partie **else** est optionnelle, les expressions combinant plusieurs structures **if** et **if – else** telles que :

**if (<expression1>) if (<expression2>) <instruction1> else <instruction2>**  
peuvent mener à des confusions.

**Exemple :**

**if (a < b) if (c < b) z = b ; else z = a ;**

A quel « **if** » **se rapporte** « **else** » ?

Règle : le « **else** » se rapporte au dernier « **if** » rencontré auquel un « **else** » n'a pas encore été attribué sauf si on utilise les accolades.

# LA STRUCTURE ALTERNATIVE

## IF sans ELSE

La bonne mise en page du code ci-dessus est :

```
if (a < b)
    if (c < b) z = b ;
    else z = a ;
```

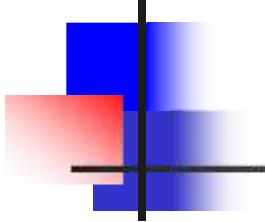
On peut mettre des { } pour rendre les choses plus claires.

```
if (a < b)
{
    if (c < b) z = b ;
    else z = a ;
}
```

Si l'on veut que « else » se rapporte au premier « if » :

```
if (a < b)
{
    if (c < b) z = b ;
}
else z = a ;
```

# LA STRUCTURE ALTERNATIVE



**if - else if - ... - else**

En combinant plusieurs structures **if - else** en une expression nous obtenons une structure qui est très courante pour prendre des décisions entre plusieurs alternatives:

```
if ( <expr1> )      <bloc1>
else if (<expr2>)    <bloc2>
else if (<expr3>)    <bloc3>
else if (<exprN>)    <blocN>
else    <blocN+1>
```

# LA STRUCTURE ALTERNATIVE

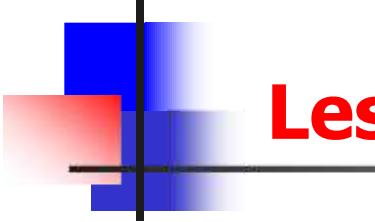
- 
- Les expressions **<expr1>** ... **<exprN>** sont évaluées du haut vers le bas jusqu'à ce que l'une d'elles soit différente de zéro. Le bloc d'instructions y lié est alors exécuté et le traitement de la commande est terminé.
  - La dernière partie **else** traite le cas où aucune des conditions n'a été remplie. Elle est optionnelle, mais elle peut être utilisée très confortablement pour détecter des erreurs.

# LA STRUCTURE ALTERNATIVE

## Exemple

```
#include <stdio.h>
void main() {
int A,B;
printf("Entrez deux nombres entiers :");
scanf("%d %d", &A, &B);
if (A > B)
    printf("%d est plus grand que %d", A, B);
else if (A < B)
    printf("%d est plus petit que %d\n", A, B);
else
    printf("%d est égal à %d\n", A, B);
Printf(« FIN »);
}
```

# LA STRUCTURE ALTERNATIVE



## Les opérateurs conditionnels

**<expr1> ? <expr2> : <expr3>**

- Si <expr1> fournit une valeur différente de zéro (i.e.vraie), alors la valeur de <expr2> est fournie comme résultat
- Si <expr1> fournit la valeur zéro (i.e. fausse), alors la valeur de <expr3> est fournie comme résultat
- l'opérateur conditionnel « ?: » a l'avantage de pouvoir être intégré dans une expression

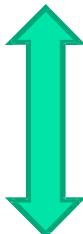
# LA STRUCTURE ALTERNATIVE

## Exemple

```
int a = 5, b = 4, max;  
if (a>b)  
    max=a;  
else  
    max=b;  
printf("Le max est : %d",max);
```



```
int a = 5, b = 4, max;  
max = (a>b) ? a : b;  
printf("Le max est : %d",max) ;
```



```
int a = 5, b = 4;  
printf("Le max est : %d", (a>b) ? a : b) ;
```

# LA STRUCTURE ALTERNATIVE

## OPÉRATIONS CONDITIONNELLES MULTIPLES

L'instruction conditionnelle

Au cas où

expression = valeur\_1

    <Bloc d'instructions\_1>

expression = valeur\_2

    <Bloc d'instructions \_2>

expression = valeur\_n

    <Bloc d'instructions\_n>

sinon

    <Bloc instruction\_par\_défaut;]>

Fin-cas

# LA STRUCTURE ALTERNATIVE

## OPÉRATIONS CONDITIONNELLES MULTIPLES

se traduit en langage C par « **switch** » : elle permet de remplacer plusieurs if-else imbriqués lorsqu'il s'agit d'effectuer un choix multiple

**switch (expression)**

```
{      case valeur_1 :  
            instruction_1;  
            [break;]  
      case valeur_2 :  
            instruction_2;  
            [break;]  
      ....  
      case valeur_n :  
            instruction_n;  
            [break;]  
      [default:  
            instruction_par_défaut;]  
}
```

# LA STRUCTURE ALTERNATIVE

## OPÉRATIONS CONDITIONNELLES MULTIPLES

- expression et valeur\_i ne peuvent être ni de type réel ni de type chaîne de caractères
- valeur\_1, valeur\_2,..., valeur\_n doivent obligatoirement être distinctes
- break et default sont optionnels
- Les parenthèses qui suivent le mot clé *switch* indiquent une expression dont la valeur est testée successivement par chacun des *case*. Lorsque l'expression testée est égale à une des valeurs suivant un *case*, la liste d'instructions qui suit celui-ci est exécutée. Le mot clé *break* indique la sortie de la structure conditionnelle. Le mot clé *default* précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une valeur
- Si on oublie d'insérer des instructions *break* entre chaque test, aucune erreur n'est signalée. L'exécution continue dans les blocs suivants !

# LA STRUCTURE ALTERNATIVE

## OPÉRATIONS CONDITIONNELLES MULTIPLES

**N.B :** Si on veut faire exécuter les mêmes instructions pour différentes valeurs consécutives, on peut ainsi mettre plusieurs cases avant le bloc :

switch(variable)

{

case 1:

case 2:

{ instructions exécutées pour variable = 1 ou pour variable = 2 }

break;

case 3:

{ instructions exécutées pour variable = 3 uniquement }

break;

default:

{ instructions exécutées pour toute autre valeur de variable }

}

# LA STRUCTURE ALTERNATIVE

## OPÉRATIONS CONDITIONNELLES MULTIPLES

```
void main()
{
int jour;
printf("Entrez le numéro d'un jour de la
semaine (1 à 7) : ");
scanf("%d",&jour);
printf("Le jour %d de la semaine est le ",jour);
switch (jour)
{
case 1 : printf("DIMANCHE"); break;
case 2 : printf("LUNDI"); break;
case 3 : printf("MARDI"); break;
case 4 : printf("MERCREDI"); break;
case 5 : printf("JEUDI"); break;
case 6 : printf("VENDREDI"); break;
case 7 : printf("SAMEDI"); break;
default: printf("Erreur!");
}
```



```
void main()
{
int jour;
printf("Entrez le numéro d'un jour de la semaine
(1 à 7) : ");
scanf("%d",&jour);
printf("Le jour %d de la semaine est le ",jour);
if (jour == 1) printf("DIMANCHE");
else if (jour == 2) printf("LUNDI");
else if (jour == 3) printf("MARDI");
else if (jour == 4) printf("MERCREDI");
else if (jour == 5) printf("JEUDI");
else if (jour == 6) printf("VENDREDI");
else if (jour == 7) printf("SAMEDI");
else printf("Erreur!");
}
```

# LA STRUCTURE REPETITIVE

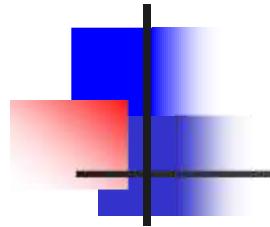
## TANT QUE

L'algorithme s'écrit donc sous la forme suivante :

```
Tant que <expression logique>
    Bloc d'instructions
Fin-tantque
```

- Au moment du premier passage dans la boucle, la condition est évaluée, si elle est vérifiée, le bloc d'instructions est exécuté en séquence
- A la fin de l'exécution de cette séquence, on évalue à nouveau la condition et on répète l'exécution du bloc tant que la condition est vérifiée. Dès que celle-ci devient fausse, l'exécution du programme se poursuit à partir de la 1<sup>ère</sup> instruction qui suit immédiatement la marque fin-tque.

# LA STRUCTURE REPETITIVE



## TANT - QUE

- Il se peut que le bloc d'instructions ne soit jamais exécuté si, par exemple, la condition n'est pas vérifiée au moment du premier passage
- La condition de « tant que » peut rester toujours vraie : dans ce cas on a une boucle infinie
- Le bloc d'instructions est exécuté zéro ou plusieurs fois.

# LA STRUCTURE REPETITIVE

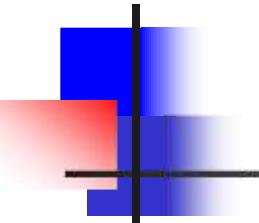


## WHILE

while ( <expression> )      <bloc d'instructions>

- Tant que l'<expression> fournit une valeur différente de zéro, le <bloc d'instructions> est exécuté.
- Si l'<expression> fournit la valeur zéro, l'exécution continue avec l'instruction qui suit le bloc d'instructions.
- Le <bloc d'instructions> est exécuté zéro ou plusieurs fois
- La partie <expression> peut désigner : une variable d'un type numérique ou une expression fournissant un résultat numérique.
- La partie <bloc d'instructions> peut désigner : un (vrai) bloc d'instructions compris entre accolades ou une seule instruction terminée par un point-virgule

# LA STRUCTURE REPETITIVE



## Exemple

---

```
void main() {  
    int i = 0;  
    while (i<10) {  
        printf("%d \n", i);  
        i++;  
    }  
}
```

# LA STRUCTURE REPETITIVE

## Exemple-while

```
/*Affiche les nombres de 0 à 9*/
```

```
int I = 0;
```

```
while (I<10) printf("%d \n", I++);
```

→ post-incrémantation

```
/* Affiche les nombres de 1 à 10 */
```

```
int I = 0;
```

```
while (I<10) printf("%d \n", ++I);
```

→ pré-incrémantation

```
/*Affiche les nombres de 10 à 1*/
```

```
int I=10;
```

```
while (I) printf("%d \n", I--);
```

→ post-décrémantation

# LA STRUCTURE REPETITIVE

## Boucle JUSQU'A

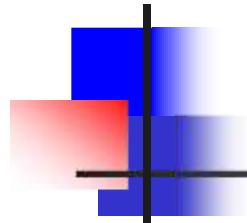
- Dans une boucle « Tant que », la condition est évaluée avant d'exécuter la première instruction du bloc dans la boucle.
- On peut souhaiter, dans certains cas , évaluer la condition après l'exécution de la dernière instruction de ce bloc.
- Un tel algorithme correspond à une boucle jusqu'à et s'écrit ainsi :

Répéter

<Bloc d'instructions>

jusqu'à condition

# LA STRUCTURE REPETITIVE



## Boucle JUSQU'A

- Le bloc d'instructions sera répété jusqu'à ce que la condition soit vérifiée
- Contrairement à une boucle tant que, l'utilisation de la boucle jusqu'à garantit que le bloc sera exécuté au moins une fois puisque le test a lieu après son exécution.
- On parle parfois, pour la boucle jusqu'à, d'un test de sortie tandis que dans le cas d'une boucle tant que, il s'agit plutôt d'un test d'entrée dans la boucle.
- En langage C, la syntaxe est :

```
do      <bloc d'instructions>
      while ( <expression> );
```

# LA STRUCTURE REPETITIVE

## DO - WHILE

➤ La structure **do-while** est semblable à la structure **while**, avec la différence suivante :

\***while** évalue la condition **avant** d'exécuter le bloc d'instructions,

\***do - while** évalue la condition **après** avoir exécuté le bloc d'instructions. Il est alors exécuté au moins une fois.

➤ Le <bloc d'instructions> est exécuté au moins une fois et aussi longtemps que l'<expression> fournit une valeur différente de zéro

int N;

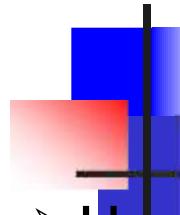
```
do { printf("Introduisez un nombre entre 1 et 10 :");
      scanf("%d", &N);
    }while (N<1 || N>10);
```

# LA STRUCTURE REPETITIVE

## EXEMPLE

```
/* calcul de la racine carrée */
#include <stdio.h>
#include <math.h>
void main(){
    float N;
    do {
        printf("Entrer un nombre (>= 0) : ");
        scanf("%f", &N);
    }while (N < 0);
    printf("La racine carrée de %f est %f\n", N,sqrt(N));
}
```

# LA STRUCTURE REPETITIVE



## Boucle FOR

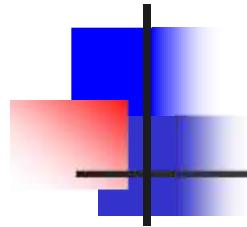
- Une itération consiste à exécuter un bloc d'instructions un certain nombre de fois. En général, le nombre de passages dans la boucle est connu au préalable.
- Une variable entière particulière appelée variable d'itérations sert à compter le nombre de répétitions du bloc d'instructions. Cette variable est initialisée avant le 1<sup>er</sup> passage dans la boucle, incrémentée à chaque passage et lorsqu'elle atteint sa valeur pré-définie, l'exécution du programme se poursuit à partir de la 1<sup>ère</sup> instruction qui suit immédiatement la boucle.

L'algorithme est le suivant :

Pour i variant de <valeur initiale> à <valeur finale>  
    <Bloc d'instructions>

fin-pour

# LA STRUCTURE REPETITIVE



## Boucle FOR

En langage C :

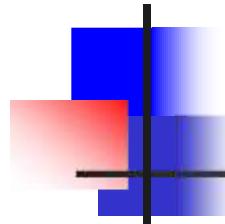
for ( <expr1> ; <expr2> ; <expr3> )

<bloc d'instructions>

ce qui est équivalent à :

```
<expr1>;  
while ( <expr2> )  
{   <bloc d'instructions>  
    <expr3>;  
}
```

# LA STRUCTURE REPETITIVE



## Boucle FOR

- <expr1> est évaluée une fois avant le passage de la boucle.  
Elle est utilisée pour initialiser les données de la boucle
- <expr2> est évaluée avant chaque passage de la boucle.  
Elle est utilisée pour décider si la boucle est répétée ou non
- <expr3> est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle
- Le plus souvent, **for** est utilisé comme boucle de comptage :  
`for ( <init.> ; <cond. répétition> ; <compteur> ) <bloc>`

# LA STRUCTURE REPETITIVE

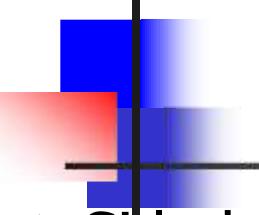
## EXEMPLES

### *Exemple1*

```
int I;  
for (I=0 ; I<=20 ; I++)  
    printf("Le carré de %d est %d \n", I, I*I);
```

### *Exemple2*

```
int n, total;  
for (total=0, n=1 ; n<101 ; n++)  
    total+=n;  
printf("La somme des nombres de 1 à 100 est %d\n",  
total);
```



# CHOIX DE LA STRUCTURE REPETITIVE

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do – while**
- Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez **while**.

# Les instructions de rupture de séquences

## L'instruction break

L'instruction **break** provoque le passage à l'instruction qui suit immédiatement le corps de la boucle *while, do-while, for ou switch*.

```
#include <stdio.h>
void main()
{ int i = -1;
  do { printf("La valeur du compteur vaut : %d\n", ++i);
        if(i == 10) break;
    }while(1);
```

```
Printf("je suis en dehors de la boucle while \n ");
```

```
}
```

➤ L'instruction break ne peut être utilisée que dans le corps d'une boucle ou d'un switch.

➤ L'action de break ne s'applique qu'à la boucle la plus intérieure dans laquelle elle se trouve. Elle ne permet pas de sortir de plusieurs boucles imbriquées.

# Les instructions de rupture de séquences

## L'instruction continue

- Elle relance immédiatement la boucle while, do, for, dans laquelle elle se trouve. C'est une instruction de branchement automatique que l'on utilise au sein d'une boucle.
- Si on exécute l'instruction "continue" au sein d'une boucle "do" ou "while" alors la prochaine itération recommence au niveau de l'évaluation de l'expression de la condition d'arrêt.

# Les instructions de rupture de séquences

## L'instruction continue

### Exemple avec "do "

```
i = 2;  
do  
{  
    i++;  
    if (i==5)  
        continue;  
    printf("\n i = %d",i);  
} while (i<7);
```

Cet exemple affichera :

```
i = 3  
i = 4  
i = 6  
i = 7
```

Si on exécute l'instruction "continue" au sein d'une boucle "do" ou "while" alors **la prochaine itération recommence au niveau de l'évaluation de l'expression de la condition d'arrêt**

Quand "i" vaut 5, le code exécute l'instruction "continue" qui "saute" le "printf" et va se brancher au niveau du test "while (i<7)" pour évaluer la condition d'arrêt.

# Les instructions de rupture de séquences

## L'instruction continue

### Exemple avec "while "

```
#include <stdio.h>
int main(){
    int i,j,x;
    x = 1;
    i = 5;
    j = 0;
    // Comme on utilise « i -- »,
    // on compare d'abord i et 0 puis i=i - 1.
    while ( (i--) > 0 )
    {
        x += 1;
        if ( x % 2 )
            continue;
        j += x * x;// j=j+x*x;
    }
    printf("\n j = %d\n",j);
}
```

Cet exemple affichera en sortie  $j = 56$   
La boucle s'effectue pour «  $i = 5,4,3,2,1$  »  
et «  $x$  » vaut respectivement 2, 3, 4, 5, 6  
avant le «  $if ( x \% 2 )$  ».

Si «  $x \% 2$  vaut 0 » alors «  $x$  » est pair et  
on fait l'instruction «  $j += x * x;$  ».

Si la valeur de l'expression «  $x \% 2$  » est  
différente de 0 ( $x$  est impair) alors le code  
exécute le «  $continue$  ».

L'instruction «  $j += x * x;$  » est « sautée »  
et on calcule «  $(i--) > 0$  ».

Ce code C effectue la somme  
«  $2*2+4*4+6*6 = 4+16+36=56$  ».

# Les instructions de rupture de séquences

## L'instruction continue

➤ Si on exécute l'instruction "continue" au sein d'une boucle  
"for (partie1; partie2 ; partie3)"  
alors toutes les instructions de la boucle qui suivent le "continue"  
sont "sautées", puis "partie3" est exécutée, puis "partie2 "



ce qui permet de déterminer si la boucle doit recommencer.

# Les instructions de rupture de séquences

## L'instruction continue

### Exemple avec « for »

```
for (i=0; i<10;i++)
{
    if ((i%3)==0) continue;
    printf("\ni = %d ",i);
}
printf("\nSortie : ");
printf("\n\ti = %d ",i);
```

Ce morceau de code C affiche :

```
i = 1
i = 2
i = 4
i = 5
i = 7
i = 8
```

Sortie:

```
i = 10
```

Si on exécute l'instruction "continue" au sein d'une boucle "for (partie1; partie2 ; partie3)" alors toutes les instructions de la boucle qui suivent le "continue" sont "sautées", puis "partie3" est exécutée, puis "partie2 "

La boucle for s'effectue pour  
« i = 0,1,2,3,4,5,6,7,8,9 »

Si « i » est divisible par 3, i.e. « (i%3)==0 », alors on effectue l'instruction « continue », ce qui va directement effectuer l'instruction « i++ » qui correspond à « partie3 », puis le test « i<10 » qui correspond à « partie2 » est effectué.

Cette boucle n'affiche pas les multiples de 3

# Les instructions de rupture de séquences

## L'instruction goto

Cette instruction permet de se brancher (inconditionnellement) à une étiquette (identificateur) à l'intérieur de la même fonction.

Sa syntaxe est la suivante : **goto étiquette ;**

Et la déclaration d'une étiquette se fait de la manière suivante :

**étiquette :** instruction ;

**Exemple :**

```
void main() { int i,j,k;
    for(i=1 ;i<=5 ;i++)
    for(j=1;j<=5;j++)
    for(k=1;k<=5;k++) {
        printf("i = %d; j = %d; k = %d",i,j,k);
        if (i*j*k== 40) goto sortie; }
sortie : printf("Fin du programme") ;
}
```

Il est déconseillé de l'utiliser systématiquement, elle n'est vraiment utile que dans des cas très extrêmes.



# Chapitre 2

- ✓ **LES TABLEAUX**
- ✓ **POINTEURS**

# LES TABLEAUX

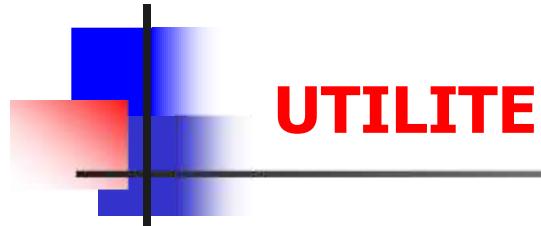
## UTILITE

- Imaginons que, dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne) ; actuellement, compte tenu des moyens de programmation dont nous disposons :
- il faut déclarer 13 variables  $N_1, N_2, \dots N_{12}$  et moy. A chaque variable  $N_i$ , nous affecterons la  $j^{\text{ème}}$  note et la variable moy contiendra la valeur :  $\sum N_i / 12$



Problème si nous disposons d'un grand nombre de notes !

# **LES TABLEAUX**



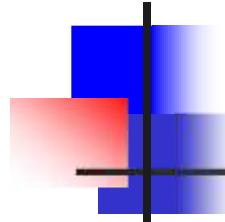
**UTILITE**

La programmation permet de rassembler toutes ces variables en une seule variable au sein de laquelle chaque valeur sera désignée par un numéro



notion de tableau

# **LES TABLEAUX**

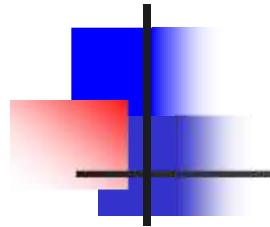


## **LE CONCEPT DU TABLEAU**

Un tableau est un ensemble de données qui :

- sont toutes du même type
- possèdent un identificateur unique (le nom du tableau)
- se différencient les unes des autres, dans ce tableau, par leur numéro d'indice.
- Les tableaux les plus fréquemment utilisées sont à une dimension (listes, vecteurs) ou à deux dimensions (matrices)
- D'un point de vue pratique, on représente un tableau par un ensemble de cases repérées par leurs indices (leurs positions dans le tableau) :

# LES TABLEAUX



**CONCEPT**

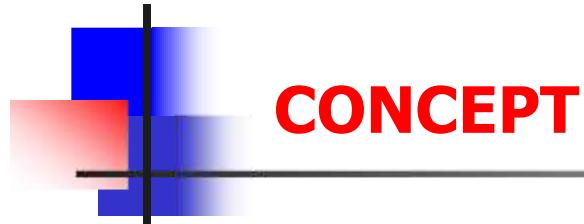
|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Tableau à deux dimensions

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|--|--|--|--|--|

Tableau à une dimension

# LES TABLEAUX



## CONCEPT

Pour évoquer le contenu d'une de ces cases, on utilise le nom du tableau suivi des coordonnées de la case dans le tableau.

- $A[i,j]$  : contenu de la case située à la  $i^{\text{ème}}$  ligne et à la  $j^{\text{ème}}$  colonne du tableau à deux dimensions nommé A
- $A[i]$  : contenu de la  $i^{\text{ème}}$  case du tableau unidimensionnel A

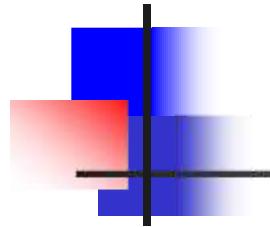
# LES TABLEAUX



## DECLARATION

- Avant d'utiliser un tableau, il faut faire sa déclaration, et ce quelque soit le langage de programmation.
- Quatre éléments fondamentaux définissent un tableau :
  - ◆ Son nom qui sera un identificateur choisi en respectant les règles usuelles de dénomination des variables
  - ◆ Le nombre de ses dimensions
  - ◆ Sa taille
  - ◆ Le type de données qu'il contient

# **LES TABLEAUX**



## **DECLARATION**

Pour caractériser entièrement un tableau dans un algorithme, on écrira :

tableau T[10] : entier  $\Rightarrow$  déclare un tableau de 10 entiers

tableau Tab[10,20] : reel  $\Rightarrow$  déclare un tableau de 10 lignes et 20 colonnes de réels

# LES TABLEAUX



## EXEMPLE

Pour i variant de 1 à n

    Pour j variant de 1 à m

        Lire(T[i,j])

    Fin-pour

Fin-pour

# LES TABLEAUX

## EXEMPLE

```
Variables    i:entier
              som,Moy : reel
              Tableau Note[10] : reel
```

Debut

Pour i variant de 1 à 10

Lire (Note[i])

Fin-pour

som  $\leftarrow$  0

pour i variant de 1 à 10

som  $\leftarrow$  som+Note[i]

fin-pour

moy  $\leftarrow$  som/10

Fin

# LES TABLEAUX



## REMARQUES

La valeur d'un indice doit toujours être égale au moins à 1 ou 0 (en C)

- Elle doit être un nombre entier
- Elle doit être inférieure ou égale au nombre d'éléments du tableau
- Il n'y a aucun rapport entre l'indice i et le contenu de la  $i^{\text{ème}}$  case

# TABLEAUX UNI-DIMENSIONNELS EN C

## DECLARATION

➤ En C, les tableaux à une dimension se déclarent selon la syntaxe :

**<type> <identificateur>[n]**

où : **n** et **type** désignent respectivement le nombre de cases et le type des données du tableau

➤ L'indice qui permet de localiser le contenu d'une case varie de 0 à n-1

# TABLEAUX UNI-DIMENSIONNELS EN C

## DECLARATION ET INITIALISATION

```
int MOIS[12]={31,28,31,30,31,30,31,31,30,31,30,31}
```

- définit un tableau de type int et de dimension 12
- Les 12 composantes sont initialisées par les valeurs respectives : 31, 28, 31, ... , 31.
- MOIS[0] désigne le contenu de la 1<sup>ère</sup> case du tableau MOIS
- MOIS[11] désigne la dernière composante du tableau MOIS
- MOIS[i] désigne la (i+1)<sup>ème</sup> composante du tableau MOIS

# TABLEAUX UNI-DIMENSIONNELS EN C

## REMARQUES

- le nombre de valeurs dans la liste doit correspondre à la taille du tableau.
- Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro.
- Si la taille n'est pas indiquée explicitement lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.
- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau
- Les valeurs entre accolades doivent être des constantes • Il doit y avoir au moins une valeur entre accolades

# TABLEAUX UNI-DIMENSIONNELS EN C

## EXEMPLES

```
int Tab1[5]={10, 20, 13, 4, 5};  
float Tab2[5]={1.5,6.75};  
int Tab3[]={10, 2 , 75};  
int Tab4[3]={10,20,30,40,50}; /*Erreur ! */
```

Variables

I : entier

tableau A[5] : entier

debut

    pour I variant de 1 à 5

        lire A[I]

    fin-pour

Fin

```
#include <stdio.h>  
void main() {  
    int A[5];  
    int I; /* Compteur */  
    for (I=0; I<5; I++)  
        scanf("%d", &A[I]);  
}
```

# TABLEAUX UNI-DIMENSIONNELS EN C

## EXAMPLE

Variables

I : entier

tableau A[5] :entier

debut

pour I variant de 1 à 5

Ecrire (A[I])

fin-pour

fin

```
#include <stdio.h>
void main() {
    int A[5];
    int I; /* Compteur */
    for (I=0; I<5; I++)
        {printf("%d", A[I]);
         printf("\n");}
}
```

# TABLEAUX UNI-DIMENSIONNELS EN C

## EXEMPLE

```
#define NB_ELEVES 30
void main()
{
float notes[NB_ELEVES], som, moy;
int i, nbre;
printf("Donnez vos %d notes : \n\n", NB_ELEVES);
for(i=0, som=0; i<NB_ELEVES; i++)
{ printf("Note N°%d : ",i+1);
scanf("%f",&notes[i]);
som += notes[i];
}
for(nbre=0, moy=som/NB_ELEVES, i=0; i<NB_ELEVES; i++)
if(notes[i] < moy) nbre++;
printf("La moyenne de la classe est %f\n",moy);
printf("%d élèves ont moins de cette moyenne",nbre);
}
```

# TABLEAUX UNI-DIMENSIONNELS EN C

## Opérateurs de post et pré incrémentation

Soit T un tableau

### Post incrémentation

$$T[i++] = \text{exp;} \longleftrightarrow T[i] = \text{exp;} i = i+1;$$

### Pré incrémentation

$$T[++i] = \text{exp;} \longleftrightarrow i = i+1; T[i]=\text{exp};$$

### Post décrémentation

$$T[i--) = \text{exp;} \longleftrightarrow T[i] = \text{exp;} i = i-1;$$

### Pré décrémentation

$$T[--i] = \text{exp;} \longleftrightarrow i = i-1; T[i]=\text{exp};$$

# TABLEAUX UNI-DIMENSIONNELS EN C

## RECHERCHE SÉQUENTIELLE

Pour chercher une donnée ou éventuellement sa position dans un tableau non trié, la recherche doit se faire de manière séquentielle, c'est à dire en comparant la donnée avec les données successives de tout le tableau .

```
i=0; while( i<n && Tab[i]!=donnee) i++;
/* Sortie i == n ou (i <n et Tab[i] == donnee) */
```

```
for(i=0; i<n && Tab[i]!=donnee; i++);
/* Sortie i == n ou (i <n et Tab[i] == donnee) */
```

```
for(i=0, trouve=0; i<n && trouve == 0; i++)
    if(Tab[i]==donnee)  trouve = 1;
/* Sortie i == n ou trouve= = 1*/
```

# TABLEAUX UNI-DIMENSIONNELS EN C

## EXEMPLE

```
#define Nmax 100
void main()
{
    int tab[Nmax] , n, i, valeur;
    /* Saisie des données */
    do
    { printf("Donnez le nombre
            d'éléments ");
        scanf("%d",&n);
    }while(n<=0 || n>=Nmax);

    for(i=0; i<n; i++)
    {
        printf("tab[%d] = ",i);
        scanf("%d",&tab[i]);
    }
    printf("Donnez la valeur à rechercher : ");
    scanf("%d",&valeur);

    /* Recherche séquentielle */
    for(i=0; i<n && tab[i] != valeur ; i++ ) ;

    /* Affichage du résultat */
    if(i == n)
        printf("La valeur recherchée est inexistante");
    else
        printf("La valeur recherchée se trouve à la
                position %d",i);
}
```

# TABLEAUX UNI-DIMENSIONNELS EN C

## EXEMPLE DE TRI

Le *tri par extraction* (ou *tri par sélection*) consiste à parcourir le tableau une première fois pour trouver le plus petit élément (ou le plus grand élément, suivant le type de tri), ensuite mettre cet élément au début (par une permutation), puis parcourir une seconde fois le tableau (de la 2ème au dernier élément) pour trouver le second plus petit élément (le second plus grand élément), le placer en 2ème position, et ainsi de suite...

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION

En C, le tableau à deux dimensions se déclare selon la syntaxe :

**<type> <identificateur>[nblignes][nbcColonnes]**

➤ **nblignes**, **nbcColonnes** et **type** désignent respectivement le nombre de lignes, colonnes et le type des données du tableau

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION

- Les indices du tableau varient de 0 à 'nblignes-1' et de 0 à 'nb-colonnes-1'
- La composante de la  $i^{\text{ème}}$  ligne et la  $j^{\text{ème}}$  colonne du tableau A est notée :  $A[i-1][j-1]$
- Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION ET INITIALISATION

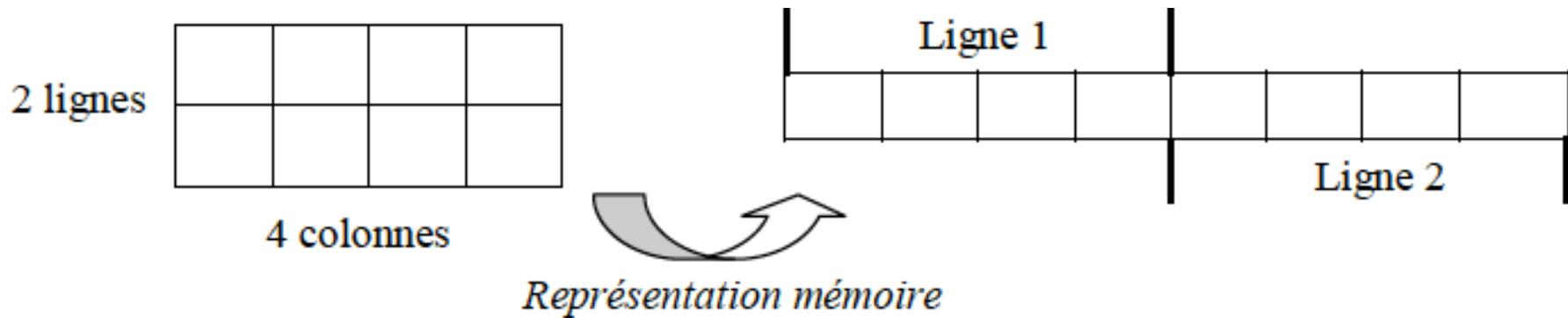
```
int A[3][10] ={{ 0,10,20,30,40,50,60,70,80,90},  
                {10,11,12,13,14,15,16,17,18,19},  
                {21,22,23,24,25,26,27,28,29,30}};
```

- Définit un tableau à 3 lignes et 10 colonnes
- On peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades. A l'intérieur de la liste, les composantes de chaque ligne du tableau sont encore une fois comprises entre accolades

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION ET INITIALISATION

- Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite.



- On peut ne pas indiquer toutes les valeurs, celles qui manquent seront initialisées par zéro.
- Il est défendu d'indiquer trop de valeurs pour un tableau.

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION ET INITIALISATION

```
int MatA[2][4] = { {1, 2, 3, 4},  
                   {5, 6, 7, 8} } ;
```

```
int MatA[][4]= { {1, 2, 3, 4},  
                  {5, 6, 7, 8} } ;
```

```
int MatA[2][4] = {1, 2, 3, 4, 5, 6, 7, 8} ;
```

```
int MatB[3][3] = { {1},{0,1},{0,0,1} } ;
```

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

MatA

|   |   |   |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

MatB

# TABLEAUX A DEUX DIMENSIONS EN C

## DECLARATION ET INITIALISATION

```
int MatC[][][4] = {1, 2, 3, 4, 5, 6};
```

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 0 | 0 |

MatC

```
int MatD[3][3] = {{1, 2, 3}, {4}} ;
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 0 | 0 |
| 0 | 0 | 0 |

MatD

```
Mat[0][0] = 1 ;
```

```
Mat[0][1] = 2 ;
```

```
Mat[1][0] = 3 ;
```

```
Mat[1][1] = 4 ;
```

|   |   |
|---|---|
| 1 | 2 |
| 3 | 4 |

Mat

# TABLEAUX A DEUX DIMENSIONS EN C

## EXEMPLE DE SAISIE

Variables :

tableau A[5,10] : entier

I,J : entier

Debut

pour I variant de 1 à 5

pour J variant de 1 à 10

lire A[I,J]

fin-pour

fin-pour

fin

```
#include <stdio.h>
void main(){
    int A[5][10];
    int I,J;
    for (I=0; I<5; I++)
        for (J=0; J<10; J++)
            scanf("%d", &A[I][J]);
}
```

# TABLEAUX A DEUX DIMENSIONS EN C

## EXEMPLE D'AFFICHAGE

tableau A[5,10] :entier

I,J : entier

debut

pour I variant de 1 à 5

    pour J variant de 1 à 10

        écrire A[I,J]

    finpour

Finpour

fin

```
#include <stdio.h>
void main(){
    int A[2][2];
    int I,J;
    for (I=0; I<2; I++)
        { for (J=0; J<2; J++)
            printf("%d   ", A[I][J]);
            printf("\n");
        }
}
```

# TABLEAUX A DEUX DIMENSIONS EN C

## EXEMPLE

Le programme suivant demande à l'utilisateur d'introduire les notes des élèves pour chaque matière, puis calcule la moyenne de chaque élève et la moyenne de la classe par matière.

La note [i][j] représente la note de l'élève i de la matière j.

```
#define LMAX 10
#define CMAX 10
void main()
{
float notes[LMAX][CMAX],som;
int i,j,ligne,colonne;
do
{
printf("Nombre d'élèves : ");
scanf("%d",&ligne);
}while(ligne<1|| ligne>LMAX);
do
{
printf("Nombre de matières : ");
scanf("%d",&colonne);
}while(colonne<1 || colonne>CMAX);
```

# TABLEAUX A DEUX DIMENSIONS EN C

## EXEMPLE

```
for (i=0;i<ligne;i++)
{
printf("\nNotes pour l'élève %d =\n",i+1);
for (j=0;j<colonne;j++) {
printf("\tNote de la matière %d = ",j+1);
scanf("%f",&notes[i][j]);}
}

for (i=0;i<ligne;i++)
{ for (som=0,j=0;j<colonne;j++) som += notes[i][j];
printf("\nLa moyenne de l'élève %d : %f", i+1, som/colonne);
}
for (j=0;j<colonne;j++) {
    for (som=0,i=0;i<ligne;i++) som += notes[i][j];
    printf("\nLa moyenne de la matière %d : %f", j+1, som/ligne);}
}
```

# POINTEURS



## RAPPEL

- Une variable est destinée à contenir une valeur du type avec lequel est déclarée. Physiquement, cette valeur se situe en une mémoire

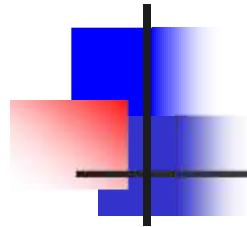
`int x` → réserve un emplacement pour un entier en mémoire

`x=10` → écrit la valeur 10 dans l'emplacement réservé

- Cette valeur est située à l'emplacement `&x` (adresse de `x`) dans la mémoire

- Pour obtenir l'adresse d'une variable on fait précéder son nom avec l'opérateur `&`

# POINTEURS



## ADRESSAGE DES VARIABLES

Deux modes d'adresses principaux :

➤ Adressage direct : accès au contenu d'une variable par le nom de la variable

```
int a ; a=10 ;
```

➤ Adressage indirect : Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable

```
P=&A ;
```

# POINTEURS



## RAPPEL

Lorsqu'on déclare une variable et ce quelque soit le langage, le compilateur réserve la place suffisante à l'hébergement du contenu de cette variable

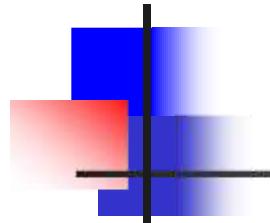
# POINTEURS



## DEFINITION

- Un pointeur est une variable, elle est destinée à contenir une adresse mémoire i.e. une valeur identifiant un emplacement en mémoire
- C'est une référence sur une donnée. Il s'agit d'une adresse en mémoire où est stockée la donnée référencée

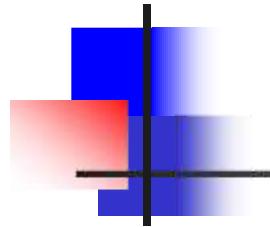
# POINTEURS



## DEFINITION

- Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom par le signe '\*'
- L'opérateur '\*' désigne le contenu de l'adresse pointée
- L'opérateur & est un opérateur unaire qui fournit comme résultat l'adresse de son opérande
- Si un pointeur p contient l'adresse d'une variable A, on dit que 'p pointe sur A'

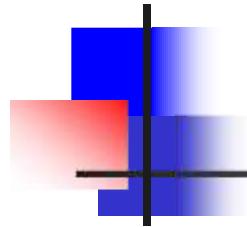
# POINTEURS



## DECLARATION

- <type> \*<nom-p>
- La déclaration int \*p peut être interprétée comme suit :
  - \*p est de type int
  - ou* p pointeur sur int
  - ou* p peut contenir l'adresse d'une variable de type int

# POINTEURS



## PONTEURS ET TABLEAUX

- Le nom d'un tableau tab représente l'adresse de son 1<sup>er</sup> élément i.e. tab[0]
- Tab+i : représente l'adresse du (i+1)<sup>ème</sup> élément du tableau tab

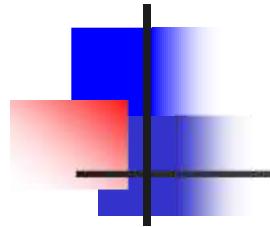
Exemple :

```
int tab[10]={5,8,4,3,9,6,5,4,3,8}
```

```
printf(« %d »,tab[0]) affiche 5
```

```
printf(« %d »,*tab) affiche 5
```

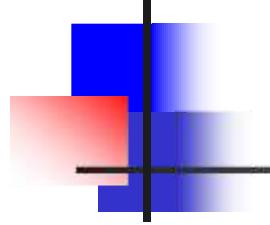
# POINTEURS



## ARITHMETIQUE DES POINTEURS

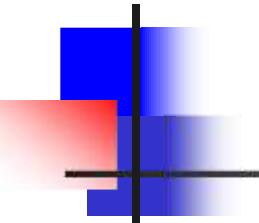
- $P+i$  = adresse de  $p + i * \text{taille}(\text{élément pointé par } p)$   
Si  $p$  est un pointeur d'entier,  $p+1$  est donc le pointeur sur l'entier qui suit immédiatement celui pointé par  $p$
- Il faut retenir que l'entier qu'on additionne au pointeur est multiplié par la taille de l'élément pointé pour obtenir la nouvelle adresse
- La taille est donnée par l'opérateur : `sizeof(type expression)`

# POINTEURS



## POINTEUR NULL

La constante NULL, définie dans <stdlib.h>, est la valeur standard pour un pointeur vide.



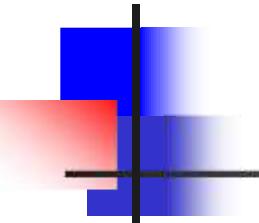
## EXEMPLES

### Exemple1

```
int x = 10, *ptr;  
ptr = &x; /* affecte l'adresse de la variable x au pointeur ptr */  
printf("Adresse de x : %p\n",&x) ;  
printf("Valeur de ptr : %p\n",ptr);
```

### Exemple2

```
int *ptr,  
int i;  
i = 10;  
ptr = &i;  
printf("la valeur de i avant : %d\n",i);/*affiche :la valeur de i avant */  
*ptr = 20; /*la variable pointée par ptr reçoit 20, autrement dit *ptr  
désigne le contenu de i*/  
printf("la valeur de i après:%d\n",i);/*affiche : la valeur de i après */
```



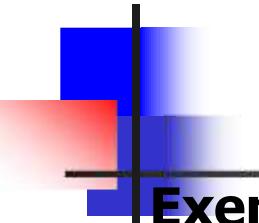
## EXEMPLES

### Exemple3

```
int i = 10 ;
int *ptr = &i;
printf("Valeur de ptr avant l'addition : %p\n",ptr);
ptr = ptr + 2 ;
printf("Valeur de ptr après l'addition : %p\n",ptr);
```

### Exemple4

```
int i = 10 ;
int *ptr = &i;
printf("Valeur de ptr avant incrémentation : %p\n",ptr);
ptr++ ;
printf("Valeur de ptr après incrémentation : %p\n",ptr);
```



## EXAMPLE

### Exemple 5

Qu'affiche le programme suivant :

```
void main()
{
    int A = 10 , B = 20 , C , D;
    int *ptr1=NULL, *ptr2=NULL ;
    ptr1 = &A;
    ptr2 = &B;
    C = *ptr1 + *ptr2;
    ptr1 = &C;
    ++*ptr1;
    D = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = D;
    printf("A = %d \nB = %d \nC = %d \nD = %d", A, B, C, D);
}
```



# Chapitre 3

- ✓ LES CHAINES DE CARACTERES
- ✓ LES FONCTIONS
- ✓ TYPES DE DONNEES EVOLUES
- ✓ L' ALLOCATION DYNAMIQUE

# LES CHAINES DE CARACTERES

## LE TYPE CARACTÈRE

- Une chaîne de caractères est une suite de caractères. La structure des chaînes est comparable à celle des tableaux : tableau de caractères.
- Chaque caractère d'une chaîne nommée X est accessible comme peut l'être un élément quelconque d'un tableau par :  
 $X[i]$  où i est le rang du caractère dans la chaîne.

# **LES CHAINES DE CARACTERES**

## **LE TYPE CARACTÈRE**

➤ Les langages de programmation disposent de fonctions spécifiques permettant d'effectuer des traitements sur l'ensemble de la chaîne :

Déclaration et initialisation d'une chaîne

Concaténation de deux chaînes

Longueur d'une chaîne

Comparaison de deux chaînes

.....

# LES CHAINES DE CARACTÈRES

## DÉCLARATION

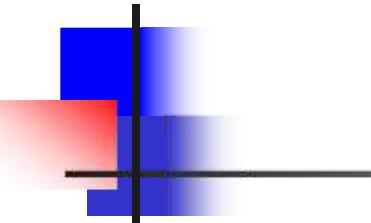
☞ Une chaîne de caractères se déclare comme suit :

char <NomVariable> [<Longueur>];

*Exemple* : char NOM [20];

☞ Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne.

# LES CHAINES DE CARACTERES

- 
- La représentation interne d'une chaîne de caractères est terminée par le symbole '\0' (NUL). Ainsi, pour un texte de **n** caractères, il faut prévoir **n+1** octets.
  - Le nom d'une chaîne est le représentant de ***l'adresse du premier caractère*** de la chaîne.

*Exemple: Mémorisation d'un tableau*

```
char TXT[10] = "BONJOUR !";
```

- Les chaînes de caractères constantes sont indiquées entre guillemets.  
La chaîne de caractères vide est alors: ""

# LES CHAINES DE CARACTÈRES

## Initialisation des chaînes de caractères

- En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades:

```
char CHAINE[] = {'H','e','l','l','o','\0'};
```

Pour les chaînes de caractères, il suffit de faire

```
char CHAINE[] = "Hello";
```

- Lors de l'initialisation par [], l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne, c.-à-d.: le nombre de caractères + 1

# **LES CHAINES DE CARACTÈRES**

## **Accès aux éléments d'une chaîne**

L'accès à un élément d'une chaîne de caractères

A se fait comme dans les tableaux par :

$A[i]$  où i est l'indice

# LES CHAINES DE CARACTÈRES

## Fonctions d'écriture des chaînes de caractères

***Fonctions disponibles dans <stdio.h>***

- ☛ **printf("%s",<chaîne>)** permet d'afficher <chaîne>
- ☛ **puts(<chaîne>)** affiche <chaîne> et retourne à la ligne.

# LES CHAINES DE CARACTÈRES

## Fonctions de lecture

### *Fonctions disponibles dans <stdio.h>*

- **gets(<chaîne>)** lit une chaîne de caractères
- **scanf ("%s",x)** permet de saisir la chaîne x

#### Exemple

```
char LIEU[25];
int JOUR, MOIS, ANNEE;
printf("Entrez lieu et date de naissance : \n");
scanf("%s %d %d %d", LIEU, &JOUR, &MOIS, &ANNEE);
```

# LES CHAINES DE CARACTÈRES

## Fonctions définies sur les chaînes de caractères

- La fonction **scanf** a besoin des adresses de ses arguments
- Les noms des variables numériques (**int**, **char**, **float**, ...) doivent être marqués par le symbole '**&**'
- Comme le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne, il ne doit pas être précédé de l'opérateur '**&**' !
- La fonction **scanf** avec plusieurs arguments presuppose que l'utilisateur connaisse exactement le nombre et l'ordre des données à introduire!

# LES CHAINES DE CARACTÈRES

## Fonctions disponibles dans <string.h>

- **strupr(<s>)** convertit la chaîne en majuscules
- **strlen(<s>)** fournit la longueur de <s> sans compter le '\0'
- **strcpy(<s>, <t>)** copie <t> vers <s>
- **strcat(<s>, <t>)** ajoute <t> à la fin de <s>
- **strcmp(<s>, <t>)** : compare <s> et <t> et fournit un résultat :
  - \* négatif si <s> précède <t>
  - \* zéro si <s> est égal à <t>
  - \* positif si <s> suit <t>

# LES CHAINES DE CARACTÈRES

## Fonctions disponibles dans <string.h>

### ***Remarque :***

Comme le nom d'une chaîne de caractères représente une adresse fixe en mémoire, on ne peut pas faire cette affectation:

```
A = "Hello";
```

Il faut par exemple utiliser la fonction strcpy :

```
strcpy(A,"Hello");
```

# LES CHAINES DE CARACTÈRES

## Tableaux chaînes de caractères en langage C

### Déclaration, initialisation et mémorisation

- Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type char, où *chaque ligne contient une chaîne de caractères.*
- La déclaration char JOUR[7][9]; ⇒ réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs).

# **LES CHAINES DE CARACTÈRES**

## **Tableaux chaînes de caractères en langage C**

### **Déclaration, initialisation et mémorisation**

- Lors de la déclaration il est possible d'initialiser toutes les composantes du tableau par des chaînes de caractères constantes:

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi", "jeudi",
                  "vendredi", "samedi", "dimanche"};
```

- Les tableaux de chaînes sont mémorisés ligne par ligne

# LES CHAINES DE CARACTÈRES

## Tableaux chaînes de caractères en langage C

### Accès aux chaînes

Il est possible d'accéder aux différentes *chaînes de caractères* d'un tableau, en indiquant simplement la ligne correspondante.

### *Exemple*

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi","jeudi",
    "vendredi","samedi", "dimanche"};
```

```
int I = 2; printf("Aujourd'hui, c'est %s !\n", JOUR[I]);
```



Aujourd'hui, c'est mercredi !

# LES CHAINES DE CARACTÈRES

## Tableaux chaînes de caractères en langage C

### Affectation

L'attribution d'une chaîne de caractères à une composante d'un tableau de chaînes se fait en général à l'aide de la fonction **strcpy**

```
strcpy(JOUR[4], "Friday");
```



change le contenu de la 5<sup>e</sup>composante du tableau JOUR soit "vendredi" en "Friday".

# LES CHAINES DE CARACTÈRES

## Tableaux de chaînes de caractères en Langage C

### Accès aux caractères

```
for(I=0; I<7; I++) printf("%c ", JOUR[I][0]);
```



affiche les premières lettres des jours de la semaine:

I m m j v s d

# LES FONCTIONS

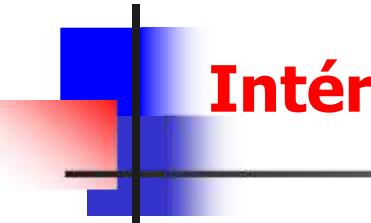
## Introduction

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes, fonctions ou procédures, plus simples et plus compacts que l'on pourra invoquer n'importe où dans le programme en faisant référence à leur nom



Modularité et simplification des problèmes

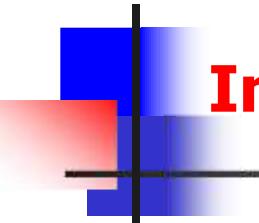
# LES FONCTIONS



## Intérêt de la modularité

- \* *Meilleure lisibilité*
- \* *Diminution du risque d'erreurs*
- \* *Possibilité de tests sélectifs*
- \* *Simplicité de l'entretien*
- \* *Favorisation du travail en équipe*
- \* *Hiérarchisation des modules*

# LES FONCTIONS



## Introduction

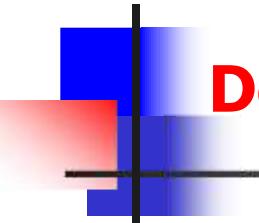
- Un module désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe.
- Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

# LES FONCTIONS

## Introduction

- Lorsque le module donne un paramètre en sortie on parlera de fonction sinon on parlera de procédure
- Il faut donc déclarer l'existence de ces procédures ou fonctions, leur donner un nom, définir leur type lorsqu'il s'agit de fonctions (i.e. le type du résultat renvoyé : entier, réel, etc...) et enfin le traitement qu'elles permettent de réaliser.

# LES FONCTIONS



## Définition d'une fonction

Dans la définition d'une fonction, on indique :

- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction
- les instructions à exécuter

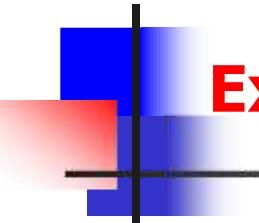
# LES FONCTIONS



## Définition

```
<TypeResultat> <NomFonct>(<TypePar1> <Nompar1>,.....)  
{  
    <déclarations locales>  
    <instructions>  
}
```

# LES FONCTIONS



## Exemple

---

```
int MAX(int A, int B)
```

```
{
```

```
    if (A > B)
```

```
        return A;
```

```
    else return B;
```

```
}
```

# LES FONCTIONS

## Remarques

- La procédure est une fonction de type void
- Le type par défaut d'une fonction est int
- On ne peut pas définir une fonction à l'intérieur d'une fonction
- Une fonction ne peut pas renvoyer comme résultat des tableaux

# LES FONCTIONS

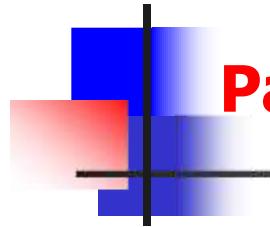
## Renvoi du Résultat

Une fonction renvoie un résultat avec : return <expression>

Ce qui implique :

- Évaluation de <expression>
- Conversion automatique du résultat de l'expression dans le type de la fonction
- Renvoi du résultat
- Fin de la fonction

# **LES FONCTIONS**



## **Paramètres d'une fonction**

Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction.

# LES FONCTIONS



## Déclaration d'une fonction

Il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur des types des paramètres et du résultat de la fonction. Si la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

# LES FONCTIONS

## Prototype d'une fonction

La déclaration d'une fonction se fait par **un prototype** de la fonction qui indique uniquement le type des données transmises et reçues par la fonction

<TypeRes> <NomFonct> (<TypePar1> Par1, .......);

# LES FONCTIONS



## Règles pour la déclaration des fonctions

### ***Déclaration locale :***

Une fonction peut être déclarée localement dans la fonction qui l'appelle (avant la déclaration des variables). Elle est alors disponible à cette fonction.

# LES FONCTIONS

## Règles pour la déclaration des fonctions

### ***Déclaration globale :***

Une fonction peut être déclarée globalement au début du programme (derrière les # include). Elle est alors disponible à toutes les fonctions du programme

# LES FONCTIONS



## Règles pour la déclaration des fonctions

### ***Déclaration implicite par la définition :***

La fonction est automatiquement disponible à toutes les fonctions qui suivent sa définition

# LES FONCTIONS

## Exemple de déclaration des fonctions

On se propose de calculer la surface d'un cercle

```
Void main()
```

```
{ float a=3.14;
```

```
float r;
```

```
Printf(<< saisir le rayon r : >>); scanf(<< %f >>,&r);
```

```
printf(<< %f >>,a*r*r);
```

```
}
```

# LES FONCTIONS

## Exemple de déclaration des fonctions

On utilise :

- \* la fonction Surface qui calcule la surface
- \* la fonction Pi qui donne la valeur 3,14

Main()

appelle

Surface

appelle

Pi

# LES FONCTIONS

## Exemple de déclaration locale

```
#include <stdio.h>
void main()
{ float Surface (float rayon);
  float r;
  printf(« saisir le rayon du cercle: »); scanf(« %f »,&r);
  printf(« la surface vaut : %f\n »,Surface(r));
}
float Surface(float rayon)
{ float Pi();
  return Pi()*rayon*rayon;
}
float Pi()
{ return 3.14;}
```

# LES FONCTIONS

## Exemple de déclaration implicite par la définition

```
#include <stdio.h>
float Pi()
{ return 3.14;}
```

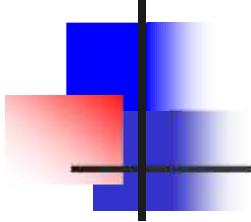
```
float Surface(float rayon)
{
    return Pi()*rayon*rayon;
}
void main()
{ float r;
    printf("saisir le rayon du cercle: "); scanf("%f",&r);
    printf("la surface vaut : %f\n ",Surface(r));
}
```

# LES FONCTIONS

## Exemple de déclaration globale

```
#include <stdio.h>
float Surface (float rayon);
float Pi();
void main()
{ float r;
    printf(« saisir le rayon du cercle: »); scanf(« %f »,&r);
    printf(« la surface vaut : %f\n »,Surface(r));
}
float Surface(float rayon)
{ return Pi()*rayon*rayon;}
float Pi()
{ return 3.14;}
```

# **LES FONCTIONS**



## **Variables et Fonctions**

### **Variables locales ou internes**

Les variables dites locales sont celles qui sont déclarées dans un bloc ({}). Elles ne sont visibles (donc utilisables) que dans ce bloc. Leur durée de vie va de l'exécution du début du bloc jusqu'à la fin du bloc (variables volatiles)

### **Variables globales ou externes**

Ce sont les variables déclarées hors de tout bloc. Elles sont visibles à partir de leur définition

# LES FONCTIONS

## EXEMPLE

```
#include <stdio.h>
float x ; /*variables globales*/
int N ;
float f1()
{
    int i ;

}
int main()
{int i;
    /* dans le main, x et N sont accessibles mais pas i de f1*/
}
```

# LES FONCTIONS

## Passage par valeur/adresse

Le passage des paramètres se fait par valeur : les fonctions n'obtiennent que les valeurs de leurs paramètres et n'ont pas accès aux variables elles-mêmes.

A l'intérieur de la fonction, on peut donc changer les valeurs des paramètres sans influencer les variables originales.

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

permuter (x,y)



x et y restent inchangés

# LES FONCTIONS

## Passage par valeur/adresse

Si au contraire, on désire modifier les valeurs des paramètres effectifs, on parlera de passage des paramètres par adresse. Ainsi, la fonction appelante doit fournir l'adresse de la variable et la fonction appelée doit déclarer les paramètres comme pointeurs

```
void permuter (int *a, int *b)
{ int aide;
  aide= *a; *a= *b; *b= aide;
}
```

permuter (&x,&y)



les contenus de x et y  
sont échangés

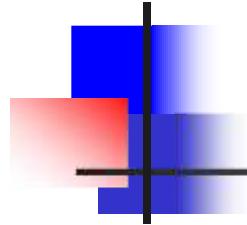
# LES FONCTIONS

## Passage par valeur/passage par adresse

```
#include <stdio.h>
void plus(int *a, int b)      void main()
{
    *a = *a+b
}
int c,d;
c=10;
d=3;
plus(&c,d);printf
}
```

- ⇒ on ajoute 3 à la valeur 10 et donc c contient  
après l'appel de plus  $10+3=13$

# FONCTION

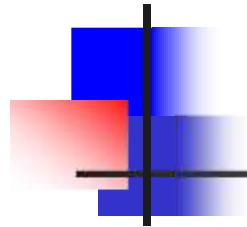


## Version Itérative

## Version Récursive

- La fonction s'appelle elle même  
Exemple : Le clacul de la factorielle
- Sauvegarde des variables locales et des paramètres  
utilisation de piles
- Point terminal ou point d'arrêt

# **TYPES DE DONNEES EVOLUES**

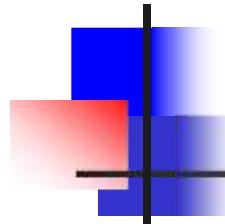


## **Enregistrement et structure**

➤ La notion d'enregistrement ou de structure permet de regrouper un ensemble de données de types différents correspondant à un même objet.

Exemple : nom, prénom et date de naissance  
d'une même personne

# **TYPES DE DONNEES EVOLUES**



## **Enregistrement et structures**

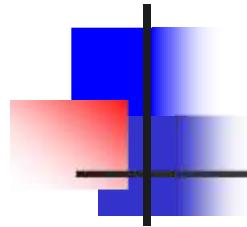
➤ Un enregistrement est un nouveau type défini par un identificateur et possédant, en son sein, un certain nombre de variables :

**<nom de l'enregistrement> : <nom du champ1> : type1  
                                  <nom du champ2> : type2  
                                  <nom du champ3> : type3**

➤ Une fois ce nouveau type défini, on peut déclarer des variables de ce type :

**<Nom de variable> : type <nom de la structure>**

# **TYPES DE DONNEES EVOLUES**



## **Enregistrement et Structure**

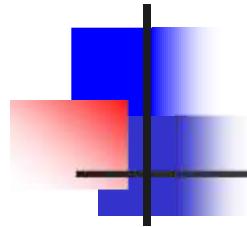
- Pour accéder à un champ de cette structure, on fait :  
<nom de variable>.<nom du champ>

### *Traduction en C:*

En C, ce type se définit sous la syntaxe suivante :

```
struct personne {  
    char nom[20] ;  
    char prenom[20] ;  
};  
Struct personne ami;
```

# **TYPES DE DONNEES EVOLUES**



## **Tableau d' Enregistrements**

Un tableau d'enregistrements est un tableau dont les composantes sont des enregistrements.

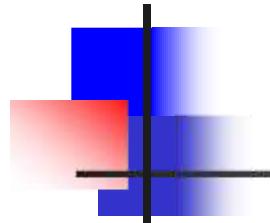
Déclaration :

<nom de la structure> : <nom du champ1> : type1  
                                  <nom du champ2> : type2  
                                  <nom du champ3> : type3

tableau t[10] : <nom de la structure>

Pour accéder à un champ donné d'une structure :  
<nom du tableau>[indice].<nom du champ>

# **TYPES DE DONNEES EVOLUES**



## **Tableau d' Enregistrements**

### **Exemple :**

```
struct agenda {  
    char nom[20] ;  
  
    int telephone ;  
  
} ;  
  
struct agenda coordonnee[10];
```

# L' ALLOCATION DYNAMIQUE



## ALLOCATION

- L'allocation dynamique permet la réservation d'un espace mémoire. Ceci est à mettre en opposition avec l'allocation statique de mémoire. En effet, dans ce type d'allocation, la mémoire est réservée dès le début de l'exécution d'un bloc.
- Lorsque l'on souhaite utiliser des informations en grand nombre, la meilleure structure de données que nous ayons vu jusqu'à présent, reste le tableau.

# L' ALLOCATION DYNAMIQUE



## ALLOCATION

- Toutefois, la déclaration préalable de la taille de ce tableau demeure une contrainte, notamment lorsqu'il est impossible, à priori, de connaître le nombre d'éléments qui seront nécessaires.
  
- Le concept d'allocation dynamique de mémoire permet de lever cette contrainte : chaque fois que l'on a besoin d'enregistrer en mémoire une information, on demande au compilateur de nous allouer l'espace nécessaire en mémoire.

# L' ALLOCATION DYNAMIQUE



## ALLOCATION

- Imaginons par exemple que nous voulions enregistrer une liste de noms en utilisant ce principe d'allocation dynamique de mémoire : Le seul moyen de ne pas perdre les informations enregistrées consiste à les chaîner.
  
- Pour ce faire, on crée une structure contenant deux variables : Le nom à enregistrer et une variable de type pointeur qui pointera, en mémoire, vers l'emplacement où se trouve le nom suivant.

# **ALLOCATION DYNAMIQUE**

## **ALLOCATION**

Une telle organisation de l'information permet d'ajouter des informations autant que nécessaire, de supprimer ou d'en insérer, sans avoir besoin de connaître le nombre d'éléments de la liste

# ALLOCATION DYNAMIQUE

## ALLOCATION DE MÉMOIRE EN LANGAGE C

- Pour réclamer l'allocation d'un espace vide en mémoire, il faut, au préalable déclarer l'existence d'un pointeur vers l'objet à mémoriser puis lui affecter le résultat de la fonction **malloc**.
- **nouveau = malloc(sizeof(liste))** : sélectionne une adresse en mémoire et un espace suffisant pour y ranger une structure de type liste. L'adresse est alors rangée dans le pointeur nouveau.

# ALLOCATION DYNAMIQUE

## ALLOCATION DE MÉMOIRE EN LANGAGE C

- La fonction **malloc** est dans **<stdlib.h>**.
- Pour accéder à l'information de la structure pointée par le pointeur nouveau, on fait : **nouveau→nom**
- Le dernier pointeur doit donc pointer vers un emplacement particulier appelé **NULL**

# ALLOCATION DYNAMIQUE

## LIBERATION DE MÉMOIRE EN LANGAGE C

**Free(p)** : permet de libérer la mémoire allouée via  
malloc()



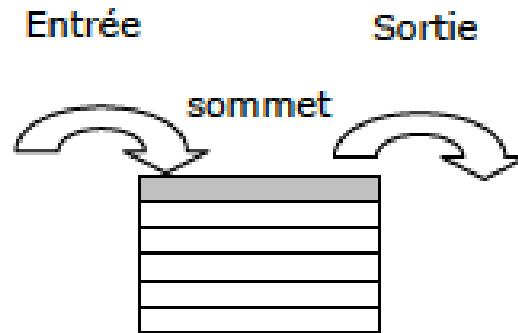
## Chapitre 4

- ✓ LES PILES
- ✓ LES FILES
- ✓ LES ARBRES

# LES PILES

## INTRODUCTION

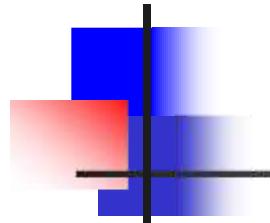
- Une pile (stack en anglais) est une liste dans laquelle l'insertion ou la suppression se fait toujours à partir de la même extrémité appelée sommet de la pile.



- Une pile permet de modéliser un ensemble régi par la discipline 'dernier arrivé premier sorti'

➔ LIFO : Last In First Out

# LES PILES



## CARACTERISTIQUES GENERALES

- Ensemble de données dans lequel on a la possibilité de demander :
  - \* ensemble vide ?
  - \* éventuellement ensemble plein ?
- On peut faire dans cet ensemble :
  - \* Ajouter un élément (empiler)
  - \* Retirer le 1er élément (dépiler)
  - \* Donner l'élément qui se trouve au sommet
  - \* Eventuellement, vider la pile

# **LES PILES**

## **EXEMPLES D'UTILISATION**

- Pile courante
- Pile informatique

Seules actions possibles :

L'ajout d'un élément au sommet

Le retrait d'un élément au sommet

# LES PILES

## EXEMPLES D'UTILISATION

Opérations effectuées sur les piles :

vider( $p$ ) : vide le contenu de la pile  $P$

Premier( $P$ ) : retourne (sans le dépiler) l'élément au sommet de  $P$

Dépiler( $P$ ) : supprime physiquement l'élément au sommet de  $P$

Empiler( $x, P$ ) : insère l'élément  $x$  au sommet de  $P$

Pile-vide( $P$ ) : fonction qui teste si  $P$  est vide

Pile\_pleine( $P$ ): teste si la pile  $P$  est pleine ou non

# LES PILES

## IMPLEMENTATIONS

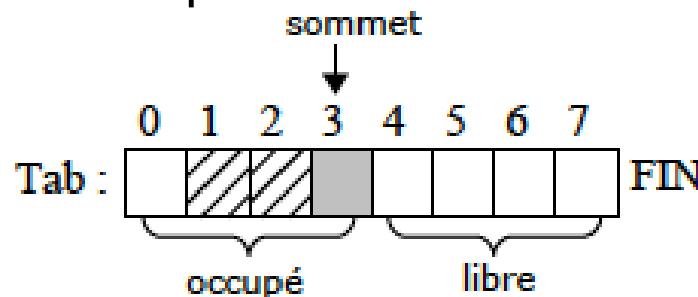
### Implémentation dynamique par une liste chaînée

Une pile en dynamique n'a pas de taille limite, hormis celle de la mémoire centrale (RAM) disponible.

```
struct element { int val;  
                struct element *suiv;} ;  
typedef struct element *pile;
```

### Implémentation statique sous forme de tableau

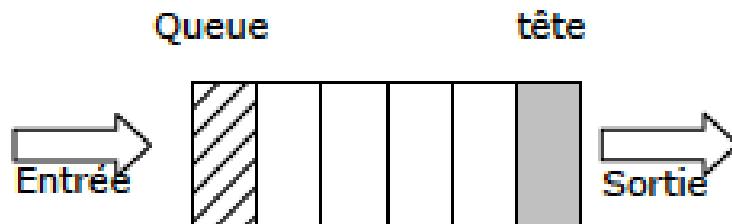
```
struct pile{typeelt tab[max] ;  
           int pointp;}; // pointeur de pile indice du sommet de la pile  
Typedef struct pile * ppile;
```



# LES FILES

## REPRESENTATION INTUITIVE

- Type particulier de liste où les éléments sont insérés en queue et supprimés en tête.



- Le nom vient des files d'attente à un guichet où « le premier arrivé » est « le premier parti » : ce qui justifie le terme anglo-saxon « FIFO List »

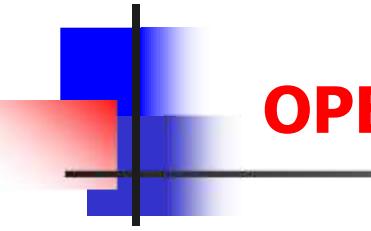
# **LES FILES**



## **EXEMPLES**

- Gestion d'accès à une ressource informatique : en système les files sont utilisées pour gérer tous les processus en attente de ressource système
- D'une façon générale, la notion de file intervient dans un modèle dès qu'il est question d'une file d'attente : systèmes de réservation, gestion de pistes d'aéroport .....etc.

# LES FILES



## OPERATIONS SUR LES FILES

---

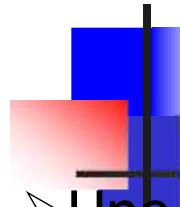
**Tete(F)** : retourne l'élément en tête de F

**Queue(F)** : retourne l'élément en fin de F

**Enfiler(x,F)** : insère l'élément x à la fin de la file F

**Defiler(F)** : supprime le premier élément de la file F

# LES FILES



## MISE EN ŒUVRE DES FILES PAR POINTEURS

- Une file est une liste dont on conserve en permanence le premier élément qui représente la tête de la file et le dernier élément qui représente la queue de la file.
- Elle est définie par deux pointeurs :
  - dernier : pointeur sur la queue pour enfiler les éléments
  - premier : pointeur sur la tête pour défiler les éléments
- A part cette spécificité c'est un cas normal de liste chainée
- En dynamique une file n'a pas de taille limite, hormis celle de la mémoire centrale (RAM) disponible.

# LES FILES

## MISE EN ŒUVRE DES FILES PAR POINTEURS

```
struct elem{ typelt val ;  
            struct elem *suiv;};  
Typedef struct elem * pelem;
```

```
struct file{  
    pelem premier; // sortie  
    pelem dernier; // entrée  
};
```

```
Typedef struct file * pfile;
```

# LES FILES

## IMPLEMENTATION PAR POINTEURS

**Enfiler:** insérer un nouveau élément consiste à effectuer les opérations suivantes :

- \* le suivant du dernier devient le nouveau
- \* le dernier devient le nouveau.

Si la file est vide alors premier et dernier prennent la valeur du nouvel élément

**Défiler:** consiste à retourner l'adresse de l'élément sortant et passer la tête au suivant

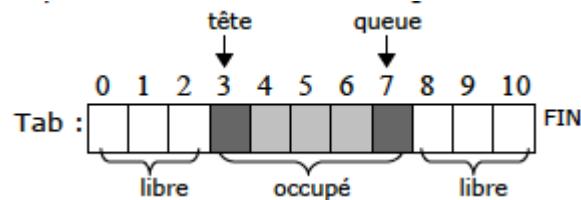
Deux cas sont à envisager :

- \* *file avec un seul élément* : on retourne son adresse et on met les deux pointeurs premier et dernier à NULL.
- \* *file avec plusieurs éléments* : on retourne l'adresse du premier et le premier prend l'adresse du suivant

# LES FILES

## IMPLEMENTATION PAR UN TABLEAU

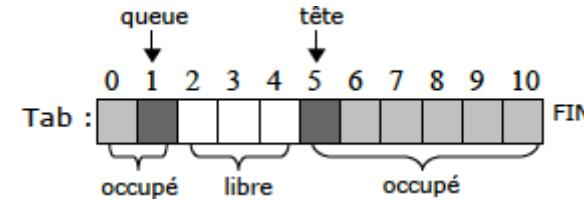
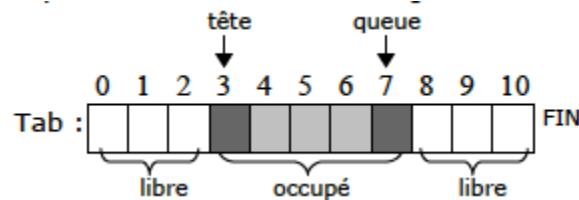
- premier : indice pour la tête de la file
- dernier : indice pour la queue de la file
- un tableau d'éléments de taille Nbmax
- Les données se trouvent entre 'premier' et 'dernier'



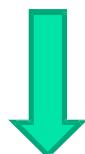
```
#define Nbmax ....  
struct file { int premier,dernier;  
    type1t Tab[Nbmax];  
}  
typedef struct file *pfile;
```

# LES FILES

## IMPLEMENTATION PAR UN TABLEAU

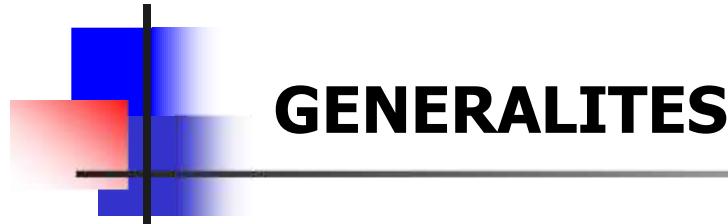


Les nouveaux éléments entrants peuvent être placés au début ou à la fin selon la place restante dans le tableau



Gestion du tableau de manière cyclique

# **LES ARBRES**



## **GENERALITES**

Les listes sont des structures dynamiques unidimensionnelles. Les arbres sont leur généralisation multidimensionnelle. Comme pour le premier d'une liste, l'adresse de la racine est nécessaire et suffisante pour accéder à l'intégralité d'un arbre.

# LES ARBRES

## DEFINITION

Un arbre peut se définir de manière récursive :

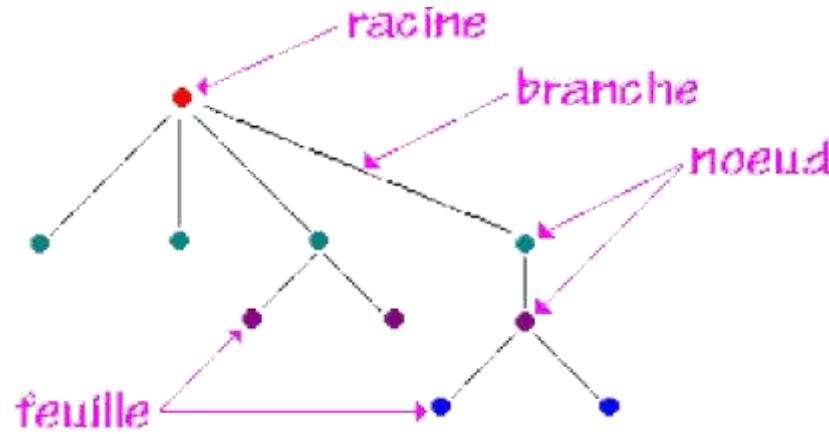
- un sommet unique, par lui même, est un arbre. Dans ce cas, il est aussi la racine de l'arbre
- si  $n$  est un sommet et  $A_1, A_2, \dots, A_k$  sont des arbres de racines respectives  $n_1, n_2, \dots, n_k$ ; on peut construire un nouvel arbre en associant comme parent unique aux sommets  $n_1, n_2, \dots, n_k$  le sommet  $n$ .

Dans cet arbre,  $n$  est la racine et  $A_1, A_2, \dots, A_k$  sont les ss-arbres. Les sommets  $n_1, n_2, \dots, n_k$  sont appelés les fils de  $n$ .

# LES ARBRES

## TERMINOLOGIE

- On utilise un vocabulaire inspiré des arbres généalogiques
- Chaque composante d'un arbre contient une valeur et des liens vers ses fils.



- Chaque élément d'un arbre se nomme un nœud
- Le nœud racine est le seul qui n'est le fils d'aucun nœud.
- Un nœud qui n'a pas de fils est un noeud externe ou feuille
- Un nœud qui n'est pas une feuille est dit interne

# LES ARBRES

## TERMINOLOGIE

- Chemin: suite unique de sommets
- Longueur d'un chemin : nombre d'arcs intervenant dans ce chemin
- La hauteur d'un noeud  $s$  : la longueur du plus long chemin issu de  $s$
- La profondeur d'un noeud  $s$  : longueur du chemin  $r \rightarrow s$ .
- La hauteur de l'arbre : la profondeur maximale de ses nœuds.
- On appelle degré d'un nœud, le nombre des fils qu'il possède
- Etant donné un sommet  $u$ , tous les sommets, autres que lui même, situés sur le chemin de la racine à  $u$  sont appelés des ancêtres, le dernier étant son père.
- Deux sommets qui ont le même père sont des frères

# LES ARBRES

## TERMINOLOGIE

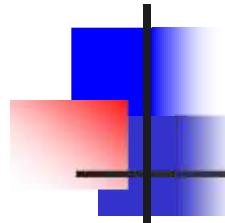
### Arité d'un arbre :

Un arbre dont les nœuds ne comportent qu'au maximum  $n$  fils est un arbre d'arité  $n$ . On parle alors d'un arbre  $n$ -aire. Un arbre d'arité 2 est un arbre binaire. On parle alors de fils gauche et de fils droit.

### Taille d'un arbre :

La taille d'un arbre est le nombre de nœuds qui le composent

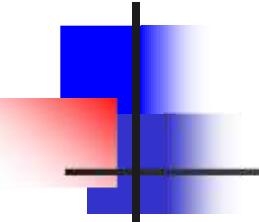
# LES ARBRES



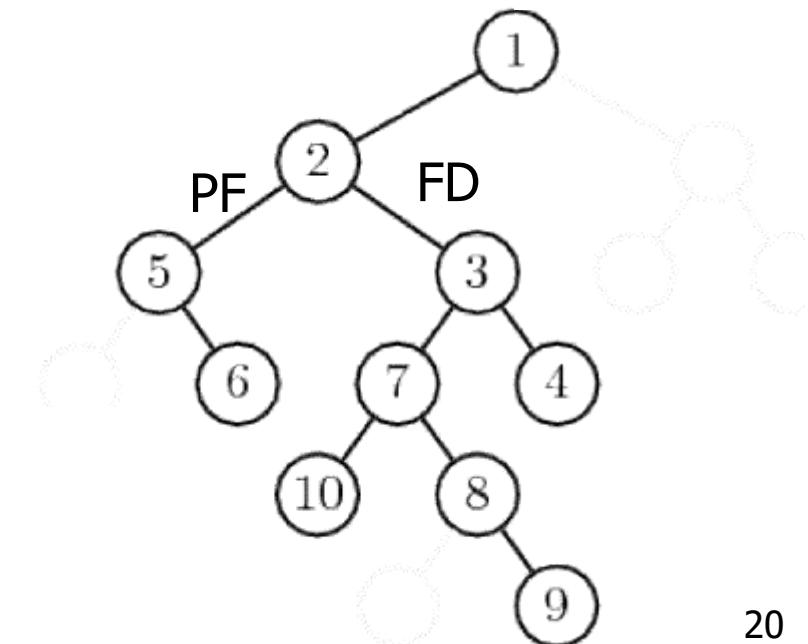
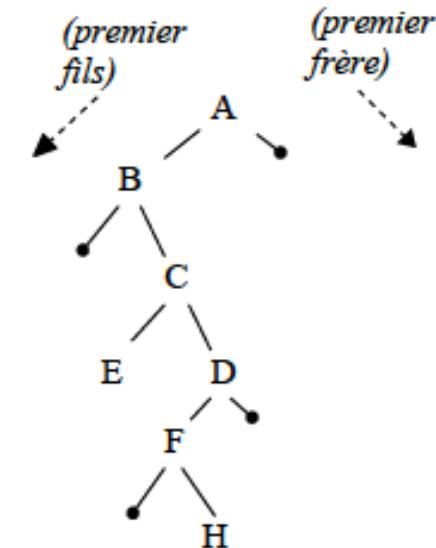
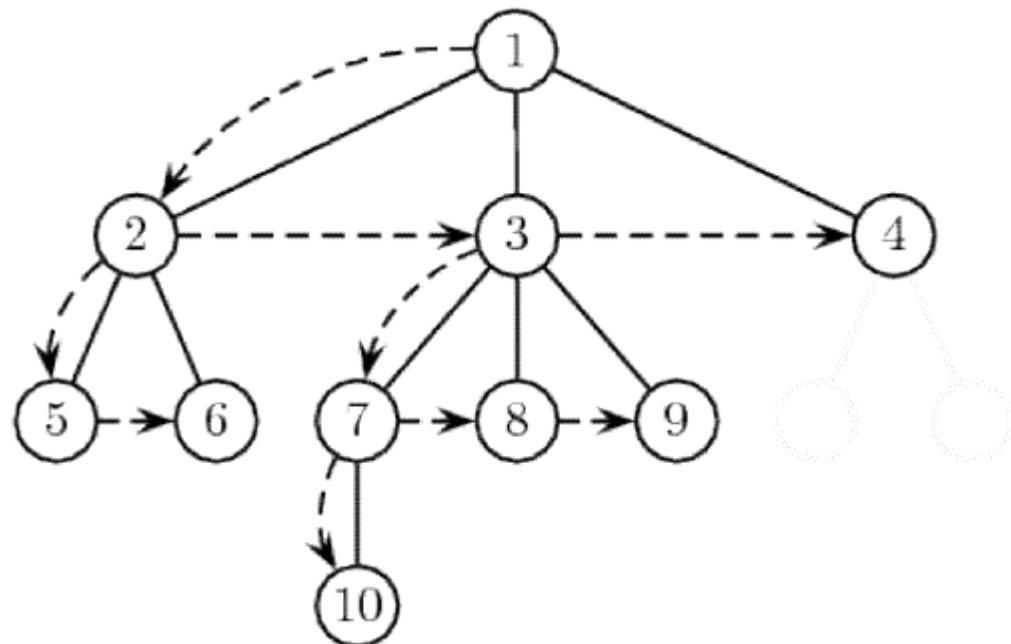
## TERMINOLOGIE

- Notion d'ordre
  - Tout sommet est la racine d'un arbre formé par sa descendance et lui-même
    - ↓  
récursivité de certains algorithmes sur les arbres
  - Les fils d'un sommet ne sont pas dans un ordre arbitraire et dans ce cas, on parlera de fils aîné, cadet, etc .. l'ordre étant de gauche à droite
  - $d^+(s)$  « degré sortant » est le nombre de fils du sommet s
  - $d^-(s)$  « degré entant » est le nombre de pères du sommet s
- N.B:**  $\sum d^+(s) = \sum d^-(s) = a = n-1$   
où a est le nombre d'arcs et n le nombre de sommets

# LES ARBRES



Possibilité de passer d'un arbre n-aire à un arbre binaire grâce aux primitives Premier fils (PF) et Frère Droit ou Premier frère (FR)



# LES ARBRES

# **PARCOURS D'ARBRES**

Le **parcours** d'arbre est une opération qui consiste à **retrouver** systématiquement tous les noeuds de cet arbre et d'y appliquer un **même traitement**

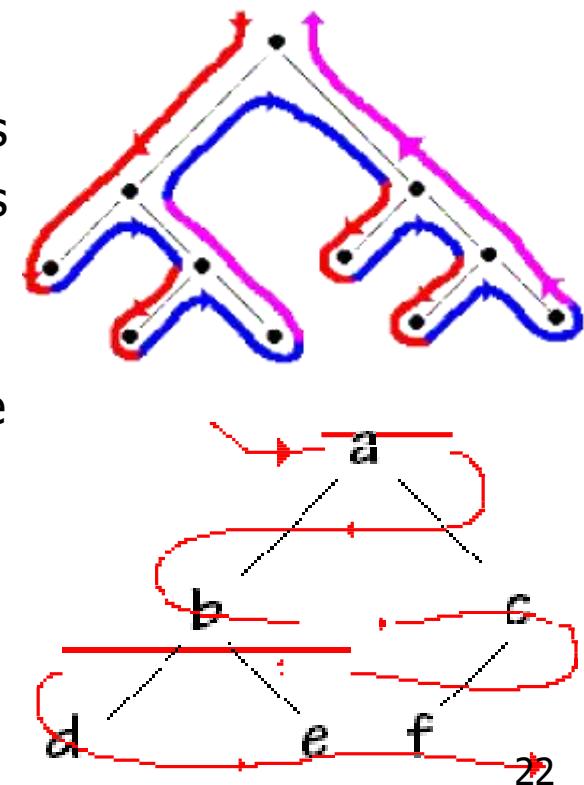
## ➤ Parcours en profondeur

Parcourir un arbre en profondeur consiste à visiter ses noeuds en descendant le plus profondément possible dans l'arbre.

Lorsque l'on arrive sur un arbre vide, on remonte jusqu'au noeud supérieur et on redescend dans le fils encore inexploré

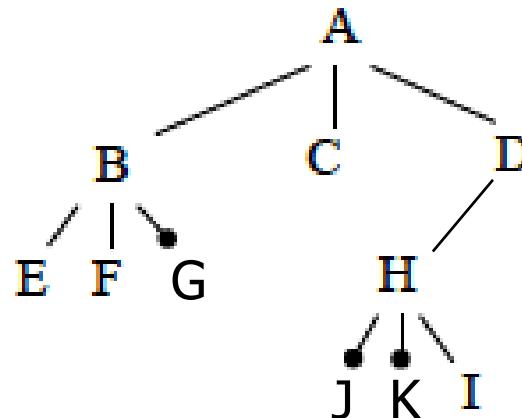
## ➤ Parcours en largeur

On explore chaque noeud d'un niveau donné de gauche à droite, puis on passe au niveau suivant.



# LES ARBRES

## PARCOURS D'ARBRES



**Parcours en largeur :** A B C D E F G H J K I

**Parcours en profondeur :** A B E F G B C D H J K I H D A

# LES ARBRES

## PARCOURS EN PROFONDEUR

Début

$x \leftarrow$  racine

Repeter

Si  $x$  a un fils alors  $x \leftarrow$  Premier\_fils( $x$ )

Sinon

Debut

tant que  $x$  n'a pas de frère et  $x \neq$  racine faire

$x \leftarrow$  père( $x$ )

Si  $x \neq$  racine alors  $x \leftarrow$  Frère\_droit ( $x$ )

Fin

Jusqu'à  $x =$  racine

Fin

# LES ARBRES

## PARCOURS EN PROFONDEUR

Début

$x \leftarrow$  racine

Creer-Pile(P)

Repeter

Si  $x$  a un fils alors

Empiler( $x, P$ )

$x \leftarrow$  Premier\_fils( $x$ )

Sinon

Debut

tant que  $x$  n'a pas de frère et  $x \neq$  racine faire

$x \leftarrow$ sommet( $P$ ) //tete( $P$ )

depiler( $P$ ) // père( $x$ )

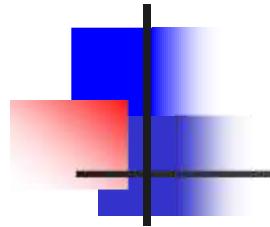
Si  $x \neq$  racine alors  $x \leftarrow$  Frère\_droit ( $x$ )

Fin

Jusqu'à  $x =$  racine // ou pilevide( $P$ )

Fin

# LES ARBRES



## PARCOURS EN LARGEUR

Début

    créer-file(F)

    Enfiler (F,racine)

    Tant que non-vide(F) faire

        y ← Tete(F)

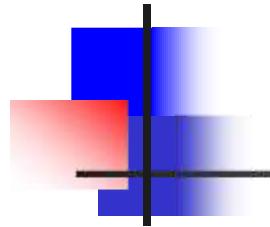
        Defiler(F)

        Pour tout fils z de y    Faire Enfiler(F,z)

    fin

Fin

# LES ARBRES



## PARCOURS EN LARGEUR

Début

    créer-file(F) **initialiser le tableau prof[n] à 0**

    Enfiler (F,racine)

    Tant que non-vide(F) faire

        y  $\leftarrow$  Tete(F)

        Defiler(F)

        Pour tout fils z de y    Faire Enfiler(F,z) **Prof[z] =prof[y]+1**

    fin

Fin

# ARBRES BINAIRES

## Définition et Implémentation

Un arbre binaire est soit un arbre vide, soit un arbre où chaque sommet a un fils gauche, un fils droit ou les deux fils à la fois.

```
struct noeud {  
    typeلت     info ;  
    struct noeud *fg ;  
    struct noeud *fd ;} ;  
  
typedef struct noeud *arbre ;
```

# LES ARBRES BINAIRES



## PARCOURS

### ➤ Parcours en largeur

On parcourt tous les sommets de l'arbre binaire par niveau de profondeur.  
On commence par la racine, puis tous les sommets (de gauche à droite) de profondeur 1 , puis tous les sommets de profondeur 2 , etc.

### ➤ Parcours en profondeur

On parcourt récursivement le sous-arbre gauche puis le sous-arbre droit.  
La visite de la racine se fait :

**Avant** les appels récursifs pour un **parcours préfixé**.

**Entre** les deux appels récursifs pour un **parcours infixé**.

**Après** les appels récursifs pour un **parcours postfixé**.

# ARBRES BINAIRES

## PARCOURS

Soit un arbre binaire de racine n et de ss-arbres  $A_1$  (fg) et  $A_2$  (fd)

**Parcours préfixé** : on commence par la racine n puis on entreprend le parcours préfixé des sommets de  $A_1$ , puis de  $A_2$

```
void prefixe(arbre r)
{
    if(r!=NULL){
        traiter( r );
        prefixe(r->fg);
        prefixe(r->fd);
    }
}
```

# ARBRES BINAIRES

## PARCOURS

**Parcours infixé** : on commence par le parcours infixé des sommets de  $A_1$ , on passe par la racine  $n$ , puis on termine par le parcours infixé des nœuds de  $A_2$

```
void infixe(arbre r)
{
    if(r!=NULL){
        infixe(r->fg);
        traiter(r->info);
        infixe(r->fd);
    }
}
```

# ARBRES BINAIRES

## PARCOURS

**Parcours postfixé** : on effectue le parcours postfixé des nœuds de  $A_1$ , de  $A_2$ , et l'on termine par la racine

```
void postfixe(arbre r)
{
    if(r!=NULL){
        postfixe(r->fg);
        postfixe(r->fd);
        traiter(r->info);
    }
}
```

# ARBRES BINAIRES

## QUELQUES ARBRES BINAIRES PARTICULIERS

- Arbre binaire localement complet est un arbre binaire où chaque noeud possède soit 0 soit 2 fils.
- Arbre binaire complet est un arbre localement complet et dont toutes les feuilles ont la même profondeur.
- Un arbre dégénéré est un arbre dont les noeuds ne possèdent qu'un seul fils. Cet arbre est donc tout simplement une liste chaînée. Ce type d'arbre est à éviter puisqu'il n'apportera aucun avantage par rapport à une liste chaînée simple.

# ARBRES BINAIRES

## ARBRES COMPLETS

### Définition

Un **arbre binaire complet** est un arbre binaire tel que :

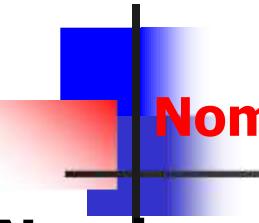
- Tous les sommets sont de degré 0 ou 2 .
- Toutes les feuilles ont la même profondeur.

### Propriété

Soit G un **arbre binaire complet** d'ordre n , de hauteur h et soit f son nombre de feuilles. On a :

- $n=2^{h+1}-1$
- $f=2^h$

# ARBRE BINAIRES



**Nombre d'arbres binaires à n noeuds/Hauteur d'un arbre binaire**

## Nombre d'arbres binaires à n noeuds

Le nombre d'arbres binaires à  $n$  noeuds  $C_n$ , dit *nombre de Catalan* est :

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

## Hauteur d'un arbre binaire

Soit un arbre binaire non vide à  $n$  noeuds de hauteur  $h$ . On a :

- $h+1 \leq n \leq 2^{h+1}-1$
- $\log_2(n+1) - 1 \leq h \leq n - 1$

Ces bornes sont optimales.

# ARBRE BINAIRES DE RECHERCHE

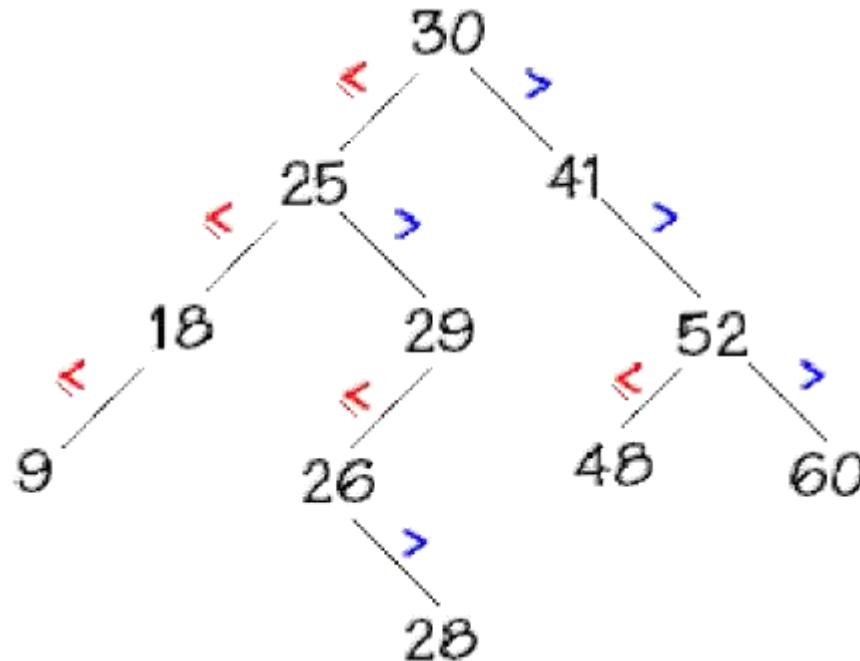
## DÉFINITION

Un arbre binaire de recherche est un arbre étiqueté tel que pour tout sommet  $v$  de l'arbre :

- les éléments de tous les sommets du sous-arbre gauche de  $v$  sont inférieurs ou égaux à l'élément contenu dans  $v$
- les éléments de tous les sommets du sous-arbre droit de  $v$  sont supérieurs à l'élément contenu dans  $v$

# ARBRE BINNAIRE DE RECHERCHE

## EXEMPLE

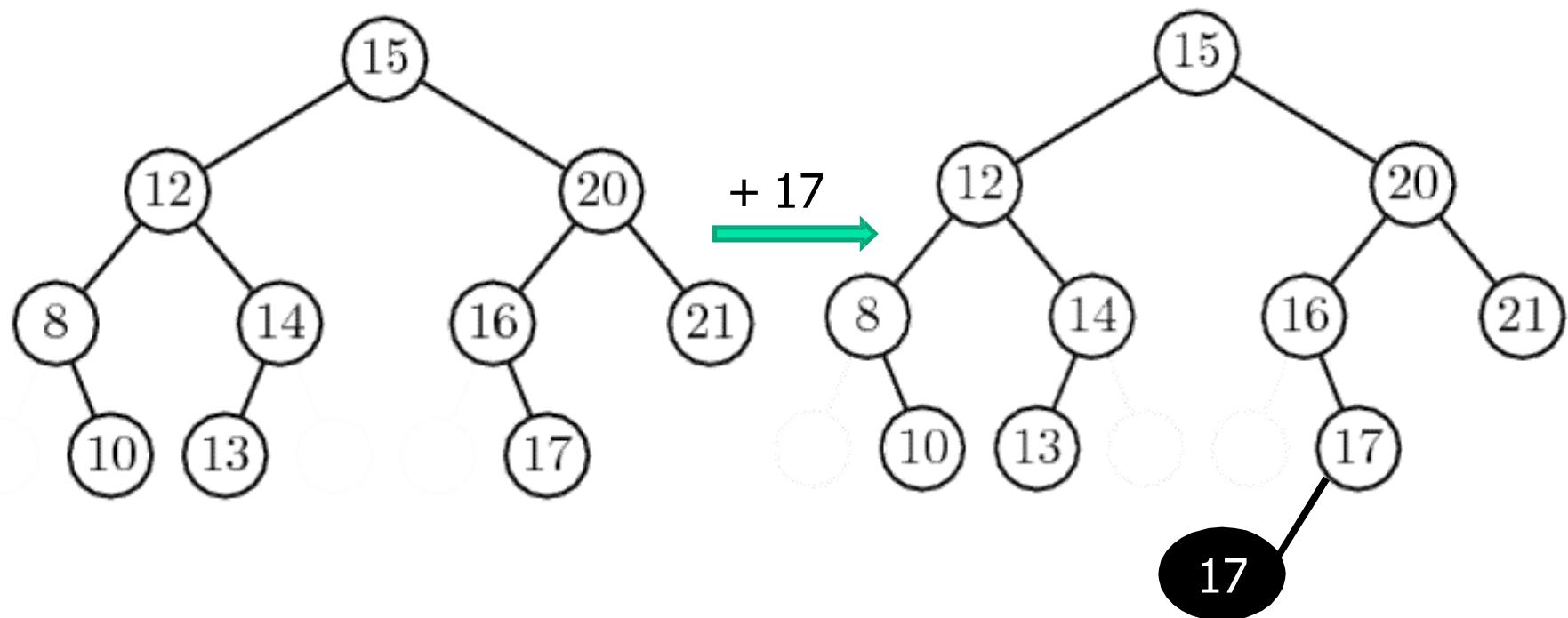


**N.B:** Le parcours infixé d'un arbre binaire de recherche produit la suite des éléments triée par ordre croissant

# ARBRE BINNAIRE DE RECHERCHE

## INSERTION

L'adjonction du nœud est effectuée de telle manière à ce que l'arbre reste binaire de recherche



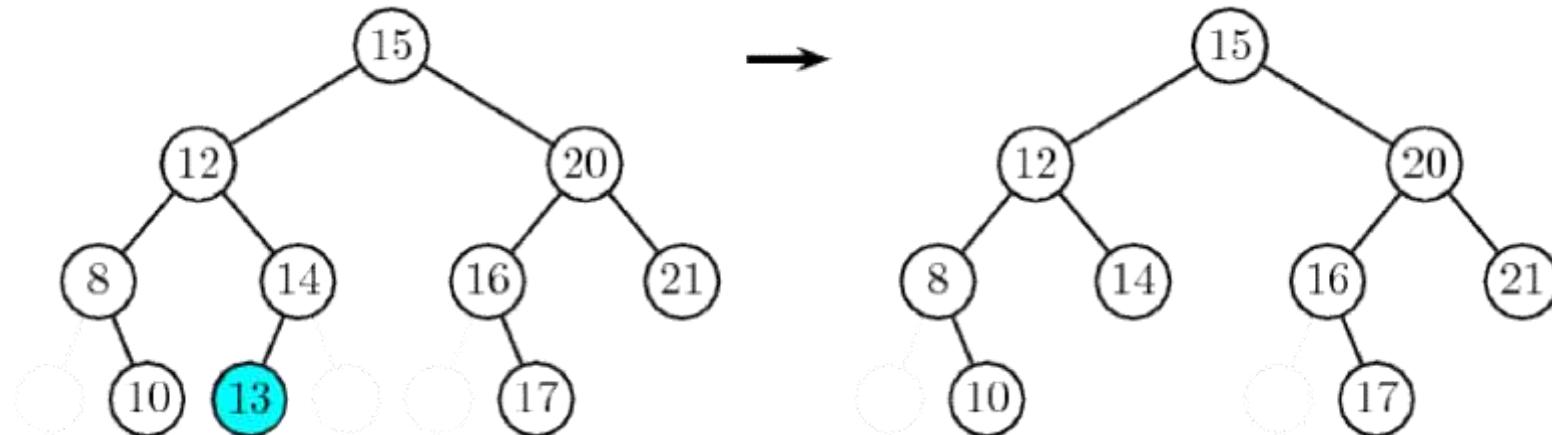
# ARBRE BINNAIRE DE RECHERCHE

## SUPPRESSION

On désire supprimer le nœud  $s$  d'étiquette  $x$

Trois cas sont à considérer selon le nombre de fils du nœud  $x$ :

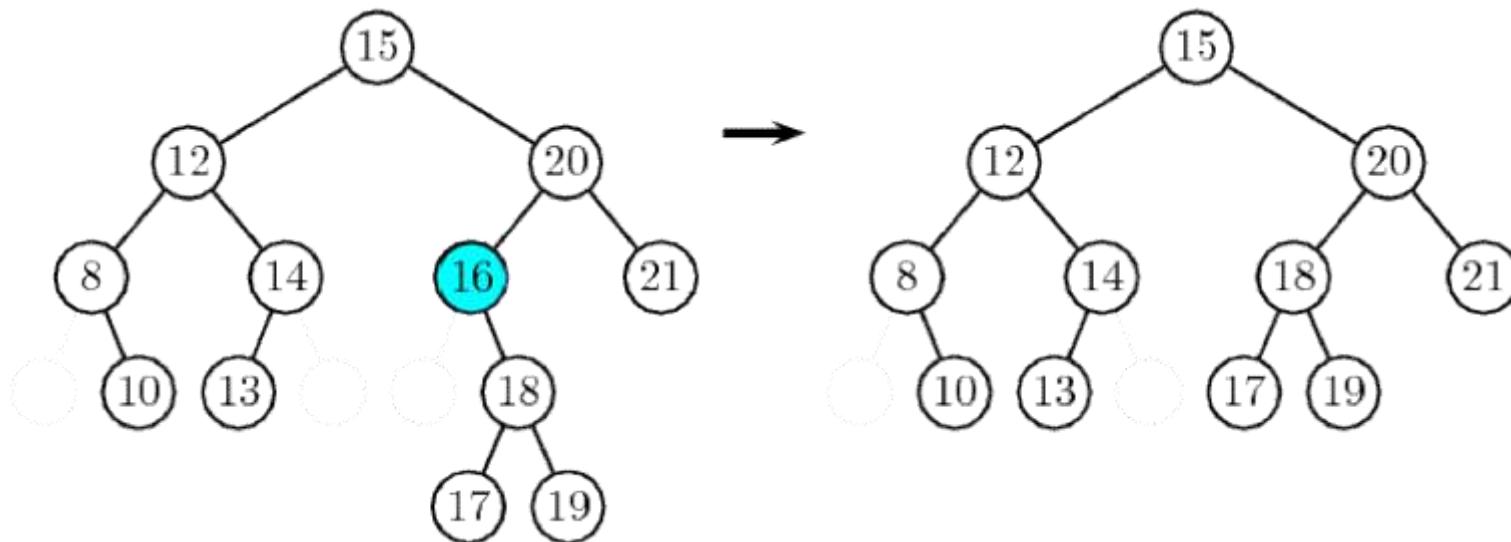
**1<sup>er</sup> cas :** si le noeud  $s$  est une feuille, alors on l'élimine



# ARBRE BINAIRES DE RECHERCHE

## SUPPRESSION

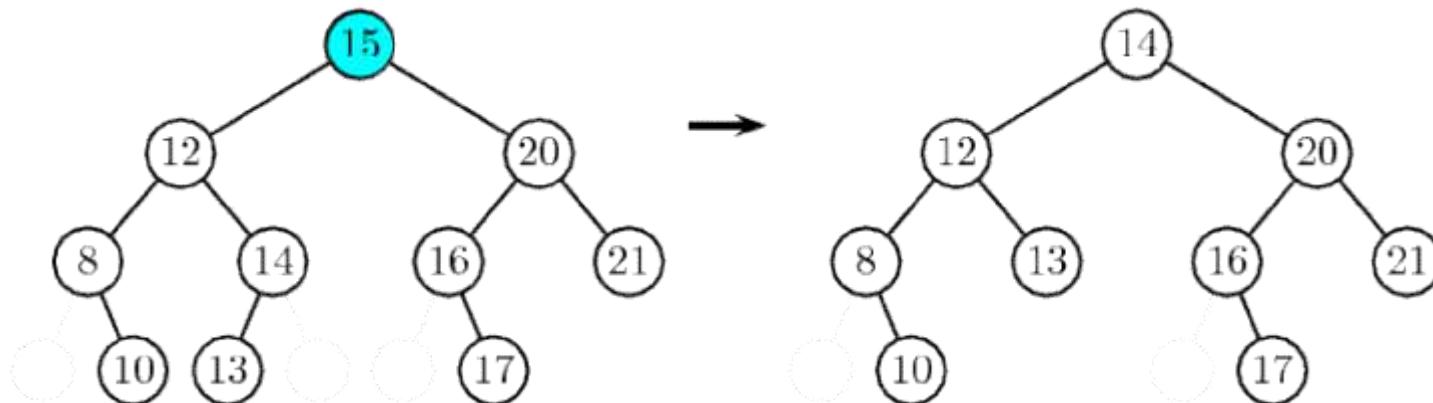
2<sup>ème</sup> cas : si le nœud  $s$  possède un seul fils, on élimine  $s$  et on « remonte » ce fils.



# ARBRE BINNAIRE DE RECHERCHE

## SUPPRESSION

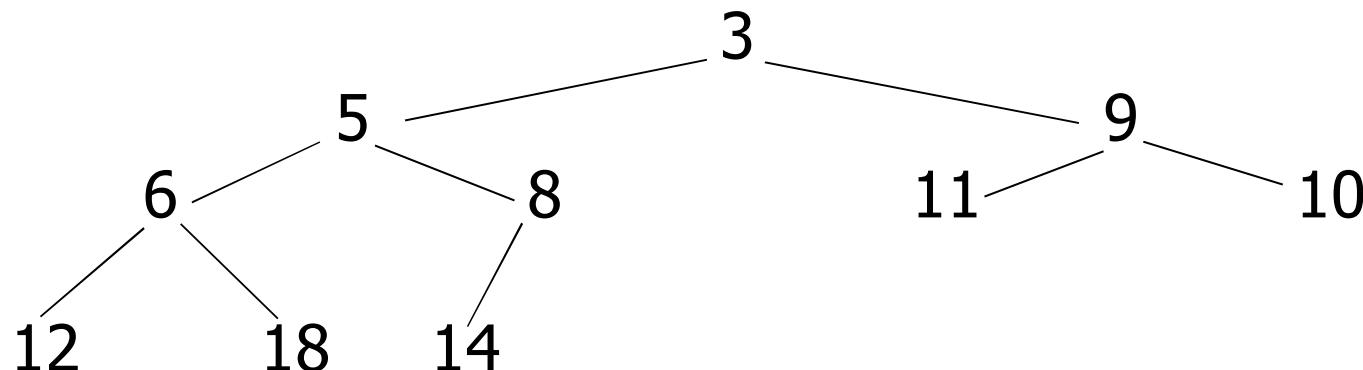
3ème cas : si le nœud s possède deux fils, Pour conserver l'ordre sur les clés, il n'y a que deux choix : la clé du prédécesseur dans l'ordre infixé, ou la clé du successeur.



# ARBRE PARFAIT PARTIELLEMENT ORDONNE

## DEFINITION

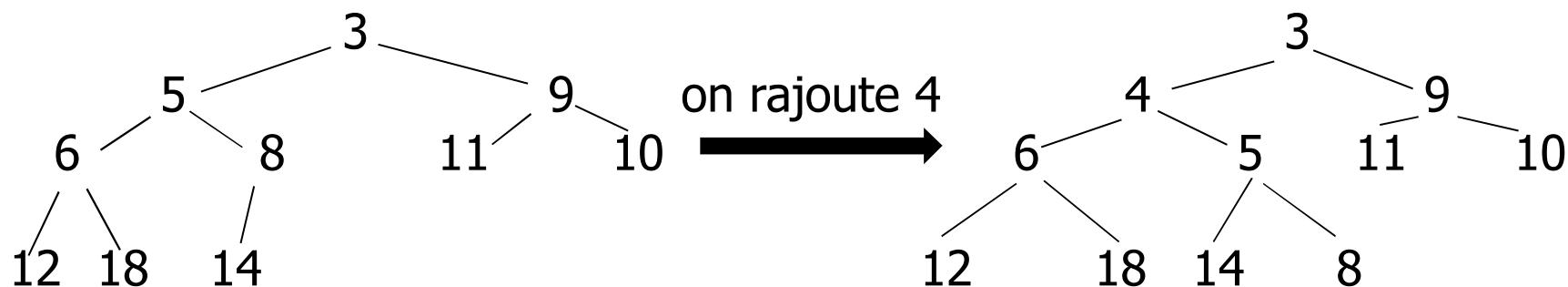
- Un arbre partiellement ordonné est un arbre étiqueté par des éléments d'un ensemble, muni d'un ordre total, tel que l'étiquette de tout nœud est inférieure à celles de ses fils.
- Un arbre parfait est un arbre dont toutes les feuilles sont sur deux niveaux seulement, avec l'avant-dernier niveau qui est complet et les feuilles du dernier niveau qui sont regroupés le plus à gauche possible.



# ARBRE PARFAIT PARTIELLEMENT ORDONNE

## INSERTION

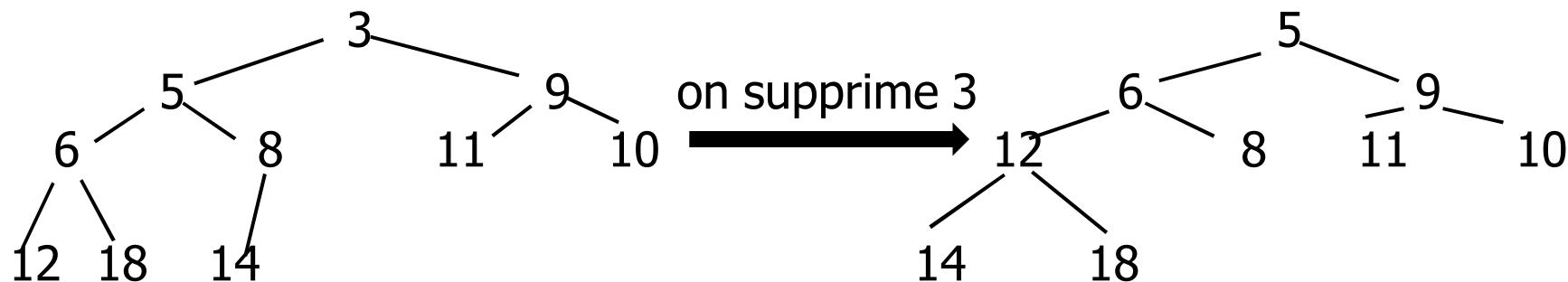
- Si on représente une liste par un tel arbre; le minimum se trouve à la racine
- L'adjonction d'un nouvel élément implique l'ajout d'une feuille au dernier niveau de l'arbre et la réorganisation de cet arbre



# ARBRE PARFAIT PARTIELLEMENT ORDONNE

## SUPPRESSION

Pour supprimer le minimum, la dernière feuille du dernier niveau disparaît et on la place à la racine et on réorganise l'arbre de telle sorte qu'il reste partiellement ordonné.



Le parcours en largeur d'un arbre binaire P.P.O est appelé **TAS**