

Structures de données en C

Pr. Youness KHOUREDIFI, PhD en Informatique

**Professeur à la Faculté Polydisciplinaire – Khouribga –
Université Sultan Moulay Slimane – Béni Mellal –**

Consultant IT : SQL 2016 Database Administration, Core Infrastructure 2016,
Azure Solutions Architect Expert, Data Analyst Associate, Ingénieur DevOps.

Membre Springer : ORCID ID 0000-0002-1144-8455

Membre IEEE : ID # 94836813

Membre IAENG : ID 214078

Scopus Author ID: 57202871015

youness.khourdifi@usms.ac.ma

Plan

☐ Chapitre 1 : Rappels

- Tableaux
- Pointeurs & Fonctions
- Récursivité
- Allocation dynamique & Structure
- Tri et recherche & Gestion des fichiers

☐ Chapitre2 : Les listes

- Les listes chaînées
- Manipulation (simplement et doublement chaînées)

Plan

❏ Chapitre3 : Les piles et les files

- Les piles
- Les files

❏ Chapitre4 : Les arbres binaires

- Arbres binaires : définition et représentation
- Opérations sur les arbres binaires
- Parcours des arbres binaires
- Arbres binaires de recherche

❏ Conclusion et Evaluation

Chapitre

Rappels

Tableaux :

Activité : Programme des notes

- ❑ Créer un programme en C qui permet de stocker des notes afin de calculer la moyenne, le max et le min.

```
#include <stdio.h>
int main () {
    float N1, N2, N3, N4;
    printf ( " Donner la note de l'étudiant num 1 : " );
    scanf ( "%f", &N1 );
    printf ( " Donner la note de l'étudiant num 2 : " );
    scanf ( "%f", &N2 );
    printf ( " Donner la note de l'étudiant num 3 : " );
    scanf ( "%f", &N3 );
    printf ( " Donner la note de l'étudiant num 4 : " );
    scanf ( "%f", &N4 );
    return 0;
}
```

Tableaux :

- ❑ Un **tableau** est une variable qui se compose d'un certain nombre de données de même type, rangées en mémoire les unes après les autres.
- ❑ Chaque donnée représente elle-même une variable.
- ❑ Le type d'un tableau peut être n'importe lequel :
 - **Type élémentaires** : char, short, int, long, float, double;
 - **Pointeur**
 - **Structure**
 - ...



Tableaux à une dimension

- ❑ Un tableau unidimensionnel est composé d'éléments qui ne sont pas eux-mêmes des tableaux.
- ❑ On pourrait le considérer comme ayant un nombre fini de colonnes, mais une seule ligne. Par exemple, le tableau suivant est constitué de N éléments de type int :

Tableaux à une dimension

- ❑ Dans un tableau il faut préciser le **type** de données leurs **nombre**, ainsi que le **nom** sous lequel le programme pourra y accéder a ces éléments.
- ❑ La définition d'un tableau unidimensionnel admet la syntaxe suivante :

Type **Nom_Du_Tableau**[**Nombre d'éléments**] ;

- ❑ Cette déclaration signifie que le compilateur réserve **Nombre d'éléments** places en mémoire pour ranger les éléments du **Nom_Du_Tableau**.
- ❑ **Type** spécifie le type des éléments du tableau.
- ❑ Le **Nom_Du_Tableau** obéit aux règles régissant les noms de variables.
- ❑ **Nombre d'éléments** est une valeur constante entière, qui détermine le nombre d'éléments du tableau.

Remarque Nous avons ici à faire à une donnée statique, dont la taille n'est pas variable.

Accès aux éléments d'un tableau

Comment accède-t-on à un élément quelconque d'un tableau ?

- ❑ Un élément du tableau est repéré par son **indice**.
- ❑ En langage C les tableaux commencent à l'indice **0**.
- ❑ L'indice maximum est donc **taille-1**.

Example :

On déclare un tableau de type entier et de dimension 6.

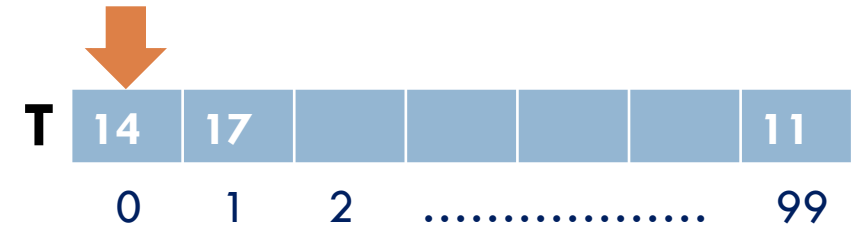
```
int T[6];
```

Pour accéder au éléments de T on procède comme suit:

- Syntaxe d'affectation : `nom_tab[indice] = valeur;`
- Syntaxe lecture : `scanf("%...", &nom_tab[indice]);`
- Syntaxe écriture : `printf("%...", nom_tab[indice]);`

Affectation de la note du candidat étudié de :
l'étudiant numéro 2 :

```
N[0] = 14;  
printf("%6f", N[99]);
```



Exemple 1 : Programme qui lit et qui affiche un tableau

```
1  #include <stdio.h>
2  int main()
3  ▼ {   int T[6], i; //Lecture des éléments du tableau T
4      for (i=0; i<6; i++)
5      ▼ {
6          printf("La saisie de l'element numero %d : ", i);
7          scanf("%d", &T[i]);
8      }
9      printf("Vous avez donner comme tableau \n");
10     for(i=0; i<6; i++)
11     printf("%d  ,\n",T[i]);
12 }
```

```
La saisie de l'element numero 0 : 2
La saisie de l'element numero 1 : 12
La saisie de l'element numero 2 : 22
La saisie de l'element numero 3 : 34
La saisie de l'element numero 4 : 65
La saisie de l'element numero 5 : 7
Vous avez donner comme tableau
2 ,
12 ,
22 ,
34 ,
65 ,
7 ,
Process returned 0 (0x0)   execution time : 14.874 s
Press any key to continue.
```

Exemple 2 : Programme qui lit et qui affiche un tableau

```
1  #include <stdio.h>
2  int main()
3  {   int i, taille;
4  do {
5      printf("Donnez la taille du tableau : ");
6      scanf("%d", &taille);
7  }
8  while (taille <=0);
9  float tab[taille];
10 for (i=0; i<taille; i++)
11 {
12     printf("La saisie de l'element numero %d : ", i);
13     scanf("%f", &tab[i]);
14 }
15 printf("Tab = \n");
16 for(i=0; i<taille ; i++)
17     printf("%f \n",tab[i]);
18 }
```

```
Donnez la taille du tableau : 4
La saisie de l'element numero 0 : 12
La saisie de l'element numero 1 : 32
La saisie de l'element numero 2 : 11
La saisie de l'element numero 3 : 8
Tab =
12.000000
32.000000
11.000000
8.000000
```

```
Process returned 0 (0x0)   execution time : 18.239 s
Press any key to continue.
```

Exercices :

Exercice 1 :

- ❑ Ecrire un programme qui effectue le produit scalaire de deux vecteurs de même taille (3 éléments) représentés par des tableaux à une dimension.

```
1  #include <stdio.h>
2  int main()
3  {
4      float U[3], V[3];
5      int i;
6      float P;
7      printf("Veuillez saisir les valeurs des deux vecteurs : \n");
8      for (i = 0; i < 3; i++)
9      {
10         printf("U[%d] = ", i);
11         scanf("%f", &U[i]);
12         printf("V[%d] = ", i);
13         scanf("%f", &V[i]);
14     }
15     P = 0;
16     for (i = 0; i < 3; i++)
17     {
18         P = P + U[i] * V[i];
19     }
20     printf("Le produit scalaire des deux vecteurs est : %.2f", P );
21 }
```

Exercices :

Exercice 2 :

- ☐ Ecrire un programme qui demande à l'utilisateur de saisir 10 entiers qu'on stocke dans un tableau T.
- ☐ Ensuite, le programme détermine et affiche le minimum des éléments du tableau T.

```
2  int main()
3  {
4      int T[10];
5      int i, min;
6      printf("Veuillez saisir les elements du tableau : \n");
7      for (i = 0; i < 10; i++)
8      {
9          printf("T[%d] = ", i );
10         scanf("%d",&T[i]);
11     }
12     min = T[0];
13     for (i = 0; i < 10; i++)
14     {
15         if (min>T[i])
16         {
17             min = T[i];
18         }
19     }
20     printf("Le minimum des elements du tableau est : %d", min);
21 }
```

Exercices :

Exercice 3 :

- ❑ Ecrire un programme qui demande à l'utilisateur d'entrer des éléments dans un tableau, puis le programme place les éléments pairs et impairs dans deux tableaux séparés.

```
1  #include<stdio.h>
2  int main()
3  {
4      int T[100], P[100], I[100];
5      int i, Taille, Pcmp, Icmp;
6
7      printf("Veuillez saisir la taille du tableau : ");
8      scanf("%d", &Taille);
9      printf("Veuillez saisir les elements du tableau : \n");
10     for (i=0; i<Taille;i++) {
11         printf("T[%d] = ", i+1);
12         scanf("%d", &T[i]); }
13     Pcmp=0;
14     Icmp=0;
15     for (i=0; i<Taille;i++) {
16         if (T[i]%2==0) {
17             P[Pcmp]=T[i];
18             Pcmp++; }
19         else {
20             I[Icmp]=T[i];
21             Icmp++; }
22     }
23     printf("\nLes elements pairs du tableau sont ");
24     for(i=0; i<Pcmp;i++)
25         printf("%d ", P[i]);
26
27     printf("\nLes elements impairs du tableau sont ");
28     for(i=0; i<Icmp;i++)
29         printf("%d ", I[i]);
30     return 0;
31 }
```

Les pointeurs en langage C

Introduction :

- ☐ Toutes les variables qu'on utilise dans nos programmes sont stockées quelque part dans la mémoire centrale.
- ☐ La mémoire peut être assimilée à un tableau où chaque case est identifiée par une "adresse".
- ☐ Pour retrouver une variable, il suffit donc, de connaître l'adresse de la case (l'emplacement mémoire où elle est stockée)
- ☐ C'est le compilateur qui fait le lien entre l'identificateur (nom) d'une variable et son adresse dans la mémoire.

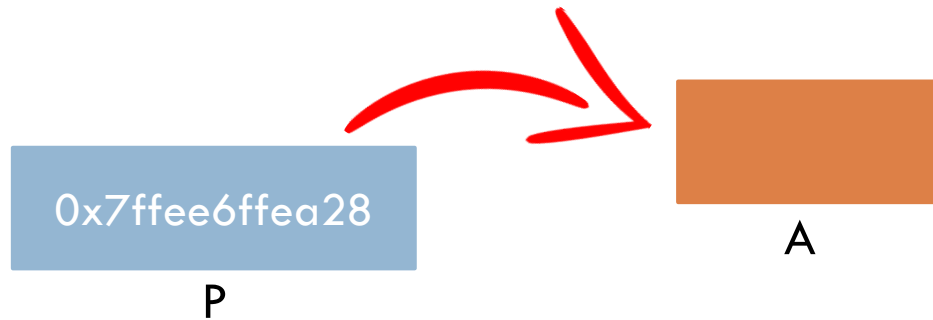


Il peut être plus intéressant d'écrire une variable non plus par son identificateur mais directement par son adresse.

Les pointeurs en langage C

Définition d'un pointeur :

- ❑ Un pointeur est une variable spéciale qui contient l'adresse d'une autre variable.
- ❑ Chaque pointeur est limité à un type de données.
- ❑ Si un pointeur P contient l'adresse d'une variable A, on dit que 'P pointe sur A'.



Les pointeurs en langage C

Déclaration et initialisation d'un pointeur en C

Déclaration

- ❑ Un pointeur est une variable dont la valeur est égale à l'adresse d'une autre variable. En C, on déclare un pointeur par l'instruction :

type *nom_du_pointeur;

Où


- ❑ **type** est le type de la variable pointée,
- ❑ l'identificateur **nom_du_pointeur** est le nom de la variable pointeur
- ❑ ***** est l'opérateur qui indiquera au compilateur que c'est un pointeur.

Exemple : **int *p;** On dira que :

- ❑ **p** est un pointeur sur une variable du type **int**, ou bien p peut contenir l'adresse d'une variable du type int
- ❑ ***p** est de type **int**, c'est l'emplacement mémoire pointé par p.

Les pointeurs en langage C

Remarques :

- ☐ A la déclaration d'un pointeur p, il ne pointe a priori sur aucune variable précise : p est un pointeur non initialisé.
-  Toute utilisation de p devrait être précédée par une initialisation.
- ☐ La valeur d'un pointeur est toujours un entier (codé sur 32bits ou 64bits). Le type d'un pointeur dépend du type de la variable vers laquelle il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet :
 - ☐ Pour un pointeur sur une variable de type char, la valeur donne l'adresse de l'octet où cette variable est stockée.
 - ☐ pour un pointeur sur une variable de type short, la valeur donne l'adresse du premier des 2 octets où la variable est stockée
 - ☐ Pour un pointeur sur une variable de type float, la valeur donne l'adresse du premier des 4 octets où la variable est stockée.

Les pointeurs en langage C

Quelque types de variables en C :

Type	Taille	Signé	Non signé
char	1 octet	-128 à 127	0 à 2^8-1
short int	2 octets	-2^{15} à $2^{15}-1$	0 à $2^{16}-1$
int	4 octets	-2^{31} à $2^{31}-1$	0 à $2^{32}-1$
long int	4/8 octets (processeur 32/64 bits)	-2^{63} à $2^{63}-1$	0 à $2^{64}-1$
int *	4/8 octets (processeur 32/64 bits)	-2^{63} à $2^{63}-1$	0 à $2^{64}-1$
float	4 octets	-2^{31} à $2^{31}-1$	0 à $2^{32}-1$
double	8 octets	-2^{63} à $2^{63}-1$	0 à $2^{64}-1$
long double	12/16 octets (processeur 32/64 bits)	-2^{95} à $2^{95}-1$ -2^{127} à $2^{127}-1$	0 à $2^{96}-1$ 0 à $2^{128}-1$

Les pointeurs en langage C

Initialisation :

- ❑ Pour initialiser un pointeur, le langage C fournit l'opérateur unaire `&`.
- ❑ Ainsi pour récupérer l'adresse d'une variable A et la mettre dans le pointeur P (P pointe vers A) :

`P = &A`

Exemple 1 :

*int A, B, *P; /* supposons que ces variables occupent la mémoire à partir de l'adresse 01A0 */*

A = 10;

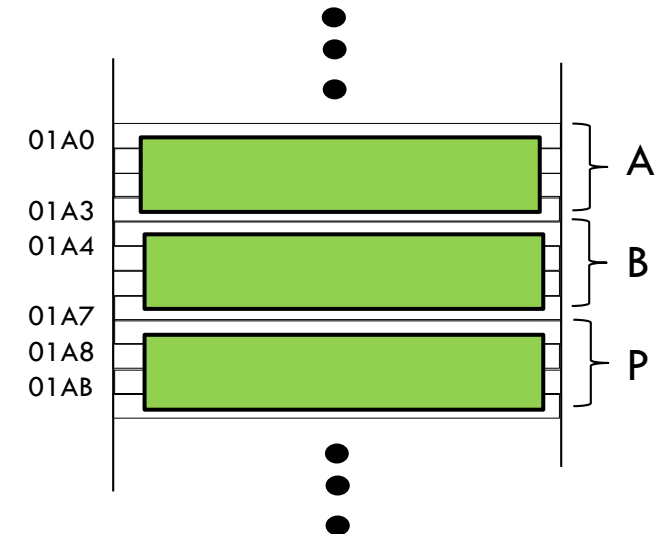
B = 50;

P = &A ; // se lit mettre dans P l'adresse de A

*B = *P; /* mettre dans B le contenu de la variable pointé par *P */*

**P = 20; /* mettre la valeur 20 dans la variable pointé par p*/*

P = &B; // P pointe sur B



Les pointeurs en langage C

Exemple 1 : Solution

```
int A, B, *P;
```

```
A = 10;
```

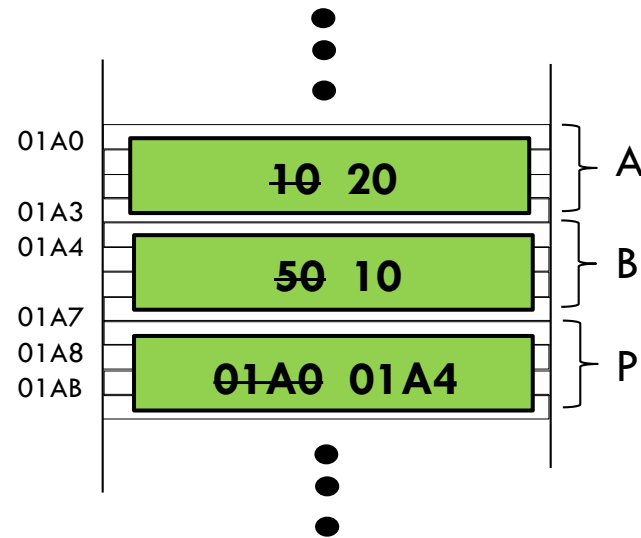
```
B = 50;
```

```
P = &A;
```

```
B = *P;
```

```
*P = 20;
```

```
P = &B;
```



Les pointeurs en langage C

Arithmétique des pointeurs :

On peut appliquer sur un pointeur quelques opérations arithmétiques :

- ☐ Ajouter un entier à un pointeur.
- ☐ Soustraire un entier d'un pointeur.
- ☐ Soustraire un pointeur d'un autre pointeur (de même
- ☐ type).

Les pointeurs en langage C

- ❑ Soit i et j deux entiers et p un pointeur sur un élément de type T ,
- ❑ L'expression $p' = p + i$ ($p' = p - i$) désigne un pointeur p' sur un élément de type T ,
- ❑ La valeur de p' est égale à la valeur de p incrémenté (décrémenté) de : $i * \text{sizeof}(T)$

Exemple:

```
main () {  
    int i = 5;  
    int *p1 = NULL, *p2 = NULL;  
    p1 = &i + 2;  
    p2 = p1 - 2;  
    int j = p1 - p2;  
}
```

Lvalue	Adresse	Valeur
i	6422000	5
p1	6422004	?
p2	6422008	?
j	6422016	?

Avec : $\text{sizeof}(int) == 4$ Octet

Les pointeurs en langage C

Solution :

```
main () {  
    int i = 5;  
    int *p1 = NULL, *p2 = NULL;  
    p1 = &i + 2;  
    p2 = p1 - 2;  
    int j = p1 - p2;  
}
```

Avec : sizeof (int) == 4 Octet

Lvalue	Adresse	Valeur
i	6422000	5
p1	6422004	6422008
p2	6422008	6422000
j	6422016	2

Les pointeurs en langage C

Opérations élémentaires sur les pointeurs :

- ❑ L'opérateur & : 'adresse de' : permet d'obtenir l'adresse d'une variable.
- ❑ L'opérateur * : 'contenu de' : permet d'accéder au contenu d'une adresse.
- ❑ Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple : `int X=1, Y, *P`

Après l'instruction, `P = &X`; On a :

<code>Y = X + 1</code>	équivalente à	<code>Y = *P + 1</code>
<code>X += 2</code>	équivalente à	<code>*P += 2</code>
<code>++X</code>	équivalente à	<code>++ *P</code>
<code>X++</code>	équivalente à	<code>(*P)++</code>

Les pointeurs en langage C

Opérations élémentaires sur les pointeurs :

- ❑ Le seul entier qui puisse être affecté à un pointeur d'un type quelconque P est la constante entière 0 désignée par le symbole **NULL** défini dans **<stddef.h>**
- ❑ On dit alors que **le pointeur P ne pointe « nulle part »**.

Exemple :

```
#include <stddef.h>
...
int *p, x, *q;
short y = 10, *pt=&y;
p = NULL; // Correct
p = 0; // Correct
x = 0;
p = x // Incorrect ! bien que x vaille 0
q = &x;
p = q; // Corrct : p et q pointe sur des variables de même type
p = pt; // Incorrect : p et pt pointe sur des variable de type différent
```

Les pointeurs en langage C

Les pointeurs et les tableaux :

En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

Adressage et accès aux composantes d'un tableau à une dimension :

En déclarant un tableau T de type int (`int T[N]`) et un pointeur P sur des variables entières (`int *P`), l'instruction `P = T` crée une liaison entre le pointeur P et le tableau T en mettant dans P l'adresse du premier élément de T (de même `P = &T[0]`).

```
int T[n];
```

```
int *p;
```

Créer une liaison entre le tableau T et le pointeur p:

```
p = T;
```



```
p = &T[0];
```

Les pointeurs en langage C

A partir du moment où $\mathbf{P} = \mathbf{T}$, la manipulation du tableau \mathbf{T} peut se faire par le biais du pointeur \mathbf{P} .

En effet :

- | | |
|-----------------------|-----------------------------|
| ➤ p pointe sur T[0] | *p désigne T[0] et p[0] |
| ➤ p+1 pointe sur T[1] | *(p+1) désigne T[1] et p[1] |
| ➤ ... | ... |
| ➤ p+i pointe sur T[i] | *(p+i) désigne T[i] et p[i] |

où $i \in [0, N-1]$

Les pointeurs en langage C

Exemple :

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int T[5] = {1, 2, 3, 4, 5};
    int i, *p;
    p = T;
    for (i=0; i<5; i++){
        printf ("*(p+%d)=%d \t", i, *(p+i));
        printf ("p[%d]= %d \t", i, p[i]);
        printf ("T[%d]=%d \n", i, T[i]);
    }
    exit(0); // ou exit(EXIT_SUCCESS) sortir du prog avec succès
}
```

Donner le résultat d'exécution de ce programme:



La fonction *exit* avec les paramètres suivants: 1 ou *EXIT_FAILURE* pour signaler un problème.

Les pointeurs en langage C

```
*(p+0)=1      p[0]= 1      T[0]=1
*(p+1)=2      p[1]= 2      T[1]=2
*(p+2)=3      p[2]= 3      T[2]=3
*(p+3)=4      p[3]= 4      T[3]=4
*(p+4)=5      p[4]= 5      T[4]=5

Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

❑ Nous retenons que si p pointe sur T alors:

`*(p+i)` = **`p[i]`** = **`T[i]`** // valeurs

`p+i` = **`&p[i]`** = **`&T[i]`** // adresses

Les pointeurs en langage C

Exercice :

- ❑ Ecrire 2 procédures pour lire et afficher les éléments d'un tableau de réelles à l'aide d'un **pointeur** et de l'indice **i**:

```
void lecture_pti (float t[], int n);  
void affichage_pti (float t[], int n);
```

Les pointeurs en langage C

Lecture et affichage d'un tableau avec un pointeur et l'indice i :

```
void lecture (float t[], int n)
{ int i ;
  for(i=0 ; i<n ; i++)
  {
    printf("donner une valeur");
    scanf("%f", &t[i]) ;
  }
}

void affichage (float t[], int n)
{ int i ;
  for(i=0 ; i<n ; i++)
    printf("%f", t[i]);
}
```



```
void lecture_pti ( float t[], int n )
{ int i;
  float *pt ;
  pt = &t[0]; // ou pt=t
  for(i=0 ; i<n ; i++)
  {
    printf("donner une valeur");
    scanf("%f", pt+i) ; // ou &pt[i]
  }
}

void affichage_pti (float t[], int n)
{ int i;
  float *pt ;
  pt = &t[0]; // ou pt=t
  for(i=0 ; i<n ; i++)
    printf("%f", *(pt+i)); // ou pt[i]
}
```

Les pointeurs en langage C

Exercice :

- ❑ Réécrire les 2 procédures précédentes à l'aide d'un **pointeur** et sans utiliser l'indice **i**:

```
void lecture_pt (float t[], int n);  
void affichage_pt (float t[], int n);
```


Les pointeurs en langage C

Lecture et affichage d'un tableau avec un pointeur et sans utiliser i :

```
void lecture_pti ( float t[], int n )
{ int i;
  float *pt ;
  pt = &t[0]; // ou pt=t
  for(i=0 ; i<n ; i++)
  {
    printf("donner une valeur");
    scanf("%f", pt+i) ; // ou &pt[i]
  }
}

void affichage_pti (float t[], int n)
{ int i;
  float *pt ;
  pt = &t[0]; // ou pt=t
  for(i=0 ; i<n ; i++)
    printf("%f", *(pt+i)); // ou pt[i]
}
```



```
void lecture_pt(float t[], int n)
{
  float *pt ;
  for( pt=t; pt<t+n ; pt++)
  {
    printf("donner une valeur");
    scanf("%f", pt) ;
  }
}

void affichage_pt(float t[], int n)
{
  float *pt ;
  for( pt=t; pt<t+n; pt++)
    printf("%f", *pt);
}
```

Les pointeurs en langage C

Sources d'erreurs :

Un grand nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable.

Résumons :

`int A ;` //déclare une variable simple de type `int`

`A` désigne le contenu de `A`

`&A` désigne l'adresse de `A`

`int B[10] ;` déclare un tableau de 10 éléments de type `int`

`B` \Leftrightarrow `&B[0]` désigne l'adresse de la première composante de `B`.

`*(B+i)` \Leftrightarrow `B[i]` désigne le contenu de la composante `i` du tableau

`B+i` \Leftrightarrow `&B[i]` désigne l'adresse de la composante `i` du tableau

Les pointeurs en langage C

Exercice :

Soit P un pointeur qui 'pointe' sur un tableau A : `int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};`

`int *P;`

`P = A;`

Quelles valeurs ou adresses fournissent ces expressions sachant que **`&A[0]=2293480`** :

a) P

b) `*P+2`

c) `*(P+2)`

d) `&A[7]-P`

e) `P+1`

f) `&A[4]-3`

g) `A+3`

h) `*(P+*(P+8)-A[7])`

i) `(*P)++`

j) `*P++`

Les pointeurs en langage C

Solution :

Soit P un pointeur qui 'pointe' sur un tableau A : `int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};`

`int *P;`

`P = A;`

Quelles valeurs ou adresses fournissent ces expressions sachant que **&A[0]=2193480** :

a) `P` = **2193480**

b) `*P+2` = **14**

c) `*(P+2)` = **34**

d) `&A[7]-P` = **L'indice 7**

e) `P+1` = **L'adresse de A[1]**

f) `&A[4]-3` = **L'adresse de A[1]**

g) `A+3` = **L'adresse de A[3]**

h) `*(P+*(P+8)-A[7])` = **23**

i) `(*P)++` = **13**

j) `*P++` = **23**



La soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

La programmation modulaire : fonctions

Comme tous les langages, C permet de découper un programme en plusieurs parties nommées souvent "modules". Cette programmation dite "**modulaire**". Justifie pour multiples raisons :

1. Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte.
2. La programmation modulaire permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus d'argument permet de paramétrer certains modules.
3. La programmation modulaire permet le partage d'outils communs qu'ils suffit d'avoir écrits et mis au point une seule fois.

La programmation modulaire : fonctions

Fonction : Seule sorte de module existant en C :

Dans beaucoup de langages, on trouve deux sortes de "modules", à savoir :

- ☐ Les **fonctions**, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments qui correspondent à des informations qui lui sont transmises et elle fournit un unique résultat.
 - ☐ Les **procédures** (terme Pascal) ou **sous-programme** (terme Fortran ou Basic) qui élargissent la notion de fonction.
-
- En langage C, il y a **une seule sorte de sous-programmes** : les **fonctions**.
 - Une seule de ces fonctions **existe obligatoirement** ; c'est la **fonction principale main**.
 - Cette **fonction principale** peut, éventuellement, **appeler une ou plusieurs fonctions secondaires**.
 - De même, **chaque fonction secondaire** peut **appeler d'autres fonctions secondaires** ou **s'appeler elle-même** (dans ce dernier cas, on dit que la fonction est **récursive**).

La programmation modulaire : fonctions

Qu'est ce qu'une fonction ?

Une fonction est **un suite d'instruction** regroupées sous un nom; elle prend en entrée des **paramètres (arguments)** et retourne un **résultat**.

Une fonction est **écrite séparément** du corps de programme principal (**main**) et sera **appelée** par celui-ci lorsque cela sera nécessaire.



L'idée est de regrouper certaines tâches effectuées de manière courante ou répétée et de **créer une fonction** permettant **d'appeler la fonction** au lieu **d'écrire le même code** encore et encore pour différentes entrées.

La programmation modulaire : fonctions

Fonction : Seule sorte de module existant en C :

Pour élaborer une fonction C, il faut coder les instructions qu'elle doit exécuter, en respectant certaines règles syntaxiques.

Ce code source est appelé **définition de la fonction**.

Une définition de fonction spécifié :

- ❑ Le type de la **valeur renvoyée** par la fonction ;
- ❑ Le **nom** de la fonction ;
- ❑ Les **paramètres** (arguments) qui sont passés à la fonction pour y être traités ;
- ❑ Les **variables locales** et **externes** utilisés par la fonction ;
- ❑ Les **instructions** que exécute la fonction.

La programmation modulaire : fonctions

Arguments formels et arguments effectifs :

Arguments formels

- ❑ Les noms des arguments figurants dans l'en-tête de la fonction se nomment des arguments formels.
- ❑ Leur rôle est de permettre, au sien de la fonction, de décrire ce qu'il doit faire.

Arguments effectifs

- ❑ Les arguments fournis (transmise) lors de l'utilisation (l'appel) de la fonction se nomment des arguments effectifs.

La programmation modulaire : fonctions

L'instruction return :

- ❑ L'instruction `return` permet de retourner (renvoyer) la valeur précisée à l'instruction appelante.
- ❑ Elle **provoque une sortie immédiate** du bloc principal de la fonction.

Exemple :

La fonction CARRE du type double calcule et fournit comme résultat le carré d'un réel fourni comme paramètre.

```
double CARRE(double X)
{
    return X*X;
}
```

La programmation modulaire : fonctions

Définition et déclaration d'une fonction :

Définition d'une fonction

- ❑ La **définition d'une fonction** est la donnée du texte de son algorithme, qu'on appelle **corps de la fonction**. Elle est de la forme :

```
type_retour nom_fonction(type_1 arg_1,..., type_n arg_n) {  
    /*déclarations de variables locales*/  
    /*liste d'instructions*/  
    return (valeur_de_retour);  
}
```

Déclaration d'une fonction

- ❑ La **déclaration d'une fonction** se fait à l'aide d'un **prototype de fonction** utilisant des **paramètres formels typés** de la forme :

```
type_retour nom_fonction(type_1,..., type_n);
```

La programmation modulaire : fonctions

Appel d'une fonction :

- ❑ L'appel d'une fonction se fait par l'expression : *nom_fonction*(*param_1*,..., *param_n*)
- ❑ L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction.

Remarques :

- ❑ En C, une fonction ne peut retourner qu'une valeur (au plus).
- ❑ Le type de la fonction doit être le même que celui de la valeur retournée.
- ❑ La fonction appelante doit stocker ce résultat dans une variable de même type (ou bien ne rien stocker).
- ❑ Quand une fonction ne retourne pas de valeur elle est typée void.

Exemples : *void* main(); *void* AfficherBonjour();...

La programmation modulaire : fonctions

Exemple : Déclaration, définition et appel d'une fonction *(1er méthode)*

Déclaration de la fonction puissance :

```
double puissance (double, int );
```

Définition de la fonction puissance :

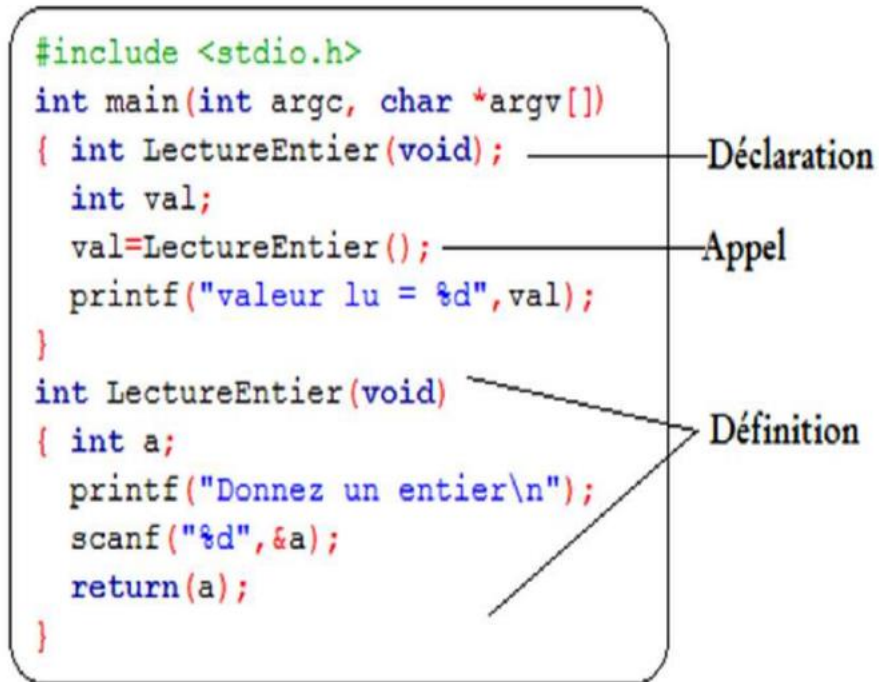
```
double puissance(double x, int n){  
    double y; /*type de la valeur de retour de la fonction*/  
    int i;  
    y=1;  
    for(i=1;i<=n;i++) y=y*x;  
    return(y);  
}
```

Appel de la fonction puissance :

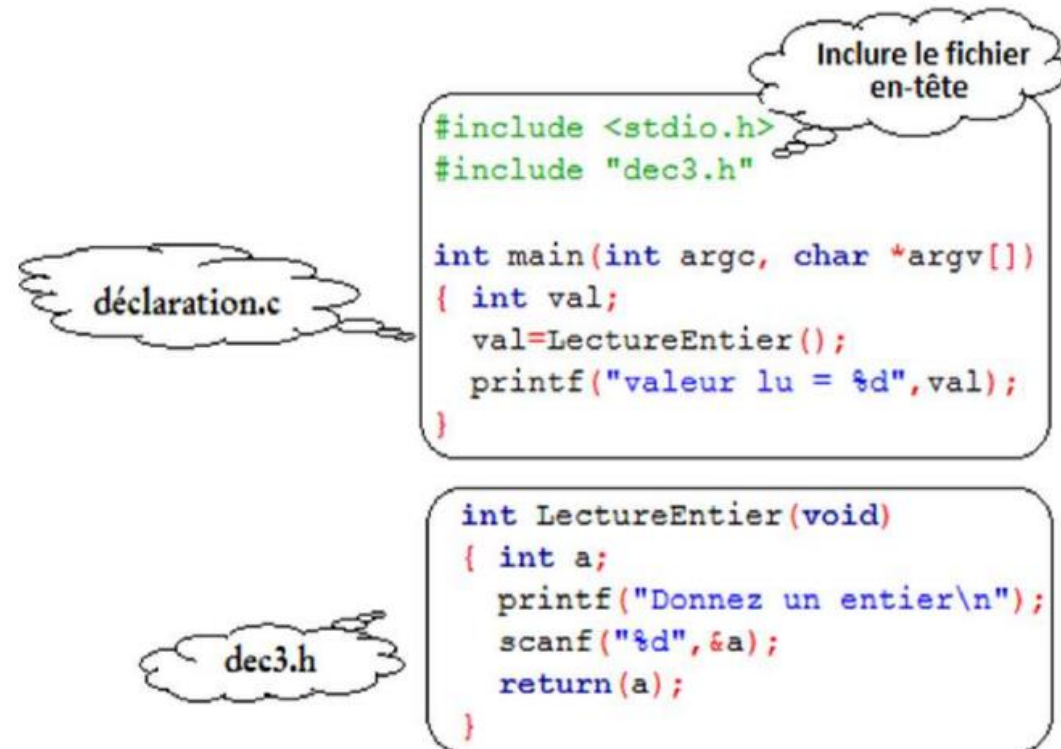
```
int main(){  
    double a=2.3;  
    int b=5;  
    printf(" %lf \n", puissance (a,b));  
}
```

La programmation modulaire : fonctions

(2ème méthode)



(3ème méthode)



Les *.h permettent de connaître les fonctions disponibles sans avoir à rentrer dans le code.

La programmation modulaire : fonctions

Types de fonctions :

Fonctions sans arguments et sans retour

```
void nom_fonction () {  
    Instruction;  
}
```

Fonctions avec retour et sans d'arguments

```
type_retour nom_fonction () {  
    Instruction;  
    return resultat;  
}
```

Fonctions sans de retour et avec arguments

```
void nom_fonction (type arg1, ...) {  
    Instruction;  
}
```

Fonctions avec retour et avec arguments

```
type_retour nom_fonction (type arg1, ...) {  
    Instruction;  
    return resultat;  
}
```

La programmation modulaire : fonctions

Fonctions sans arguments et sans retour :

Exemple :

```
#include <stdio.h>

int main()
{
    void FoncSansRtSansAr(void);
    FoncSansRtSansAr();    /* Appel */
}

void FoncSansRtSansAr(void)
{ printf("Pas de retour pas d\'arguments\n"); }
```


La programmation modulaire : fonctions

Fonctions sans de retour et avec arguments :

Exemple :

```
#include <stdio.h>
int main()
{ /*déclaration d'une fonction sans valeur de retour*/
    void FoncSansRt(int x);
    int x=5;
    FoncSansRt(x); /* Appel fonction */
}
/* Définition de de la fonction FoncSansRt */
void FoncSansRt(int a)
{
    int i;
    for(i=1;i<a;i++)
    { printf(" Pas de retour \n"); }
}
```

La programmation modulaire : fonctions

Fonctions avec retour et sans d'arguments :

Exemple d'une fonction qui renvoie un nombre aléatoire !

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ /*déclaration d'une fonction sans arguments*/
double FoncSansAr(void);
double alea;
alea=FoncSansAr(); /* Appel fonction */
printf("Un nombre aleatoire %lf\n",alea);
}
/* Définition de de la fonction FoncSansAr */
double FoncSansAr(void)
{
double a;
a=rand();
return(a);
}
```

La programmation modulaire : fonctions

Fonctions avec retour et avec arguments :

Exemple : Fonction puissance

```
#include <stdio.h>
{ /*déclaration d'une fonction avec argument et valeur de retour*/
    int puissance (int A) {
        int P;
        P = A * A;
        return P;
    }
int main() {
    int A, P;
    printf(" Veuillez saisir la valeur de A :");
    scanf("%d", &A);
    P = puissance(A);
    printf("La puissance de %d est : %d", A,P);
    return 0;
}
```

La programmation modulaire : fonctions

Exercices :

- ☐ Déclarez et appelez une fonction qui calcul la valeur absolue d'un entier.

- ☐ On dispose de 4 entiers.
 - ☐ Calculez l'entier le plus grand.
 - ☐ Pour ce faire : Déclarez et appelez plusieurs fois une fonction qui compare deux entiers et qui renvoie le plus grand.

La programmation modulaire : fonctions

Calcul la valeur absolue d'un entier :

```
1  #include <stdio.h>
2  int main() {
3      int ABS1(int); // Déclaration
4      int n;
5      printf("donnez un entier \n");
6      scanf("%d", &n);
7      printf("|%d|=%d\n", n, ABS1(n)); // Appel
8  }
9  // Définition
10 int ABS1(int b) {
11     if(b >= 0)
12         return(b);
13     else
14         return(-b);
15 }
16
```

La programmation modulaire : fonctions

Calculez l'entier le plus grand :

```
1  #include <stdio.h>
2  int main()
3  {
4      int COMP1(int, int); // Déclaration
5      int a,b,c,d,max;
6      printf("donnez a, b, c et d\n");
7      scanf("%d%d%d%d",&a,&b,&c,&d);
8      max=COMP1(COMP1(a,b),COMP1(c,d)); // Appel
9      printf("max(%d,%d,%d,%d) = %d\n",a, b, c, d, max);
10 }
11 int COMP1(int m, int n) {
12     if(m>n)
13         return(m);
14     else
15         return(n);
16 }
```

La programmation modulaire : fonctions

Paramètres d'une fonction :

- ❑ Les paramètres ou les arguments sont les "boîtes aux lettres" d'une fonction.
- ❑ Les fonctions acceptent les données de l'extérieur et déterminent les actions et les résultats.
- ❑ Nous pouvons résumer le rôle des paramètres en C de la façon suivante :
 - ❑ Les paramètres d'une fonction sont simplement des variables locales ou des variables globales, qui sont initialisées par les valeurs obtenues lors de l'appel ou passage.
 - ❑ Il existe deux types de passage d'arguments :
 - Passage par valeur
 - Passage par adresse dit aussi "par référence"

La programmation modulaire : fonctions

Variables globales :

- ❑ Une **variable globale** est déclarée en dehors de toute fonction. En générale, il est déclarée immédiatement derrière les **#include**.
- ❑ Une variable globale est **connue du compilateur** dans toute la portion de **code qui suit sa déclaration**.
- ❑ Les variables globales sont **systématiquement permanentes**.

Exemple : variable globale à une durée de vie permanente

int n; // n est une variable globale

void fonction();

void fonction() {

n++; printf("appel numero %d \n",n); }

int main(){

for (int i=0; i<3; i++) fonction();}

Ce programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
```


La programmation modulaire : fonctions

Variables globales :

- ❑ Un programme peut avoir le même nom pour les variables locales et globales, mais la valeur de la variable locale dans une fonction aura la priorité.

Exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int a; // variable global
4  void fonction_1() {
5      int a=3;
6      printf("Fonction 1");
7      printf("\n a=%d",a); }
8
9  void fonction_2() {
10     printf("\n Fonction 2");
11     printf("\n a=%d\n", a); }
12 int main(void) {
13     fonction_1();
14     fonction_2();
15     printf("bye");
16     return 0;
17 }
```

```
Fonction 1
a=3
Fonction 2
a=0
bye
Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

La programmation modulaire : fonctions

Variables locales :

- ❑ On appelle **variable locale** une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme.
- ❑ Par défaut, les **variables locales sont temporaires**.
- ❑ Les variables locales n'ont en particulier **aucun lien avec des variables globales de même nom**.

Exemple : variable locale à une durée de vie limitée à une seule exécution

```
int n; /*une variable globale*/

void fonction();

void fonction(){
    int n=0; /*une variable locale à la fonction*/
    n++; printf("appel numero %d \n",n);}

int main() {
    for (int i=0; i<3; i++) fonction();}
```

Ce programme affiche :

```
appel numero 1
appel numero 1
appel numero 1
```

La programmation modulaire : fonctions

Exercice : Fonctions - variables locales

- ❑ Ecrire un programme qui demande à l'utilisateur de saisir les valeurs de deux variables A et B (**locales**). Ensuite, il permet de définir et d'appeler les fonctions suivantes:
 - Une fonction qui retourne si les valeurs de A et B sont de même signe ou non (Une fonction sans valeur de retour et avec arguments)
 - Une fonction qui renvoie le minimum de A et B. (Une fonction avec une valeur de retour et avec arguments)
 - Une fonction qui renvoie le maximum de A et B. (Une fonction avec une valeur de retour et avec arguments)

La programmation modulaire : fonctions

Exercice : Fonctions - variables locales

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void signe (int A, int B) {
4      if (A*B > 0)
5          printf("Les valeurs de A et B on le meme signe \n");
6      else
7          printf("Les valeurs de A et B on des signes differents \n");}
8  int minimum (int A, int B) {
9      int min;
10     if (A>B) min = B; else min = A;
11     return min; }
12  int maximum (int A, int B) {
13     int max;
14     if (A>B) max = A; else max = B;
15     return max; }
16  int main() {
17     int max, min;
18     int A, B;
19     printf("Veuillez saisir la valeur de A : ");
20     scanf("%d", &A);
21     printf("Veuillez saisir la valeur de B : ");
22     scanf("%d", &B);
23     signe(A,B);
24     min = minimum(A,B);
25     max = maximum(A,B);
26     printf("La valeur minimale est : %d \n", min);
27     printf("La valeur maximale est : %d \n", max);
28     return 0;
29 }
30
```

La programmation modulaire : fonctions

Exercice : Fonctions - variables globales

- ❑ Ecrire un programme qui demande à l'utilisateur de saisir les valeurs de deux variables A et B (**globales**). Ensuite, il permet de définir et d'appeler les fonctions suivantes:
 - Une fonction qui retourne si les valeurs de A et B sont de même signe ou non (**Une fonction sans valeur de retour et avec arguments**)
 - Une fonction qui renvoie le minimum de A et B. (**Une fonction avec une valeur de retour et avec arguments**)
 - Une fonction qui renvoie le maximum de A et B. (**Une fonction avec une valeur de retour et avec arguments**)

La programmation modulaire : fonctions

Exercice : Fonctions - variables globales

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int A, B;
4  void signe () {
5      if (A*B > 0)
6          printf("Les valeurs de A et B on le meme signe \n");
7      else
8          printf("Les valeurs de A et B on des signes differents \n");}
9  int minimum () {
10     int min;
11     if (A>B) min = B; else min = A;
12     return min; }
13 int maximum () {
14     int max;
15     if (A>B) max = A; else max = B;
16     return max; }
17 int main() {
18     int max, min;
19
20     printf("Veuillez saisir la valeur de A : ");
21     scanf("%d", &A);
22     printf("Veuillez saisir la valeur de B : ");
23     scanf("%d", &B);
24     signe(A,B);
25     min = minimum(A,B);
26     max = maximum(A,B);
27     printf("La valeur minimale est : %d \n", min);
28     printf("La valeur maximale est : %d \n", max);
29     return 0;
30 }
```

La programmation modulaire : fonctions

Passage des paramètres par valeur :

- ❑ Lorsqu'on ne veut pas changer la valeur d'un paramètre on parle de **transmission par valeur**, c'est-à-dire la fonction **travaille alors uniquement sur une copie de la valeur**.
- ❑ Cette copie **disparaît** lors du retour au programme appelant.
- ❑ Si la fonction **modifie la valeur d'un de ses paramètres**, seule la copie sera modifiée ; la variable du programme appelant, elle, **ne sera pas modifiée**.

Exemple :

```
1  #include <stdio.h>
2
3  void echange (int a, int b){
4  int tmp;
5  printf("debut fonction : \n a=%d \t b=%d \n",a,b);
6  tmp=a; a=b; b=tmp;
7  printf("fin fonction : \n a=%d \t b=%d \n",a,b);}
8  int main(){
9  int a=2, b=5;
10 printf("debut programme principal : \n a=%d \t b=%d \n",a,b);
11 echange(a,b);
12 printf("fin programme principal : \n a=%d \t b=%d \n",a,b);}
13
```

```
debut programme principal :
a=2    b=5
debut fonction :
a=2    b=5
fin fonction :
a=5    b=2
fin programme principal :
a=2    b=5
```

La programmation modulaire : fonctions

Passage des paramètres par adresse dit aussi "par référence" :

- ❑ Lorsqu'on veut se permettre la possibilité de changer la valeur on parle de **transmission par adresse**.
- ❑ Les paramètres formels doivent être **des pointeurs** ; les paramètres effectifs doivent être **des adresses (noms de variables précédés de &)**.

Exemple :

```
1  #include <stdio.h>
2
3  void echange (int *adr_a, int *adr_b){
4  int tmp;
5  tmp=*adr_a; *adr_a=*adr_b; *adr_b=tmp;}
6  int main(){
7  int a=2, b=5;
8  printf("debut programme principal :\n a=%d \t b=%d \n",a,b);
9  echange(&a,&b);
10 printf("fin programme principal :\n a=%d \t b=%d \n",a,b);}
11
```

```
debut programme principal :
a=2    b=5
fin programme principal :
a=5    b=2
```


La programmation modulaire : fonctions

Passage de l'adresse d'un tableau à une dimension :

- ❑ On est souvent amené à parcourir les éléments d'un tableau pour **faire des opérations** (moyenne, classement, ...).
- ❑ Les fonctions sont **indispensables** pour ne pas avoir à **réécrire un nouveau code** pour chaque tableau.
- ❑ Un **tableau** peut donc être **passé comme argument** d'une fonction.
- ❑ On **transmet** à la fonction **l'emplacement mémoire** du début du tableau.
- ❑ On parlera de **l'adresse du tableau** qui est égale à **l'adresse du 1er élément du tableau** : il s'agit du **nom du tableau** (int tab[10] ; tab est l'adresse du tableau (et de tab[0])).
 - ❑ Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le **nom suivi de crochets** vide : type nom_tab[].
 - ❑ ou simplement par **un pointeur** sur les éléments du tableau : type ***nom_tab**.

La programmation modulaire : fonctions

Fonctions et tableaux : déclaration, utilisation, définition

Exemple

```
1  #include <stdio.h>
2  int main()
3  {
4      void AffichInv(int T[], int n);
5      int tab[]={2, -5, 3, 2, 7, 4};
6      AffichInv(tab,6);
7  }
8  void AffichInv(int T[], int n)
9  {
10     if(n==1)
11     printf ("%d, ", T[0]);
12     else
13     {
14         printf ("%d, ", T[n-1]);
15         AffichInv(T,n-1);
16     }
17 }
18
```

La programmation modulaire : fonctions

Fonctions et tableaux : déclaration, utilisation, définition

Exemple

```
1  #include <stdio.h>
2  int main()
3  {
4  void AffichInv(int *T, int n);
5  int tab[]={2, -5, 3, 2, 7, 4};
6  AffichInv(tab,6);
7  }
8  void AffichInv(int *T, int n)
9  {
10 if(n==1)
11 printf ("%d, ", *T);
12 else
13 {
14 printf ("%d, ", *(T+n-1));
15 AffichInv(T,n-1);
16 }
17 }
18
```

La programmation modulaire : fonctions

Exercices :

Tableau de multiplication

- ❑ Ecrire un programme utilisant une fonction qui affiche le **tableau de multiplication** d'un **entier positif x**, pour les dix premiers numéros.

```
1  #include <stdio.h>
2  ▼ void TMultiplication (int N) {
3      int i;
4      printf("La table de multiplication de %d est : \n",N);
5      for (i = 0; i <= 10; i++)
6      {
7          printf("%d x %d = %d \n",N, i, N*i);
8      }
9  }
10 int main()
11 ▼ {
12     int N;
13     printf("veuillez entrer la valeur de N : ");
14     scanf("%d", &N);
15     TMultiplication(N);
16     return 0;
17 }
```

Exercices :

Exercice 1:

Ecrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A. Utiliser le formalisme pointeur à chaque fois que cela est possible.

Exercices :

Exercice 1:

```
#include <stdio.h>
main()
{ int N, M ;
  int *PA, *PB ;
  int A[50], B[50] ;

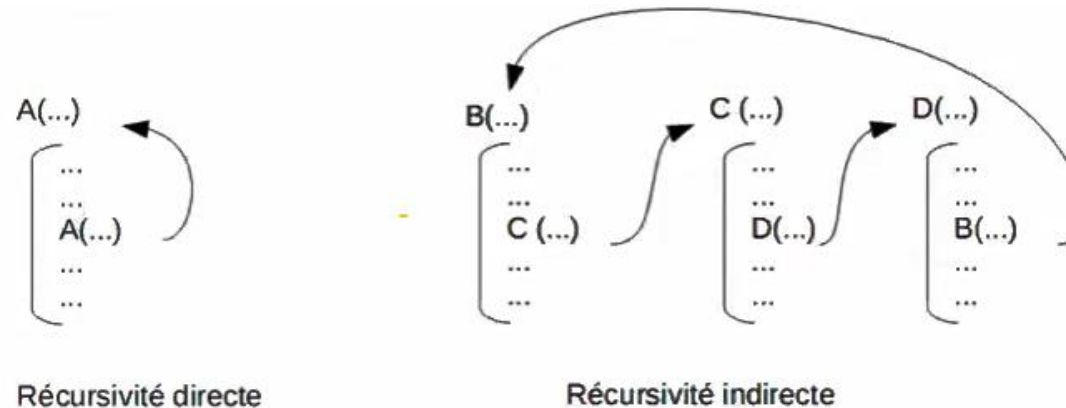
  printf("\n Donner N et M ") ;
  scanf("%d%d", &N, &M) ;

  for(PA=A ; PA<A+N ; PA++)
  {
    printf("\n Donner A[%d]: ", PA-A) ;
    scanf("%d", PA) ;
  }
```

```
for(PB=B ; PB<B+M ; PB++)
{
  printf("\n Donner B[%d]: ", PB-B) ;
  scanf("%d", PB) ;
}
for(PA=A+N, PB=B ; PA<A+N+M, PB<B+M ; PA++, PB++)
  *PA=*PB ;
/*      Affichage du tableau résultat      */
for(PA=A ; PA<A+N+M ; PA++)
  printf("\n A[%d]= %d ", PA-A, *PA) ;
}
```

La Récursivité

- ❑ Une **construction est récursive** si elle se définit à partir d'elle-même.
- ❑ On appelle **récursive** toute fonction ou procédure qui **s'appelle elle-même**, directement ou indirectement.
- ❑ La programmation récursive est une technique de programmation qui **remplace les instructions de boucle par des appels de fonction**.
- ❑ Moyen simple et élégant de résoudre certains problèmes qui utilisent des **fonctions** et qui ne peuvent être définies que **récursivement**.



- **Récursivité directe**: Appel au même module directement.
- **Récursivité indirecte**: Appel au même module indirectement.

La Récursivité

Condition d'arrêt et appel récursif :

Condition d'arrêt :

- ❑ Puisqu'une fonction récursive s'appelle elle-même, il est impératif qu'on prévoit une **condition d'arrêt à la récursion**, sinon le programme ne s'arrête jamais!
- ❑ On doit toujours **tester en premier la condition d'arrêt**, et ensuite, si la condition n'est pas vérifiée, **lancer un appel récursif**.

Appel récursif :

- ❑ Dans un algorithme récursif, on nomme **appel récursif** toute étape de l'algorithme résolvant le même problème sur une autre donnée.

La Récursivité

Ecrire un algorithme récursif :

Algorithme récursif :

- ❑ Un algorithme de résolution d'un problème P sur une donnée A est dit récursif si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée B.
- ❑ Pour écrire un algorithme récursif pour un problème réalisant un certain traitement T sur des données D. On procède de la manière suivante :
 - ❑ Décomposer le traitement T en sous traitements de même nature mais sur des données plus petites.
 - ❑ Trouver la condition d'arrêt.
 - ❑ Tester éventuellement sur un exemple.
 - ❑ Ecrire l'algorithme.

La Récursivité

Ecrire un algorithme récursif :

Exemple

- ❑ Ecrire une fonction récursive qui **recherche par dichotomie** un élément dans un tableau d'entiers. La fonction renvoie l'indice de l'élément s'il existe et -1 sinon.

Solution

Décomposition du traitement :

- ❑ Rechercher un élément dans le **tableau** va conduire, si **on ne trouve pas l'élément au milieu**, à relancer la recherche sur une **moitié du tableau**, puis sur un **quart**, etc.
- ❑ A chaque appel récursif, il faut donc **savoir entre quels indices i et j on cherche l'élément**.

Condition d'arrêt :

- ❑ La **recherche s'arrête** quand on trouve l'élément ou quand il n'y a plus de case où chercher ($i > j$).

La Récursivité

Schéma d'un module récursif :

module **RECURSIF** (**PARAMÈTRES**)

début

si **TEST_D'ARRET** alors

 début

 instructions en cas d'arrêt

 fin

} partie 1: si

sinon

 début

 instructions

RECURSIF (**PARAMÈTRES CHANGÉS**) // appel récursif

 instructions

 fin

} partie 2: sinon

finsi

fin

La Récursivité

Types de récursivité :

Récursivité simple ou linéaire :

Une fonction **simplement réursive**, c'est une fonction **s'appelle elle-même une seule fois**.

Exemple : la fonction factorielle

Ecrire la fonction itérative qui retourne le factoriel d'un entier n .

La Récursivité

Exemple : la fonction factorielle

La fonction itérative en c :

```
#include<stdio.h>

int fact_iter (int n){
int i, f;
f = 1;
for(i = 1; i<=n; i++)
    f = f * i;
return f;
}
```

$n! = n * (n-1)!$ pour $n > 0$

La fonction récursive correspondante :

```
#include<stdio.h>

int fact_recu (int n){
int f;
if(n == 0) f = 1; // test d'arrêt
else f = n * fact_recu (n-1); // cas général
return f;
}
```

et $n! = 1$ si $n = 0$

La Récursivité

Exemple : la fonction factorielle

Explications :

- Déroulement de l'algorithme pour $n=4$:
- D'abord, on compare n au test d'arrêt (0).

1- $4! = 4 * 3!$
2- $3! = 3 * 2!$
3- $2! = 2 * 1!$
4- $1! = 1 * 0!$
5- $0! = 1$ (test d'arrêt)



Faire un
retour en
arrière
pour
calculer $4!$

```
int fact_recu (int n){  
    int f;  
    if(n == 0) f = 1; // test d'arrêt  
    else f = n * fact_recu (n-1);  
    return f;  
}
```

- L'exécution de n attend la terminaison de l'exécution de $n-1$ pour continuer son traitement.

La Récursivité

Récursivité multiple :

- ❑ Un programme récursif est **multiple** si l'un des cas qu'il traite se résout avec **plusieurs appels récursifs**.
- ❑ Dans le cas où il y a **deux appels récursifs** on parle de récursivité **binaire**.

Exemple : la suite de Fibonacci

Un exemple de récursivité multiple est la suite de Fibonacci : $U_0 = U_1 = 1$ et $U_n = U_{n-1} + U_{n-2}$, pour $n \geq 2$

```
int fibonacci(int n)
{
    if ((n==0) || (n==1))
        return (1);
    else
        return (fibonacci(n-1)+fibonacci(n-2));
}
```

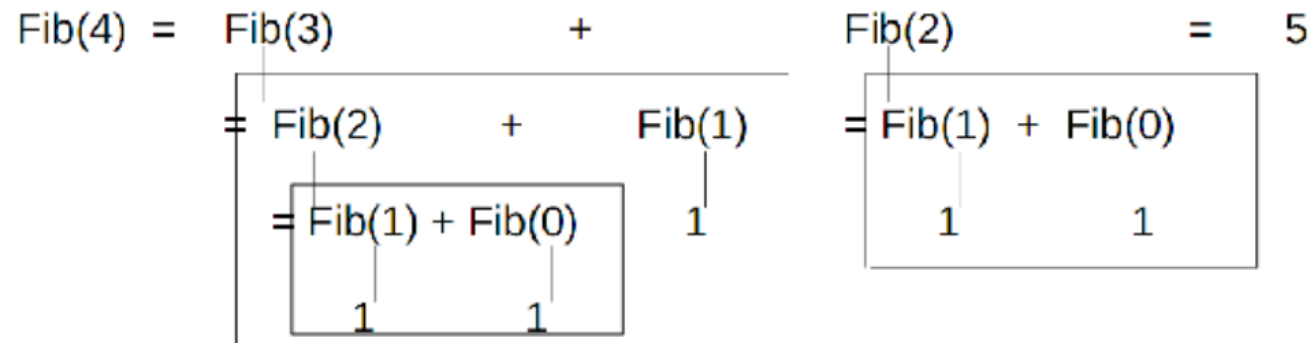
La Récursivité

Exemple : la suite de Fibonacci

$U_0 = U_1 = 1$ et $U_n = U_{n-1} + U_{n-2}$, pour $n \geq 2$

Déroulement de programme :

```
int fib_recu(int n){  
    int f;  
    if(n < 2)  f = 1;  
    else f=fib_recu(n-1)+ fib_recu(n-2) ;  
    return f;  
}
```



La Récursivité

Exercice :

Soit la procédure `compter_iter` suivante, qui affiche "bonjour" n fois.

Version itérative:

```
void compter_iter (int n) {  
    int i;  
    for (i = 0 ; i < n; i++)  
        printf(" bonjour ");  
}  
  
int main(){  
    int n;  
    printf("Entrer un entier ");  
    scanf("%d",&n);  
    compter_iter(n);  
    compter_recu(n);  
    return 0;  
}
```

Version récursive:

```
void compter_recu ( int n) {  
    // test d'arrêt implicite: n <= 0  
    if (n > 0) {  
        printf(" bonjour ");  
        compter_recu (n-1); //boucle de n à 1  
    }  
}
```

Transformer cette procédure itérative en une procédure récursive.

La Récursivité

Récursivité mutuelle :

- ❑ La récursivité **mutuelle ou croisée** consiste à écrire des fonctions qui **s'appellent l'une l'autre**.
- ❑ Autrement dit, une **fonction f** peut faire référence à une **fonction g** qui elle-même fait référence à **f**.

Exemple : parité d'un entier

Un exemple très classique de récursivité croisée est l'évaluation de la parité d'un nombre :

```
// n est pair ssi (n-1) est impair
int Pair(int n)
{
    if(n == 0)
        return (1);
    else
        return(Impair(n-1));
}
// n est impair ssi (n-1) est pair
int Impair(int n)
{
    if(n == 0)
        return (0);
    else
        return(Pair(n-1));
}
```

La Récursivité

Récursivité imbriquée :

- Une fonction récursive dont l'un des paramètres est un appel à elle-même est qualifiée de fonction récursive imbriquée.

Exemple : la fonction d'Ackerman

La fonction d'Ackermann est définie récursivement sur $\mathbb{N} \times \mathbb{N}$ comme suit :

$$Ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ack(m - 1, 1) & \text{si } n = 0 \\ Ack(m - 1, Ack(m, n - 1)) & \text{si } n \neq 0 \text{ et } m \neq 0 \end{cases}$$

```
int ack(int m, int n)
{
    if(m == 0)
        return(n + 1);
    else
        if(n == 0)
            return(ack (m - 1, 1));
        else
            return(ack (m - 1,ack (m, n - 1)));
}
```

La Récursivité

Récursivité terminale :

- ❑ On dit qu'une fonction est **réursive terminale**, si tout appel récursif est de la forme `return f(...)` (si aucune instruction n'est exécutée après l'appel de la fonction à elle-même).
- ❑ Autrement dit, la **valeur retournée est directement la valeur obtenue par un appel récursif**, sans qu'il n'y ait aucune opération sur cette valeur.

Exemple :

La forme réursive **terminale** de la factorielle.

Fact(5,1)--> 1 --> 5 --> 20 --> 60 --> 120

```
1  #include <stdio.h>
2  int main()
3  {
4      int Fact(int n, int res);
5      int n,p;
6      printf("saisir n \n");
7      scanf("%d",&n);
8      p=Fact(n,1);
9      printf("%d!=%d \n",n,p);
10 }
11 int Fact(int n,int res)
12 {
13     if(n==1)
14         return(res);
15     else {
16         printf("res=%d \n",res);
17         return(Fact(n-1,n*res));
18     }
19 }
20
```

La Récursivité

Récursivité non terminale :

- ❑ Une fonction récursive est dite **non terminale** si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).

Exemple

La forme récursive **non terminale** de la factorielle.

Fact(5)-->1--> 2 --> 6 --> 24 --> 120

```
1  #include <stdio.h>
2  int main()
3  {   int Fact(int n);
4      int n,p;
5      printf("saisir n \n");
6      scanf("%d",&n);
7      p=Fact(n);
8      printf("%d!=%d \n",n,p);
9  }
10 int Fact(int n)
11 {   int res;
12     if (n==1) {
13         res=1;
14         return(res); }
15     else {
16         res=n*Fact(n-1);
17         printf("res=%d \n",res);
18         return(res);
19     }
20 }
21
```

Allocation dynamique de la mémoire et Structure

Allocation dynamique de la mémoire :

- ❑ La déclaration d'un tableau définit un tableau "statique" (il possède un nombre figé d'emplacements). il y a donc un gaspillage d'espace mémoire en réservant toujours l'espace maximal prévisible.
- ❑ L'avantage de l'allocation dynamique par rapport à la déclaration de variables est de permettre d'adapter, lors de l'exécution du programme, la consommation de mémoire à la taille effective des données traitées.
- ❑ Il serait souhaitable que l'allocation de la mémoire dépende du nombre d'éléments à saisir. Ce nombre ne sera connu qu'à l'exécution : c'est l'allocation dynamique.

Allocation dynamique de la mémoire et Structure

Exemple :

❑ Création et remplissage d'un tableau:

```
int T[50]; // Déclaration d'un tableau avec une taille max=50
printf("Entrer la taille du tableau (max.50):");
scanf("%d", &n ); // Lire la taille n du tableau
for(i=0; i<n; i++) {
    printf("donner une valeur");
    scanf("%d", &T[i]) ; // Remplissage du tableau
}
```

Remarque:

sizeof(int)*(50-n) Octets seront perdus.

Allocation dynamique de la mémoire et Structure

Malloc :

- ❑ L'allocation dynamique en C se fait par l'intermédiaire de la fonction **malloc** de la librairie standard **stdlib.h**:

`char* malloc (nombreOctets)`

- ❑ Par défaut, cette fonction retourne un `char *` pointant vers un espace mémoire de taille **nombreOctets**.
- ❑ Il faut convertir la sortie de **malloc** à l'aide d'un **cast**, pour des pointeurs qui ne sont pas des `char *`.

Exemple :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main() {
4      int i=3, *p;
5      p=(int*) malloc (sizeof(int)); // la valeur de p est donnée par le système.
6      *p=i; // Cette instruction permet de donner 3 à *p. Si on la supprimer , *p=0
7      printf("i se trouve a l'adresse %d et elle contient %d \n ", &i, i);
8      printf("p se trouve a l'adresse %d et elle contient %d \n ", &p, p);
9      printf("*p se trouve a l'adresse %d et elle contient %d \n ", &(*p), *p);
10 }
```


Allocation dynamique de la mémoire et Structure

Calloc :

- ❑ L'allocation dynamique en C peut se faire aussi par la fonction **calloc** (contiguous allocation) de la librairie standard **stdlib.h**:

```
void * calloc (size nb_bloc, size taille);
```

- ❑ Deux arguments : nombre d'octets à allouer.
 - ❑ **nb_bloc** : nombre de blocs consécutifs à allouer,
 - ❑ **taille** : nombre d'octets par bloc.
- ❑ Une valeur de retour du type **void *** (pointeur sur void) qui correspond:
 - ❑ À l'adresse de l'emplacement alloué si tout se passe bien,
 - ❑ Au pointeur NULL en cas de problème.

Allocation dynamique de la mémoire et Structure

Exemple :

Allocation de la mémoire pour un tableau (1D) de 5 doubles.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main() {
4      double *ptr = NULL ;
5      ptr = (double *) calloc (5, sizeof(double)) ;
6      for (int i=0; i<5; i++)
7          printf("ptr+%d = %lf \n", i, *(ptr+i));
8  }
```

```
ptr+0 = 0.000000
ptr+1 = 0.000000
ptr+2 = 0.000000
ptr+3 = 0.000000
ptr+4 = 0.000000
```

- A la différence de `malloc()`, `calloc()` alloue un tableau de `nb_bloc` éléments ayant chacun `taille`.
- La mémoire allouée est initialisée à 0 avec les 2 fonctions.

Allocation dynamique de la mémoire et Structure

Realloc :

Il arrive fréquemment qu'un bloc alloué n'ait pas la taille suffisante pour accueillir de nouvelles données. La fonction `realloc` est utilisée pour changer (agrandir ou réduire) la taille d'une zone allouée par `malloc`, `calloc` ou `realloc`.

```
void * realloc (void * ancien_bloc, size nouvelle_taille);
```

- ❑ si `ancien_bloc==0`, l'appel est équivalent à `malloc(nouvelle_taille)`.
- ❑ si `nouvelle_taille==0`, l'appel est équivalent à `free(ancien_bloc)`.

En cas de succès:

- ❑ `realloc` alloue un espace mémoire de taille `nouvelle_taille`, copie le contenu pointé par `ancien_bloc` dans ce nouvel espace (en tronquant si la nouvelle taille est inférieure à la précédente),
- ❑ puis libère l'espace pointé et retourne un pointeur vers la nouvelle zone mémoire.

En cas d'échec, cette fonction ne libère pas l'espace mémoire actuel, et retourne une adresse `nulle`.

Allocation dynamique de la mémoire et Structure

Exemple :

Ce qu'il ne faut pas faire: `ptr = realloc(ptr, newsize);`

Ce qu'il faut faire:

```
void *p=NULL;
p = realloc(ptr, 10*sizeof(double));
if (p != NULL)
    ptr = (double *)p;
else
    printf("%d", p);
```

- On utilise une nouvelle variable p.
- Si la réallocation se passe bien, donc l'affectation est sûre.
- Sinon si la réallocation échoue, le processus ne peut pas continuer mais la mémoire ptr est encore utilisée.
- A la fin, on peut aussi libérer la mémoire de ptr avec `free(ptr)`

Allocation dynamique de la mémoire et Structure

Exemple complet :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main() {
4      double *ptr = NULL ;
5      ptr = (double *) calloc (5, sizeof(double)) ;
6      for (int i=0; i<5; i++){
7          *(ptr+i)=i;
8          printf("ptr+%d = %lf \n", i, *(ptr+i));
9      }
10     printf("\n Nouveau tableau \n");
11     double *p=NULL;
12     p = (double *) realloc (ptr, 10*sizeof(double));
13     if (p != NULL) {
14         ptr = p;
15         for (int i=0; i<10; i++)
16             printf("ptr+%d = %lf \n", i, *(ptr+i));
17     }
18     else
19         printf("%d", p);
20     system("pause");
21 }
```

```
ptr+0 = 0.000000
ptr+1 = 1.000000
ptr+2 = 2.000000
ptr+3 = 3.000000
ptr+4 = 4.000000
```

```
Nouveau tableau
ptr+0 = 0.000000
ptr+1 = 1.000000
ptr+2 = 2.000000
ptr+3 = 3.000000
ptr+4 = 4.000000
ptr+5 = 0.000000
ptr+6 = 0.000000
ptr+7 = 0.000000
ptr+8 = 0.000000
ptr+9 = 0.000000
```

Allocation dynamique de la mémoire et Structure

Libération de la mémoire :

Définition:

- ❑ C'est l'opération qui libère l'espace-mémoire alloué.
- ❑ En C, la libération de la mémoire se fait par l'intermédiaire de la fonction de la librairie standard `stdlib.h` :

`void free (nomPointeur);`

- ❑ Tout espace-mémoire alloué dynamiquement via `malloc` (ou équivalent) doit obligatoirement être désalloué par `free`, sinon nous rencontrons le fameux problème de fuite mémoire (memory leak), qui est une occupation croissante et non contrôlée ou non désirée de la mémoire d'un ordinateur.

Allocation dynamique de la mémoire et Structure

Allocation dynamique, saisie et affichage des valeurs d'un tableau :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main() {
4      short *pt ;
5      int N, i ;
6      printf("Entrer la taille N du tableau\n");
7      scanf("%d", &N) ;
8      pt=(short*)calloc(N, sizeof(short)) ; // pt=(short*)malloc(N*sizeof(short)) ;
9      if(pt==NULL) {
10         printf("Mémoire non disponible");
11         system("pause") ;
12         return 1 ;
13     }
14     printf("Remplissage du tableau :") ;
15     for(i=0 ;i<N ;i++)
16         scanf("%d", pt+i) ;
17     printf("Affichage du tableau ") ;
18     for(i=0 ;i<N ;i++)
19         printf("%d\t", *(pt+i)) ;
20     free(pt);
21     return 0;
22 }
```

Allocation dynamique de la mémoire et Structure

Les structures en C :

Définition d'une structure

- ❑ Une **structure** ou un **enregistrement** permet de **grouper plusieurs variables** de types différents dans une **seule variable** (c'est la notion de table dans les bases de données).

Exemple : compte = {nbanque, ncompte, solde}

compte
#nbanque
#ncompte
solde

La structure compte :

2 enregistrements



nbanque	ncompte	solde
1	10100	2000,00
1	10101	3000,00

Allocation dynamique de la mémoire et Structure

Caractéristiques d'une structure :

- ❑ Une structure est composée d'un nombre fixe de **champs** caractérisés par :
 - ❑ Des **noms**: nbanque, ncompte, solde.
 - ❑ Et des **types**: **int** pour nbanque et ncompte, **float** pour solde.

compte
#nbanque
#ncompte
solde

Donc, des champs de types différents peuvent être groupés à l'intérieur d'une structure.

Allocation dynamique de la mémoire et Structure

Déclaration d'une structure :

- ❑ La déclaration du type structure se fait par le mot-clé **struct**.

Syntaxe

```
struct mastruct {  
    type1  champ1;  
    :  
    typeN  champN;  
};
```

- ❑ On déclare une nouvelle structure,
 - de nom **mastruct**,
 - de champs nommés: **champ1** à **champN**,
 - les champs ont pour types: **type1** à **typeN**.
- ❑ **mastruct** et **champ1** à **champN** doivent être des identificateurs càd des noms de variables.

Allocation dynamique de la mémoire et Structure

Exercice :

- ☐ Déclarer la structure compte avec nbanque, ncompte et solde.
- ☐ Quel est le nombre d'octets nécessaire pour stocker une variable de type structure compte?

NB: sizeof(int)= 4 O et sizeof(float)= 4 O.

Solution:

```
Exemple
struct compte {
    int    nbanque;
    int    ncompte;
    float  solde;
};
```

Nombre d'octets= 4+4+4=12 O

Allocation dynamique de la mémoire et Structure

Déclaration de variables de type structure :

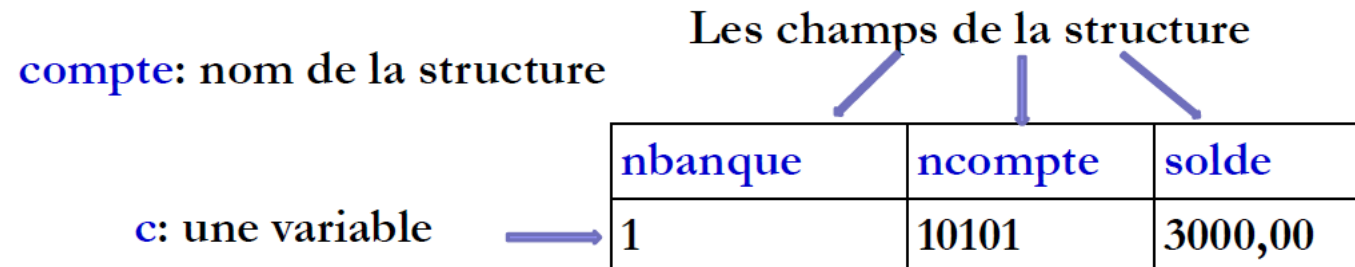
Syntaxe

```
struct mastruct variable;
```

- ❑ **struct**: variable de type structure.
- ❑ **mastruct**: le nom de la structure.
- ❑ **variable** : le nom de la variable de type struct mastruct.

Exemple :

- ❑ Déclaration d'une variable de type struct compte:



Ne pas confondre : nom de la structure (**compte**), avec la variable (**c**) et les champs (**nbanque**, **ncompte**, **solde**).

Allocation dynamique de la mémoire et Structure

Opérations sur les structures :

- ❑ Une variable de type structure peut être manipuler soit :
 - champ par champ: traiter un seul champ.
 - **exp**: lecture/mise à jour du champ '**solde**'.
- ❑ comme un tout: l'enregistrement.
 - **exp**: initialisation/suppression d'un enregistrement.

Accès aux champs d'une structure :

On utilise l'opérateur **point .** pour référencer un champ d'une variable de type structure.

Syntaxe
`variable.champ`

Exemple:

c.solde

Cette instruction permet d'accéder au champ **solde** de la variable **c**.

Allocation dynamique de la mémoire et Structure

Exemples :

Exemple 1:

Cette instruction affecte la valeur d'un champ à une variable:

```
float var = c.solde ;
```

Exemple 2:

Après l'instruction suivante:

```
struct compte c,b;
```

On peut écrire:

```
float différence = c.solde - b.solde ;
```

Exemple 3:

Les champs sont modifiables.

```
c.solde -= 200.25 ;
```

Allocation dynamique de la mémoire et Structure

Exercice :

- ☐ Déclarer la structure compte.
- ☐ Créer deux variables b et c de type compte.
- ☐ Remplir les champs des 2 variables puis afficher leurs contenus. (printf).
- ☐ Ajouter au solde de b la somme 100,50.
- ☐ Afficher le nouveau solde de b.

	nbanque	ncompte	solde
l'enregistrement b	1	10100	2000,00
l'enregistrement c	1	10101	3000,00

Allocation dynamique de la mémoire et Structure

Solution :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  struct compte{ // déclaration de la structure compte
4  int nbanque;
5  int ncompte;
6  float solde; };
7  main(){
8  struct compte b, c;
9  b.nbanque=1; b.ncompte=10100; b.solde=2000.00;
10 c.nbanque=1; c.ncompte=10101; c.solde=3000.00;
11 printf("%d, %d, %f", b.nbanque, b.ncompte, b.solde);
12 printf("\n%d, %d, %f", c.nbanque, c.ncompte, c.solde);
13 b.solde+=100.50;
14 printf("\n%f", b.solde);
15 }
16
```

```
1, 10100, 2000.000000
1, 10101, 3000.000000
2100.500000
Process returned 0 (0x0)   execution time : 0.056 s
Press any key to continue.
```


Allocation dynamique de la mémoire et Structure

Initialisation de variables de type structure :

Initialisation au moment de la déclaration :

- ❑ Comme dans les tableaux, on peut initialiser les champs d'une variable au moment de la déclaration :

Syntaxe

```
struct mastruct variable = { expr1, ..., exprN } ;
```

- ❑ l'ordre des expressions (*exp1*, ..., *expN*) doit être le même que celui des champs lors de la déclaration de la structure,

Exemple

```
struct compte c = { 1,10101,3000.00};  
struct compte b = { 1,10101};
```

- ❑ Les champs manquants (à la fin) sont initialisés à 0 ou NULL.

Allocation dynamique de la mémoire et Structure

Initialisation par recopie :

- ❑ l'opérateur = peut être utilisé avec des variables de types structures. Il copie les variables **champ à champ**.

```
Exemple
struct compte b , c = { 1,10101 , 3000.00 };
b = c ;
est équivalent à

    b.nbanque = c.nbanque ;
    b.ncompte = c.ncompte ;
    b.solde = c.solde ;
```

NB: **Ce n'était pas possible avec les tableaux !**

Contre exemple:

```
int T1[3]={1, 2, 3};
```

```
int T2[3];
```

```
T2=T1; // instruction fausse
```

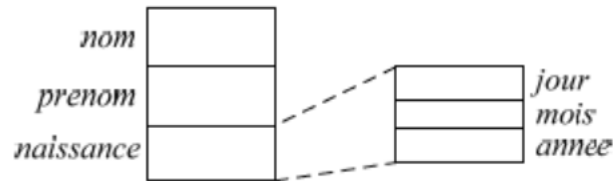
Allocation dynamique de la mémoire et Structure

Les structures imbriquées :

- ❑ Les champs d'une structure peuvent être de tout type, y compris de type structuré.
- ❑ Lorsqu'un champ est de type structure on parle de structures imbriquées.

Exemple :

- ❑ Déclaration de la structure :



```
1 struct date {  
2     short int jour ;  
3     short int mois ;  
4     short int annee ; };  
5  
6 struct personne {  
7     char nom[20] ;  
8     char prenom[20] ;  
9     struct date naissance ; /* ce champ est de type structuré */  
10 };
```

- ❑ Accès aux champs et utilisation :

```
struct personne professeur;  
professeur.nom=KHOURLIFI;  
professeur.prenom=Youness;  
professeur.naissance.annee=1984;  
professeur.naissance.jour=18;  
professeur.naissance.mois=09;
```

Tri et recherche & Gestion des fichiers

Définition d'un algorithme de Tri :

- ❑ Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité,
- ❑ Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger en ordre croissant ou décroissant,
- ❑ Trier un tableau c'est donc ranger les éléments d'un tableau en ordre
- ❑ croissant ou décroissant,
- ❑ Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur:
 - Tri par sélection;
 - Tri par insertion;
 - Tri à bulles;
 - Tri rapide ou quick sort.

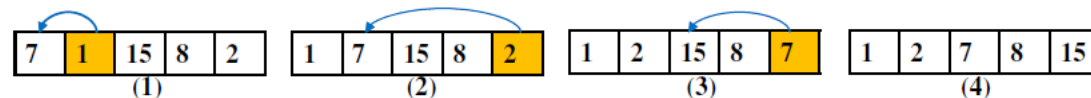
Tri et recherche & Gestion des fichiers

Tri par sélection :

Principe de l'algorithme

- ❑ Le principe de cette algorithme très intuitive consiste à :
- ❑ Chercher le **minimum** dans un sous-tableau (au départ le tableau complet contenant les N valeurs non ordonnées),
- ❑ Permuter ce minimum avec le **premier élément** du sous-tableau,
- ❑ Puis **itérer** ce traitement sur un nouveau sous-tableau de N-1 éléments (on ne tient plus compte du premier élément qui est maintenant à sa place). Continuer jusqu'à ce que le tableau soit entièrement trié.
- ❑ Généralement, pour trier le sous-tableau $T[i..nbElements]$ il suffit
- ❑ de positionner au rang i le **plus petit élément** de ce sous-tableau et de trier le sous-tableau $T[i+1..nbElements]$.

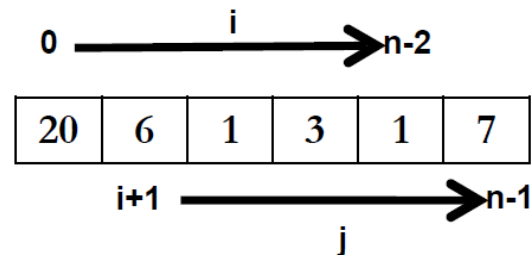
Exemple : tri par sélection du minimum du tableau {7, 1, 15, 8, 2}.



Tri et recherche & Gestion des fichiers

Procédure du tri par sélection :

```
1 void tri_selection(int t[], int n) {  
2     int i, j, min, tmp;  
3     for(i = 0 ; i <= n - 2 ; i++) {  
4         min = i;  
5         for(j = i+1 ; j <= n - 1 ; j++)  
6             if(t[j] < t[min])  
7                 min = j;  
8         if(min != i) { // Si min ≠ i, on fait l'échange.  
9             tmp = t[i];  
10            t[i] = t[min];  
11            t[min] = tmp;  
12        }  
13    }  
14 }
```



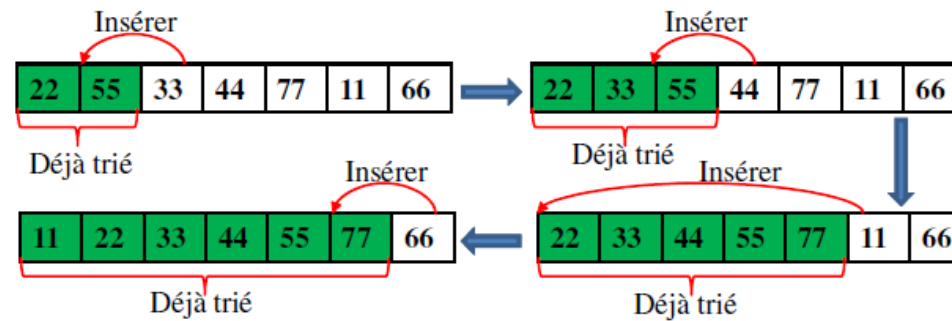
Tri et recherche & Gestion des fichiers

Tri par insertion :

Principe de l'algorithme

- ❑ Consiste à parcourir la liste : on prend les éléments dans l'ordre.
- ❑ Ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère.
- ❑ Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.

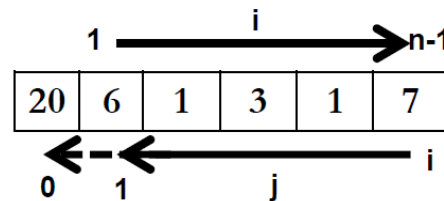
Exemple : tri par insertion du tableau {22, 55, 33, 44, 77, 11, 66}.



Tri et recherche & Gestion des fichiers

Procédure du tri par insertion :

```
1 void tri_insertion(int t[], int n) {
2     int i, j, tmp;
3     for (i = 1; i <= n - 1; i++) { // Sélection de l'élément à insérer
4         tmp = t[i];
5         j = i;
6         while (j > 0 && t[j-1] > tmp) { // Décalage des éléments plus grands
7             t[j] = t[j-1];
8             j = j - 1;
9         }
10        t[j] = tmp; // Insertion
11    }
12 }
13
```



Tri et recherche & Gestion des fichiers

Tri à bulles :

Principe de l'algorithme

- ❑ Le principe de cette méthode consiste à **comparer les couples de valeurs successives $Tab[i]$ et $Tab[i+1]$** pour i variant de 0 à $N-2$, et à **les permuter si elles sont mal ordonnées**.
- ❑ L'algorithme s'arrête lorsque l'on constate **qu'aucune permutation n'a été effectuée**.

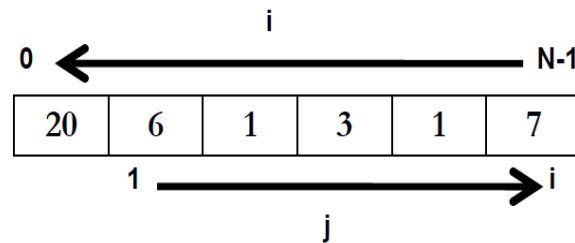
Exemple :

→	18	10	3	25	9	2
	10	18	3	25	9	2
	10	3	18	25	9	2
	10	3	18	25	9	2
	10	3	18	9	25	2
→	10	3	18	9	2	25
	3	10	18	9	2	25
	3	10	18	9	2	25
	3	10	9	18	2	25
	3	10	9	2	18	25
→	3	10	9	2	18	25
	3	9	10	2	18	25
→	3	9	2	10	18	25
	3	9	2	10	18	25
→	3	2	9	10	18	25
	2	3	9	10	18	25

Tri et recherche & Gestion des fichiers

Procédure du Tri à bulles :

```
1 void tri_a_bulles(int t[], int n){
2     int i, j, tmp;
3     for(i=n-1 ; i>=0 ; i--){ // Sélection des éléments à permuter
4         for (j=1 ; j<=i ; j++){ // décalage des éléments les plus grands
5             if (t[j-1]>t[j]){ // permutation
6                 tmp = t[j-1];
7                 t[j-1] = t[j];
8                 t[j] = tmp;
9             }
10        }
11    }
12 }
13
```



Tri et recherche & Gestion des fichiers

Tri rapide (Quick Sort) :

Principe de l'algorithme : diviser pour résoudre

- ☐ Choisir un seuil (pivot) ;
- ☐ Mettre les éléments les plus petits que le seuil à gauche et les éléments les plus grands à droite du seuil;
- ☐ Recommencer séparément sur les parties droite et gauche.

g		d		
3	7	8	5	2
i	p		j	

Tri et recherche & Gestion des fichiers

Procédure du Tri rapide :

```
1 void tri_rapide(int t[], int g, int d) {  
2     int i, j, p, tmp ;  
3     i=g; j=d; p=t[(i+j)/2];  
4     do {  
5         while (t[i]<p) i++;  
6         while (t[j]>p) j--;  
7         if(i<=j){  
8             tmp=t[i];  
9             t[i]=t[j];  
10            t[j]=tmp;  
11            i++; j--;  
12        }  
13    } while(i<=j);  
14    if( g<j) tri_rapide(t,g,j);  
15    if (i<d) tri_rapide(t,i,d);  
16 }  
17
```

g		d		
3	7	8	5	2
i	p		j	

Tri et recherche & Gestion des fichiers

Les techniques de recherche de données dans un tableau :

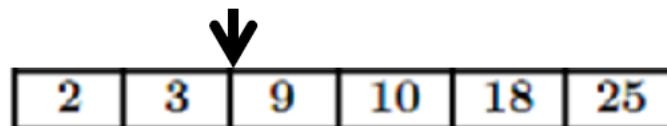
Les techniques de recherche :

- ☐ Recherche séquentielle
- ☐ Recherche dichotomique dans un tableau trié

Recherche séquentielle :

- ☐ Elle consiste à parcourir les éléments du tableau progressivement du début jusqu'à la fin et les comparer avec l'élément recherché **x**.
- ☐ Si le tableau n'est pas trié, arriver à la fin du tableau signifie que l'élément n'existe pas,
- ☐ Dans un tableau trié de manière croissante, le premier élément trouvé supérieur à l'élément recherché **x**, permet d'arrêter la recherche. Aussi, cette position correspond à celle où il faudrait insérer l'élément recherché pour garder un tableau trié.

Recherché 8



2	3	9	10	18	25
---	---	---	----	----	----

Tri et recherche & Gestion des fichiers

Recherche séquentielle: complexité

- ❑ Une recherche dans un tableau trié (de taille N) nécessite en moyenne $N/2$ comparaisons.
- ❑ Dans un tableau non trié, on se rapproche de N comparaisons lorsqu'on recherche un élément qui n'existe pas dans le tableau.

Exercice :

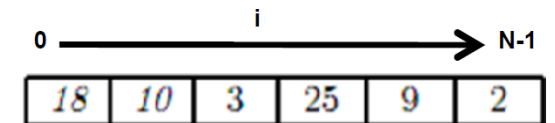
- ❑ Ecrire la fonction de recherche séquentielle en C. Elle retourne -1 si la valeur recherchée ne se trouve pas dans le tableau et l'indice de la première occurrence trouvée sinon.

Utiliser le prototype suivant:

`int rech_sequentielle (int T[], int N, int x)`

Fonction de recherche séquentielle :

```
1 //Fonction de recherche séquentielle
2 int rech_sequentielle (int T[], int N, int x) {
3     /* retourne l'indice de la première occurrence trouvée et -1 si x ne se trouve pas dans le tableau */
4     int i;
5     for(i=0; i<N; i++)
6         if(T[i]==x) return i ;
7     return -1;
8 }
```



Tri et recherche & Gestion des fichiers

Recherche dichotomique :

- ❑ Principe: On compare la valeur cherchée x avec l'élément central du tableau, si ce n'est pas la bonne, un test permet de trouver dans quelle moitié du tableau se trouve la valeur x .
- ❑ On continue récursivement jusqu'à un sous-tableau de taille 1.
- ❑ Complexité: Dans le cas d'un tableau trié, le nombre de comparaison est de l'ordre de $\log_2(N)$. Ainsi, l'espace de recherche est limité.

Exemple :

Le programme suivant affiche la position de la valeur recherchée x si elle existe et -1 sinon.

Tri et recherche & Gestion des fichiers

Recherche dichotomique :

```
1  #include<stdio.h>
2  main() {
3      int t[50], N, i, x, pos, premier, dernier, milieu;
4      printf("Entrer la dimension du tableau (Max: 50) :");
5      scanf("%d", &N);
6      for(i=0; i<N; i++) {
7          printf("t[%d]", i);
8          scanf("%d", &t[i]);
9      }
10     printf("Saisir une valeur à rechercher :");
11     scanf("%d", &x);
12     pos = -1;
13     premier = 0; dernier = N-1;
14     while((dernier >= premier) && (pos == -1)) {
15         milieu = (dernier + premier) / 2;
16         if(x < t[milieu])
17             dernier = milieu - 1;
18         else if(x > t[milieu])
19             premier = milieu + 1;
20         else
21             pos = milieu;
22     }
23     if(pos == -1)
24         printf("la valeur ne se trouve pas dans le tableau.");
25     else
26         printf("%d se trouve dans %d", x, pos);
27     return 0;
28 }
```

2	3	9	10	18	25
---	---	---	----	----	----

milieu =
premier (dernier + premier) / 2 dernier
0 N-1

2	3	9	10	18	25
---	---	---	----	----	----

premier dernier =
0 milieu-1 milieu

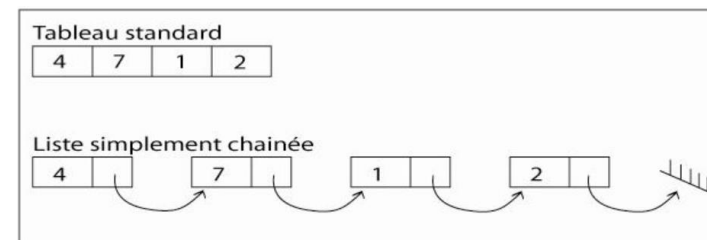
2	3	9	10	18	25
---	---	---	----	----	----

premier = milieu+1 dernier
milieu N-1

2	3	9	10	18	25
---	---	---	----	----	----

Les listes chaînées

- ❑ Les listes chaînées sont des **structures de données** semblables aux tableaux sauf que l'accès à un élément ne se fait pas par index mais à l'aide d'un pointeur.
- ❑ L'allocation de la mémoire est faite au moment de l'exécution.
- ❑ Une **liste** est un ensemble d'objets constituant les éléments (**maillons**) de la liste.
- ❑ Les éléments sont **chaînés entre eux** et on peut facilement ajouter ou extraire un ou plusieurs éléments.
- ❑ L'accès aux éléments d'une liste se fait de **manière séquentielle**.
- ❑ Chaque élément **permet l'accès au suivant** (contrairement au cas du tableau dans lequel l'accès se fait de manière absolue, par adressage direct).
- ❑ Il existe différents type de listes chaînées :
 1. Les listes simplement chaînées
 2. Les listes doublement chaînées
 3. Les listes circulaires
 4. Les piles
 5. Les files



Les listes chaînées

Représentation

- ❑ Une liste est une structure de données telle que chaque maillon contient :
 - des **données** et un **pointeur sur un autre maillon (pointeur suivant)** de la liste, ou un **pointeur NULL** s'il n'y a pas de maillon suivant.
 - la **tête (pointeur particulier)** donne **accès au premier maillon**, le reste de la liste est accessible en passant de **maillon en maillon**, suivant leur enchaînement.
- ❑ La construction du prototype d'un élément de la liste
 - Pour définir un élément de la liste le type struct sera utilisé.
 - L'élément de la liste contiendra un champ donnée et un pointeur suivant.
 - Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément.
 - Le pointeur "suivant" permettra l'accès vers le prochain élément.

```
typedef struct ListeRepere {  
    Element *debut;  
    Element *fin;  
    int taille;  
}Liste;
```

- Le pointeur **début** contiendra l'adresse du premier élément de la liste.
- Le pointeur **fin** contiendra l'adresse du dernier élément de la liste.
- La variable **taille** contient le nombre d'éléments.