Cours Programmation Langage C

Pr RAOUYANE Brahim **Pr CHBIHI LOUHDI Med Reda**

Plan Partie II

- **Séance 5 : Types composés (1) : Tableaux**
- **Séance 6 : Fonctions**
  - **Fonctions** : *Fonction*, *Procédure*
- **Séance 7 : Pointeurs**
- **Séance 8 : Types composés (2)**
  - **Variables**
    - **Structures**
    - **Champs de bits**
    - **Unions**
  - **Constantes**
    - **Énumérations**
  - **Définition de types: typedef**

Types composes : Tableaux

- Un tableau est un ensemble fini d'éléments de même type, stockes en ***mémoire a des adresses contigües***.
- La déclaration d'un tableau a une dimension se fait de la façon suivante :

  Type Nom-du-tableau[**nombre-éléments**];
- **nombre- éléments** est une expression constante entière positive.
- **Exemple:** `int tab[10];`
  - La déclaration indique que **tab** est un tableau de **10** éléments de type **int**.
  - Cette déclaration alloue donc en mémoire pour l'objet tab un espace de **10 × 4 octets consécutifs**.

Types composes : Tableaux

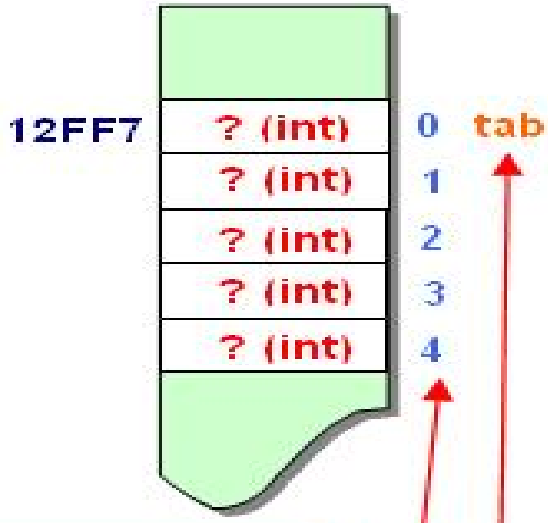
Déclaration

```
int tab[5] ;  // déclare un tableau de 5 entiers
```

- Nombre de cellules du tableau
- Les crochets [ ] indiquent qu'il s'agit d'un tableau
- Nom donné au tableau
- Type de chaque cellule du tableau

La dimension du tableau doit être une valeur constante, supérieure ou égale à 1, pour pouvoir être évaluer au moment de la compilation. Par conséquent, on ne pourra jamais utiliser une variable non *const* pour spécifier la dimension d'un tableau.

Allocation Mémoire



Attention, la première cellule du tableau est référencée par l'indice **0** et la dernière par l'indice **n-1** pour un tableau de **n** cases.

Nom du tableau.
Ce nom correspond à l'adresse de la première case du tableau.

Types composes : Tableaux

- Il est recommande de donner un nom à la constante **nombre-éléments** par une directive au *préprocesseur*.
- **Exemple** :

```
#define nombre-elements 10
```

- On accède a un élément du tableau en lui appliquant l'operateur **[]**.
- Les éléments d'un tableau sont toujours numérotes de **0** à **nombre-éléments -1.**

Types composes : Tableaux

- **Exemple:**

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10
main()
{
int tab[N];
int i;
for (i = 0; i < N; i++)
printf("tab[%d] = %d \n",i,&tab[i]);
}
```

```
tab[0] = 2686612
tab[1] = 2686616
tab[2] = 2686620
tab[3] = 2686624
tab[4] = 2686628
tab[5] = 2686632
tab[6] = 2686636
tab[7] = 2686640
tab[8] = 2686644
tab[9] = 2686648
```

Types composes : Tableaux

- On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

type nom-du-tableau[**N**] = { ***constante-1, constante-2,…,constante-N*** };

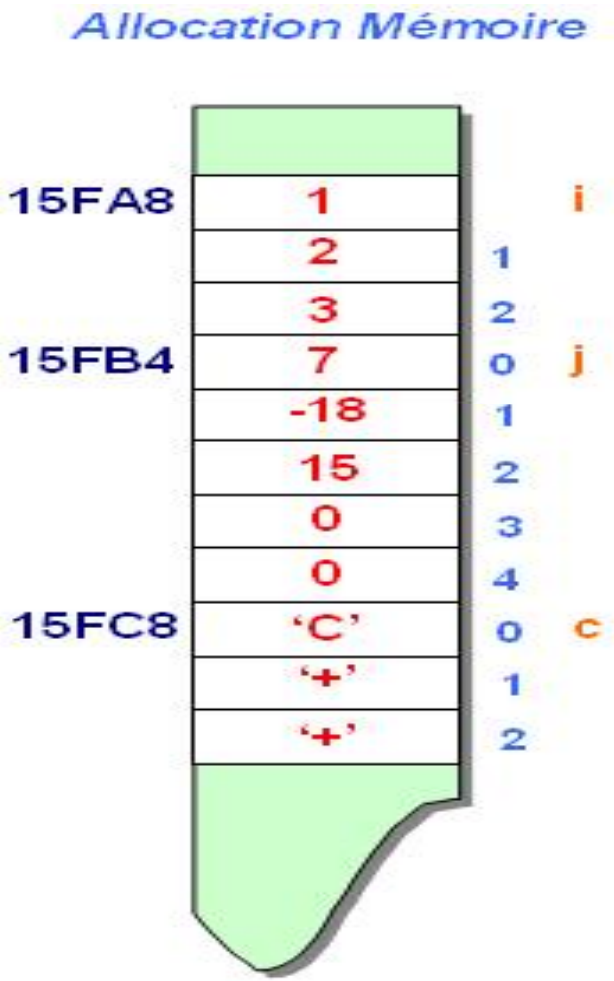
- **Exemple:**

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define N 4
4   int tab[N] = {1, 2, 3, 4};
5   main()
6   {
7   int i;
8   for (i = 0; i < N; i++)
9   printf("tab[%d] = %d\n",i,tab[i]);
10  }
```

```
tab[0] = 1
tab[1] = 2
tab[2] = 3
tab[3] = 4
```

Types composes : Tableaux

```
int main()
{
  int i[3] = {1, 2, 3};        // tableau de 3 entiers
  int j[5] = {7, -18, 15};     // tableau de 5 entiers
  char c[] = {'C', '+', '+'}; // tableau de 3 caractères
  return 0;
}
```



Types composes : Tableaux

- **Remarque :**
  - Si le nombre de données dans la liste d'initialisation est inferieur à la dimension du tableau, seuls les premiers éléments seront initialises. Les autres éléments seront mis a zéro .
  - De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaine de caractères littérale.
    - Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation.
    - **Exemple**: le programme suivant imprime le nombre de caractères du tableau tab, ici 8. char tab[] = "exemple";

Types composes : Tableaux

- Une chaîne de caractères est une collection de caractères. Elle correspond donc à la définition du tableau.
- C'est en fait, un cas particulier d'un tableau de caractères, qui comporte un caractère supplémentaire de fin de chaîne qui s'appelle le caractère nul terminal ' \0 ' . C'est un caractère de contrôle.
- Un tableau de caractères peut-être initialisé soit avec une liste de caractères littéraux séparés par des virgules, soit avec une constante littérale chaîne.
- Remarquons toutefois que les deux formes ne sont pas équivalentes. La différence se situe au niveau du caractère nul terminal.

Types composes : Tableaux

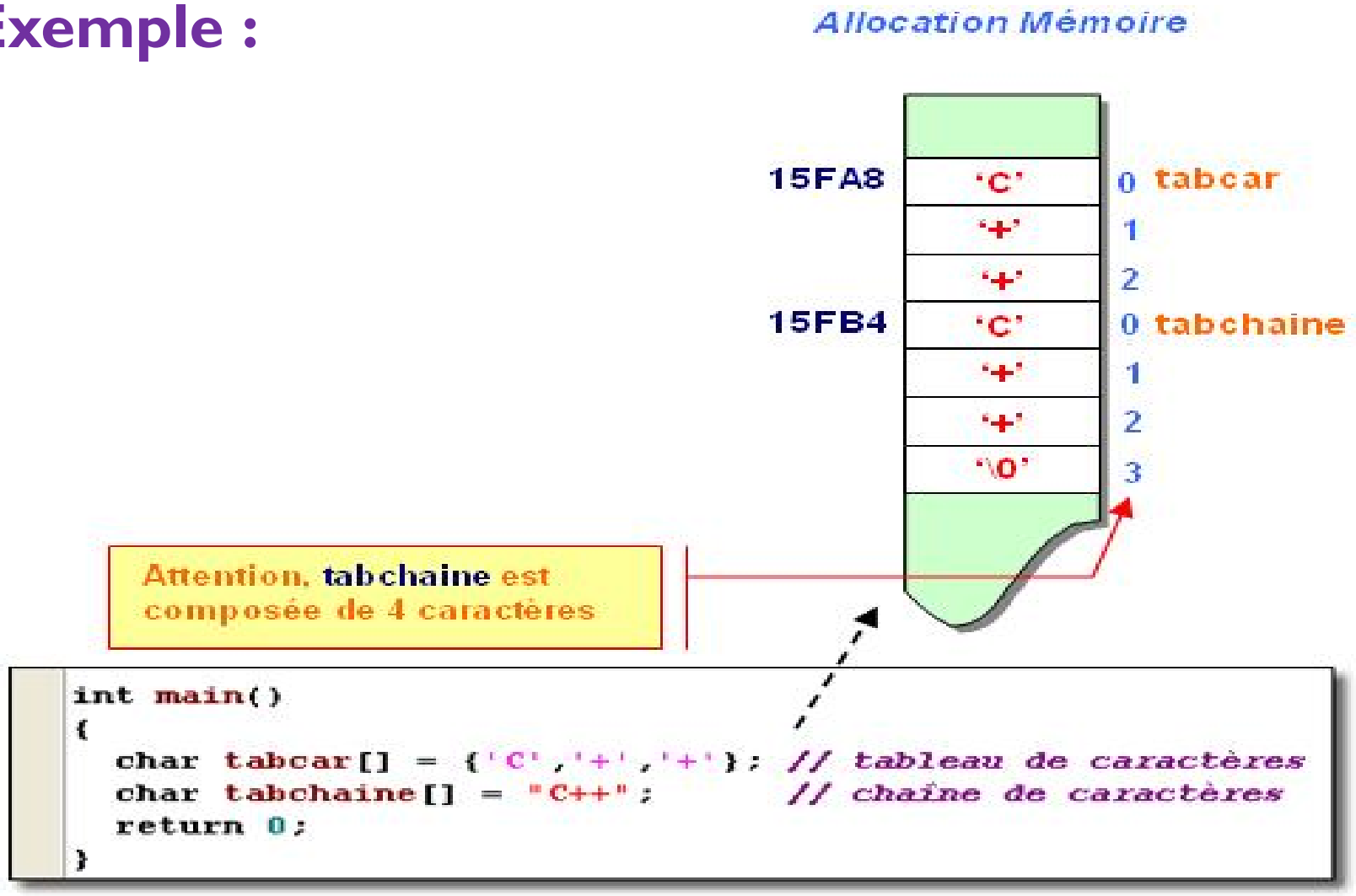
- **Exemple :**

```c
#include <stdio.h>
#include <stdlib.h>
#define N 8
char tab[N] = "exemple";
main()
{
int i;
for (i = 0; i < N; i++)
printf("tab[%d] = %c\n",i,tab[i]);
}

```

```
tab[0] = e
tab[1] = x
tab[2] = e
tab[3] = m
tab[4] = p
tab[5] = l
tab[6] = e
tab[7] =
```

Types composes : Tableaux

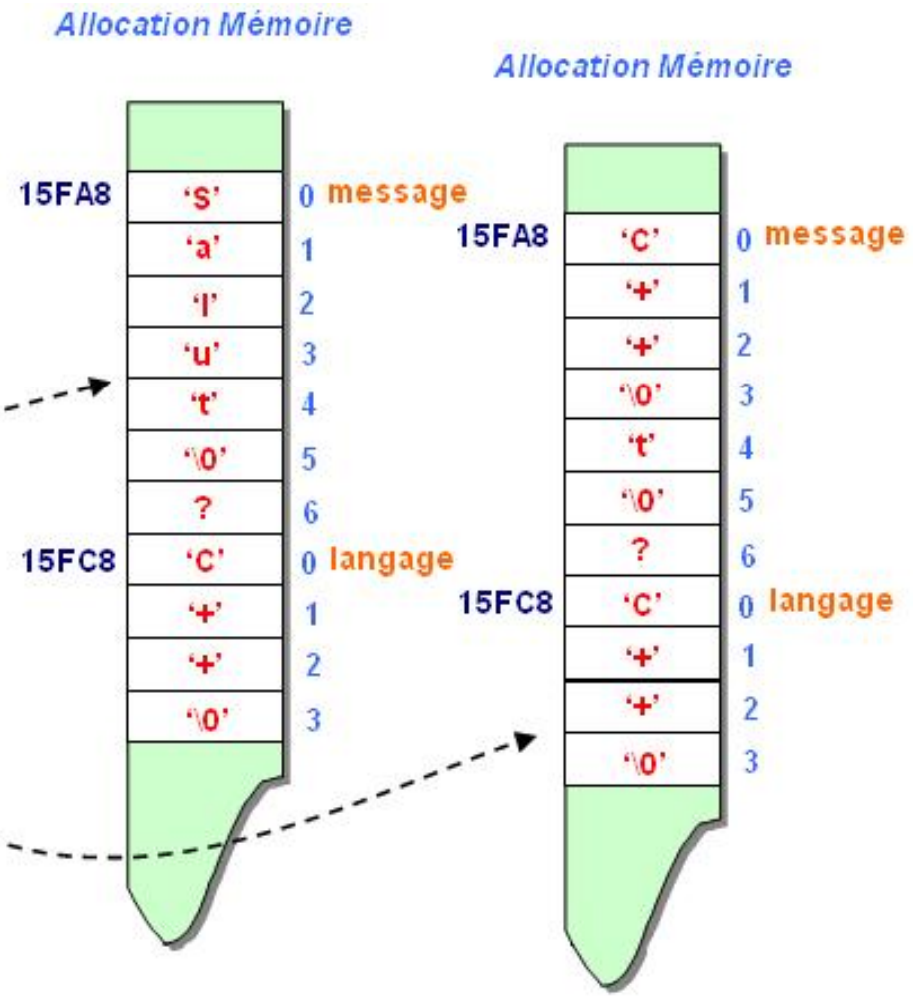
- **Exemple :**



Types composes : Tableaux

- **Exemple :**

```
int main()
{
  char message[7] = "Salut";
  char langage[] = "C++";

  message = "C++";     // écriture interdite
  message = langage;   // écriture interdite

  for (int i=0; message[i]=langage[i]; i++);
  return 0;
}
```



Types composes : Tableaux

- Pour déclarer un tableau à plusieurs dimensions:

L'opérateur [] désigne un dimension

- **Exemple:** pour un tableau a deux dimensions :

type nom-table[Nbre-lignes][Nbre-colonnes]

  - En fait, un tableau a deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau.
  - Pour accéder a un élément du tableau par l'expression **"Tab[i][j]".**

- Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes .

Types composes : Tableaux

- **Exemple:**

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define M 2
4   #define N 3
5   int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};
6   main()
7   {
8
9   int i, j;
10  for (i = 0 ; i < M; i++
11  {
12  for (j = 0; j < N; j++)
13  printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
14  }
15
16  }
```

```
tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
tab[1][0]=4
tab[1][1]=5
tab[1][2]=6
```

Types composes : Tableaux

- **Exemple:**

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
int i,j,N,M;
printf("Entrez le nbre de ligne:");
scanf("%d",&N);
printf("Entrez le nbre de colonnes:");
scanf("%d",&M);
int tab[N][M];
for (i = 0 ; i < N; i++)
{
    for (j = 0; j < M; j++)
        {
            printf("Entrez la valeur de la case tab[%d][%d]:",i,j);
            scanf("%d",&tab[i][j]);}
        }
printf("Le contenu de la table tab[%d][%d]:\n",N,M);
for (i = 0 ; i < N; i++)
        {
    for (j = 0; j < M; j++)
                printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
        }
printf("Le contenu de la table tab[%d][%d] sous forme Matrice : \n",N,M);
for (i = 0 ; i < N; i++)
        {
    for (j = 0; j < M; j++)
                printf(" \t %d \t",tab[i][j]);
                printf("\n",tab[i][j]);
        }
}
```

Types composes : Tableaux

- **Exemple:**

```
Entrez le nbre de ligne:3
Entrez le nbre de colonnes:3
Entrez la valeur de la case tab[0][0]:1
Entrez la valeur de la case tab[0][1]:0
Entrez la valeur de la case tab[0][2]:0
Entrez la valeur de la case tab[1][0]:0
Entrez la valeur de la case tab[1][1]:1
Entrez la valeur de la case tab[1][2]:0
Entrez la valeur de la case tab[2][0]:0
Entrez la valeur de la case tab[2][1]:0
Entrez la valeur de la case tab[2][2]:1
Le contenu de la table tab[3][3]:
tab[0][0]=1
tab[0][1]=0
tab[0][2]=0
tab[1][0]=0
tab[1][1]=1
tab[1][2]=0
tab[2][0]=0
tab[2][1]=0
tab[2][2]=1
Le contenu de la table tab[3][3] sous forme Matrice :
          1                    0                    0
          0                    1                    0
          0                    0                    1

--------------------------------
```

Fonction et Procedure

- Pourquoi?
- C'est quoi la différence entre Fonction et Procédure
- Comment écrire un sous Programme?
- Récursivité

Sous-algorithme: Pourquoi ?

- **Par exemple, pour résoudre le problème suivant :**
  - Écrire un programme qui affiche en ordre croissant les notes d'une classe suivies de la note la plus faible, de la note la plus élevée et de la moyenne.
  - Revient à résoudre les sous problèmes suivants :
    1) Remplir un tableau de naturels avec des notes saisies par l'utilisateur
    2) Afficher un tableau de naturels
    3) Trier un tableau de naturel en ordre croissant
    4) Trouver le plus petit naturel d'un tableau
    5) Trouver le plus grand naturel d'un tableau
    6) Calculer la moyenne d'un tableau de naturels

Sous-Programme: Utilisation

- Les **fonctions et les procédures sont des modules** ont plusieurs avantages :
  - **Permettent d'éviter de réécrire un même traitement plusieurs fois.**
  - On fait appelle à **un sous algorithme aux endroits spécifiés.**
  - Permettent **d'organiser le code et améliorent la lisibilité des programmes**
  - **Facilitent la maintenance du code (il suffit de modifier une seule fois)**
  - Ces procédures et fonctions peuvent éventuellement **être réutilisées** dans d'autres programmes

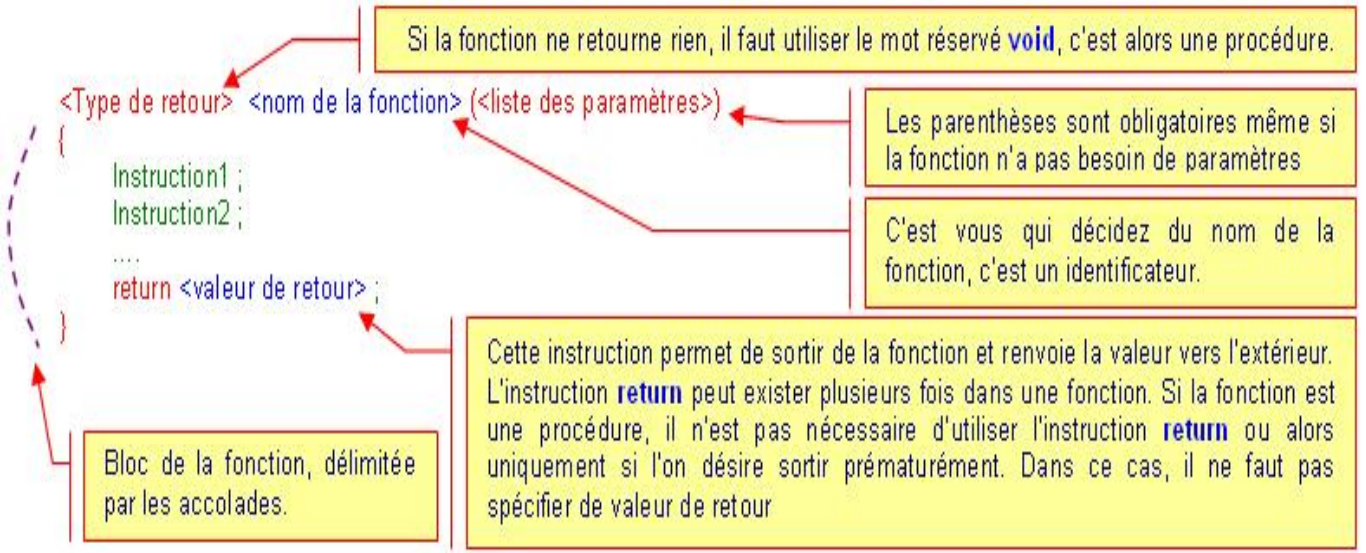
Programmation procédurale

Programmation modulaire

Définition d'une fonction

- Une **fonction** peut s'apparenter à une **opération** par l'utilisateur.
- Une **fonction** est représentée par un **nom**.
- Les **opérandes** d'une fonction, appelée **paramètres**, sont spécifiés dans une liste entourée de **parenthèses**, les paramètres étant séparés par des **virgules**.
- Le **résultat d'une fonction** se nomme valeur de **retour et le type** de la valeur de retour s'appelle type de retour.
- Une **fonction** ne renvoyant pas de valeur a un type de retour **void**, ce qui veut dire qu'elle ne renvoie rien.
- Dans ce cas particulier, cette fonction s'appelle une **procédure**. **Les actions qu'exécute une fonction sont spécifiées dans le corps de la fonction.**
- Le **type de retour** de la fonction suivi du **nom de la fonction**, la liste des **paramètres** et le **corps** de la fonction, composent la définition de la fonction.

Définition d'une fonction



- **Exemple** :

```
//-------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++)
    resultat *= y;
  return resultat;
}
//-------------------------------------
int main()
{
  int binaire = puissance(2, 10);
  return 0;
}
//-------------------------------------
```

Chaque paramètre doit être séparé des autres par une virgule. Chaque paramètre doit également posséder un nom précédé de son type.

Il est possible de déclarer des variables au sein de la fonction. Elles sont appelées variables locales.

Il peut y avoir plusieurs **return**. Comme la fonction possède une valeur de retour, la valeur renvoyée doit avoir un type compatible avec le type de retour.

Utilisation de la fonction. Pour pouvoir être utilisée, une fonction doit être connue et donc être définie au préalable avant la fonction principale main.
binaire ← 1024 (2 x 2 x 2 x.....x 2) 10 fois.

Déclaration des fonctions

<Type de retour> <nom de la fonction> (<liste des paramètres>) // Signature d'une fonction

-1-

```
int produit (int, int );


main() {
 int a = 2, b = 5;
printf("%d\n", produit(a,b));
 }


int produit (int a, int b) {
return(a*b); }
```

-2-

```
int produit (int a, int b) {
return(a*b); }


main() {
 int a = 2, b = 5;
printf("%d\n", produit(a,b));
 }
```

-3-

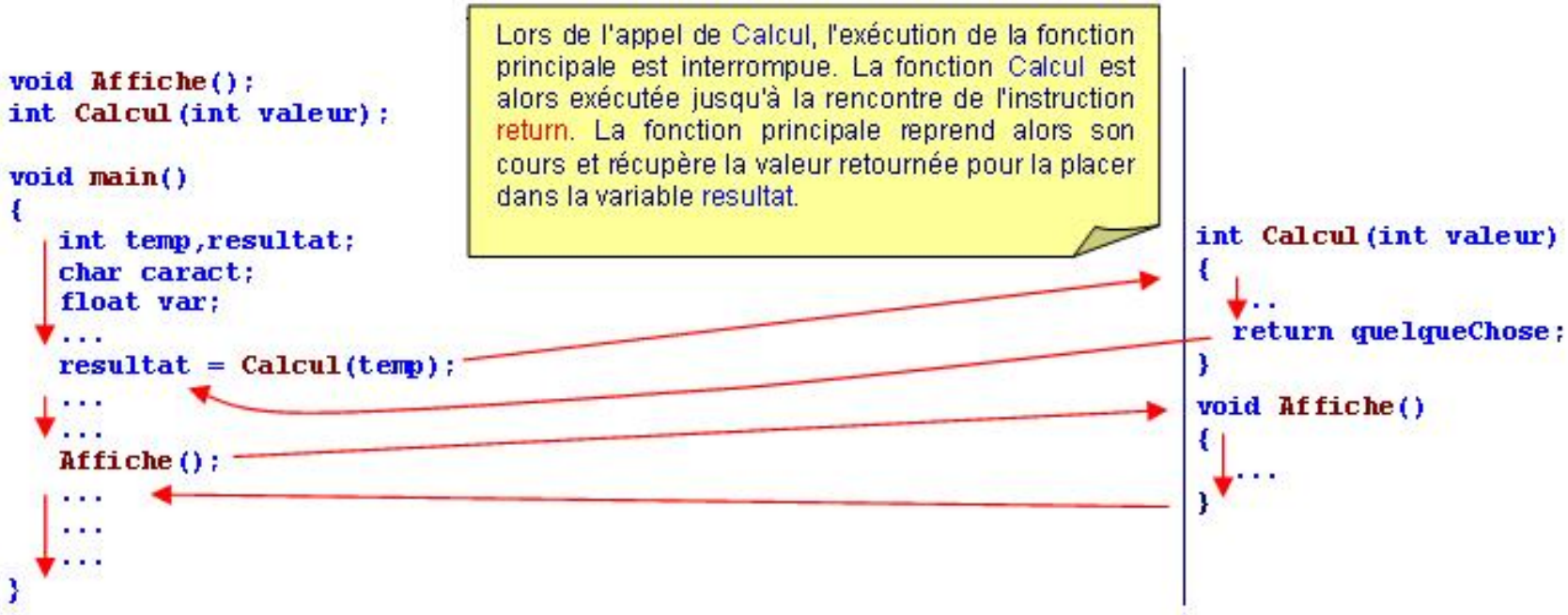
```
//File PRO.h

int produit (int, int );
```

```
//File PRO.c

#include " PRO.h "

int produit (int a, int b) {
return(a*b); }
```

```
// Programme Principale
#include " PRO.h "
#include <stdio.h>

main() {
 int a = 2, b = 5;
printf("%d\n", produit(a,b));
 }
```

Appel d'une fonction

- Lors de **l'appel d'une fonction** il y a **suspension** de l'exécution de la fonction en cours.
- On « **saute** » à l'exécution de la fonction appelée. Quand l'évaluation de la fonction appelée est terminée, la fonction suspendue reprend son exécution à l'endroit qui suit immédiatement l'appel.
- L'exécution d'une fonction se termine une fois exécutée la dernière instruction du corps de la fonction ou quand une instruction return est rencontrée dans le corps de la fonction.



Exercice Fonction

- **On dispose d'un tableau T d'entier de taille N**
  - Ecrire un **sous-programme** qui permet le remplissage d'un tableau.
  - Ecrire un **sous-programme** qui permet l'affichage d'un tableau.
  - Ecrire un **sous-programme** qui retourne la valeur de nombres pairs dans la table
  - Ecrire un **sous-programme** qui permet de supprimer un valeur **X**

Plan

- **Chapitre 5 :Pointeurs**
  - Adresse et Valeur d'un objet
  - Notion
  - Arithmétique des pointeurs
  - Allocation dynamique
  - Utilisation dans:
    - Tableaux
    - Fonction passage par Valeur/Adresse
    - Chaines de caractères
    - Structures

Pointeurs

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifies de manière univoque par un numéro qu'on appelle adresse.
- Pour retrouver une variable, il suffit donc de connaitre l'adresse de l'octet ou elle est stockée.
- Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse.
- l'identification d'une variable est fait par le compilateur, qui est le lien entre variable et son adresse en mémoire.
- Dans des cas, c'est très pratique de manipuler directement une variable par son adresse.

Pointeurs: Adresse et Valeur d'un objet

- **Lvalue** (**left value**) tout objet pouvant être place à gauche d'un opérateur d'affectation.
- Une **Lvalue** est caractérisée par :
  - **Son adresse:** l'adresse-mémoire à partir de laquelle l'objet est stocke ;
  - **Sa valeur:** le contenu est stocké à cette adresse.
- **Exemple**: **int i, j; i = 3; j = i;**
  - Si le compilateur a placé la variable **i** à l'adresse **4831836000** en mémoire, et la variable **j** à l'adresse **4831836004**, on a

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

Pointeurs: Adresse et Valeur d'un objet

- **Exemple**: **int i, j; i = 3; j = i;**

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

  - Deux **variables différentes** ont des **adresses différentes**.
  - L'affectation **i = j** ; **n'opère que sur les valeurs des variables**.
  - Les variables **i** et **j** étant de type **int**, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse **4831836000** à **4831836003**.
  - L'adresse d'un objet étant un **numéro d'octet en mémoire**, il s'agit d'un **entier** quelque soit le type de l'objet considère.

Pointeurs: Adresse et Valeur d'un objet

- **Exemple**: **int i, j; i = 3; j = i;**

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

  - Le format interne de cet entier (**16 bits, 32 bits ou 64 bits**)
  - L'operateur **&** permet d'accéder à l'adresse d'une variable.
  - **&i** n'est pas une **Lvalue** mais une **constante** : on ne peut pas faire figurer **&i** à gauche d'un operateur d'affectation.
- **Pour manipuler les adresses**, on doit recourir un nouveau type d'objets: **Pointeurs**.

Pointeurs: Notion

- Un **pointeur** est un objet (**Lvalue**) dont la valeur est égale à l'adresse d'un autre objet.
- Déclaration d'un **pointeur** :

type *nom-du-pointeur ;

  - **type** est le type de l'objet pointe.
  - **nom-du-pointeur,** associé à un objet dont la valeur est l'adresse d'un autre objet de type.
  - L'identificateur **nom-du-pointeur** est un identificateur d'adresse. Comme un **Lvalue**, sa valeur est **modifiable**.
  - Même si la valeur d'un pointeur est toujours un entier (un entier long), le **type** d'un pointeur dépend du type de l'objet vers lequel il pointe.
  - La différence est visible dans l'espace réservé par le **type**

Pointeurs: Notion

- **Exemple:**

```
int i = 3, *p;
p = &i;
```

- Dans la mémoire :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

- L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé.
- **p** est un pointeur vers un entier **i.**
- ***p** désigne la valeur de **i**.

Pointeurs: Notion

- **Exemple:** :

```c
#include <stdio.h>

int main()
{
    int i=10,*p;
    p=&i;
    printf("Valeur de   &i:   %x \n",&i);
    printf("Valeur de   i :   %d \n",i);


    printf("Valeur de   &p:   %x \n",&p);
    printf("Valeur de   p :   %x \n",p);
    printf("Valeur de   *p:   %d \n",*p);

    return 0;
}
```

```
Valeur de   &i:   712dea34
Valeur de   i :   10
Valeur de   &p:   712dea38
Valeur de   p :   712dea34
Valeur de   *p:   10
```

Pointeurs: Notion

- Dans un programme on peut manipuler à la fois les objets **p** et ***p**. Ces deux manipulations sont très différentes.
- **Exemple:** :

```
int i = 3, j = 6;
int *p1, *p2;
        p1 = &i;
        p2 = &j;
(1)     *p1 = *p2;
(2)      p1 = p2;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Pointeurs: Notion

- **Exemple:** :

```
p1 = &i;
p2 = &j;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

(1)

```
*p1 = *p2;
```

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

(2)

```
p1=p2;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

Pointeurs: Arithmétique des pointeurs

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'operateurs arithmétiques classiques.
- Les seules opérations arithmétiques valides sont :
  - *Addition (+) d'un entier à un pointeur:* Le résultat est un pointeur de même type que le pointeur de départ ;
  - *Soustraction(-) d'un entier à un pointeur:* Le résultat est un pointeur de même type que le pointeur de départ ;
  - *Différence de deux pointeurs pointant tous deux vers des objets de même type*. Le résultat est un entier.
- *Somme de deux pointeurs* n'est pas autorisée.

Pointeurs: Arithmétique des pointeurs

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'operateurs arithmétiques classiques.
- Les seules opérations arithmétiques valides sont : **+,-**
- *Somme de deux pointeurs* n'est pas autorisée.
- Si i est un entier et p est un pointeur sur un objet de type type, l'expression **p+i** désigne un pointeur sur un objet de type type dont la valeur est égale à la valeur de p incrémentée de i * sizeof(type).
- Il en va de même pour la soustraction d'un entier à un pointeur, et pour les operateurs d' incrémentation et de décrémentation ++ et J J.

Pointeurs: Arithmétique des pointeurs

- **Exemple**:

```
Entier p1 = 2686628      p2 = 2686632
double p3 = 2686616      p4 = 2686624
Char   p5 = 2686615      p6 = 2686619
```

```c
#include <stdio.h>
#include <stdlib.h>

main()
{

int i = 3;
int *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("Entier p1 = %ld \t p2 = %ld\n",p1,p2);

double j = 3;
double *p3, *p4;
p3 = &j;
p4 = p3 + 1;
printf("double p3 = %ld \t p4 = %ld\n",p3,p4);

char k = 'a';
int *p5, *p6;
p5 = &k;
p6 = p5 + 1;
printf("Char   p5 = %ld \t p6 = %ld\n",p5,p6);
}
```

Pointeurs: Allocation dynamique

- Avant de manipuler un pointeur, et d'appliquer l'operateur d'indirection *, il faut l'initialiser.
- Par défaut, la valeur du pointeur est égale à une constante symbolique notée NULL définie dans **stdio.h**.
  - En général, cette constante vaut **0.**
- Le test p== NULL permet de savoir si le pointeur p pointe vers un objet.
- On peut initialiser un pointeur p :
  - **Affectation indirecte** de p l'adresse d'une autre variable.
  - **Affectation directe** d'une valeur à ***p**:
    - Il faut d'abord réserver à *p un espace-mémoire de taille adéquate.
    - L'adresse de cet espace-mémoire sera la valeur de **p**.

Pointeurs: Allocation dynamique

- La **gestion dynamique** de la mémoire en C se fait à l'aide de principalement deux fonctions de la bibliothèque standard :
  - **malloc**, pour l'allocation dynamique de mémoire ;
  - **free**, pour la libération de mémoire préalablement allouée avec **malloc**.
- Deux autres fonctions permettent de gérer plus finement la mémoire :
  - **calloc**, pour allouer dynamiquement de la mémoire, comme **malloc**, qui a préalablement été initialisée à 0 ;
  - **realloc**, pour modifier la taille d'une zone mémoire déjà allouée.
- Ces fonctions sont déclarées dans l'en-tête **<stdlib.h>.**

Pointeurs: Allocation dynamique

- L'allocation dynamique est faite par la fonction **malloc** de la librairie standard **stdlib.h**.
- **Syntaxe**:

malloc(**nombre-octets**);

- La fonction retourne un pointeur de type **char *** pointant vers un objet de taille **nombre-octets** octets.
- Pour initialiser des pointeurs vers des objets qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction **malloc** à l'aide d'un **cast**.
- L'argument **nombre-octets** est souvent donné à l'aide de la fonction **sizeof()** qui renvoie le nombre d'octets utilisés pour stocker un objet.

Pointeurs: Allocation dynamique

- **Exemple:**
- Pour initialiser un pointeur vers un entier:

```
1) #include <stdlib.h>
2) int *p;
3) p = (int*)malloc(sizeof(int));
```

- On aurait pu écrire également

```
p = (int*)malloc(4);
```

- Puisqu'un objet de type int est stocké sur 4 octets.
- La première écriture qui a l'avantage d'être portable.

Pointeurs: Allocation dynamique

- **Exemple:**

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p;
    p  = (int*)malloc(sizeof(int));
   *p  = 6;

    printf("Valeur de   &p:   %x \n",&p);
    printf("Valeur de   p :   %x \n",p);
    printf("Valeur de   *p:   %d \n",*p);

    return 0;
}
```

```
Valeur de   &p:   841b21b8
Valeur de   p :   12fc010
Valeur de   *p:   6
```

Pointeurs: Allocation dynamique

- **Exemple:**
  - **Avant l'allocation dynamique:**

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

  - **Avant l'allocation dynamique:**
    - L'allocation dynamique permet de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de *p.

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

Pointeurs: Allocation dynamique

- **Exemple:**
  - L'allocation dynamique permet de réserver à cette adresse un espace-mémoire composé de **4 octets** pour stocker la valeur de ***p**.

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

  - L'affectation ***p = i**; a enfin pour résultat d'affecter à ***p** la valeur de i

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

Pointeurs: Allocation dynamique

- La fonction **calloc** de la librairie **stdlib.h** a le même rôle que la fonction **malloc** mais elle initialise en plus l'objet pointe *p à zéro.
- **Syntaxe**:

calloc(nb-objets,taille-objets)

- si p est de type int*, l'instruction

```
p = (int*)calloc(N, sizeof(int));
```

⟺

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
    *(p + i) = 0;
```

- L'emploi de **calloc** est simplement plus rapide.

Pointeurs: Allocation dynamique

- A la fin d'utilisation d'un pointeur et lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement.
- Il faut libérer cette place en mémoire avec l'instruction **free**
- **Syntaxe:**

free(nom-du-pointeur);

- A toute instruction de type **malloc** ou **calloc** doit être associée une instruction de type **free**.

Plan

- **Séance 5 :Pointeurs**
  - Adresse et Valeur d'un objet
  - Notion
  - Arithmétique des pointeurs
  - Allocation dynamique
  - Utilisation dans:
    - Tableaux
    - Fonction passage par Valeur/Adresse
    - Chaines de caractères
    - Structures

Pointeurs: Pointeurs et tableaux

- **Pointeurs et tableaux à une dimension**
  - Tout tableau en C est en fait un **pointeur constant**.
  - Dans la déclaration **int tab[10];**
  - **tab** est un **pointeur constant** (**non modifiable**) dont la valeur est l'adresse du premier élément du tableau.
  - **tab** a pour valeur **&tab[0].**
  - C'est possible d'utiliser un pointeur initialisé à tab pour parcourir les éléments du tableau.

Pointeurs: Pointeurs et tableaux

- **Exemple:**

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define N 5
4   int tab[5] = {1, 2, 6, 0, 7};
5   main()
6   {
7   int i;
8   int *p;
9   p = tab;
10  printf(" Le tableu tab avec compteur:\n");
11      for (i = 0; i < N; i++)
12  {
13
14      printf(" %d \n",*p);
15      p++;
16  }
17  printf(" Le tableu tab avec pointeur:\n");
18      for (p=tab;p<&tab[5]; p++)
19  {
20
21      printf(" valeur %d @ : %d \n",*p,p);
22
23  }
24  }
25
```

```
Le tableu tab avec compteur:
1
2
6
0
7
Le tableu tab avec pointeur:
valeur 1 @ : 4206596
valeur 2 @ : 4206600
valeur 6 @ : 4206604
valeur 0 @ : 4206608
valeur 7 @ : 4206612

--------------------------------
```

Pointeurs: Pointeurs et tableaux

- La manipulation de tableaux, et non de pointeurs, possède certains inconvénients dus au fait qu'un **tableau est un pointeur constant**:
  - Impossible de créer de tableaux dont la taille est une variable du programme,
  - Impossible de créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d' éléments.
- Ces opérations deviennent possibles dés que l'on manipule des **pointeurs alloués dynamiquement**.

Pointeurs: Pointeurs et tableaux

- Pour créer un tableau d'entiers à n éléments ou n est une variable du programme:

```
1) #include <stdlib.h>
2) int n;
3) int *tab;
4) tab = (int*)malloc(n* sizeof(int));
5) free(tab);
```

- Si on veut en plus que tous les éléments du tableau tab soient initialises a zéro, on remplace l'allocation dynamique avec **malloc** par :

```
4) tab = (int*)calloc(n, sizeof(int));
```

Pointeurs: Pointeurs et tableaux

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

main() {

    int i,n;
    printf("donnez la dimension de la tab :");
    scanf("%d",&n);
    int *p;
    p = (int*)malloc(n*sizeof(int));

    printf("\n *******Representation comme pointeur*******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));

    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    p = (int*)calloc(n,sizeof(int));

    printf("\n ******* Representation comme pointeur\n*****");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));
    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    free(p);
}
```

Pointeurs: Pointeurs et tableaux

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

main() {

    int i,n;
    printf("donnez la dimension de
    scanf("%d",&n);
    int *p;
    p = (int*)malloc(n*sizeof(int));

    printf("\n *******Representation comme pointeur*******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));

    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    p = (int*)calloc(n,sizeof(i

    printf("\n ******* Represen
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas
    printf("\n *******Represent
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas

    free(p);
}
```

```
donnez la dimension de la tab :5
 *******Representation comme pointeur*******
 l'adtress de cas 0 est 3544304 sa valeur : 3553392
 l'adtress de cas 1 est 3544308 sa valeur : 3539140
 l'adtress de cas 2 est 3544312 sa valeur : 1751216738
 l'adtress de cas 3 est 3544316 sa valeur : 1096576361
 l'adtress de cas 4 est 3544320 sa valeur : 1631875184
```

```
*******Representation comme table *******
l'adtress de cas 0 est 3544304 sa valeur : 3553392
l'adtress de cas 1 est 3544308 sa valeur : 3539140
l'adtress de cas 2 est 3544312 sa valeur : 1751216738
l'adtress de cas 3 est 3544316 sa valeur : 1096576361
l'adtress de cas 4 est 3544320 sa valeur : 1631875184
```

Pointeurs: Pointeurs et tableaux

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

main() {

    int i,n;
    printf("donnez la dimension de la tab :");
    scanf("%d",&n);
    int *p;
    p = (int*)malloc(n*sizeof(int));

    printf("\n *******Representation comme pointeur*******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));

    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    p = (int*)calloc(n,sizeof(int));

    printf("\n ******* Representation comme pointeur\n*****");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));
    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    free(p);
}
```

Pointeurs: Pointeurs et tableaux

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

main() {

    int i,n;
    printf("donnez la dimensic
    scanf("%d",&n);
    int *p;
    p = (int*)malloc(n*sizeof(

    printf("\n *******Represer
    for(i=0;i<n;i++)
    printf("\n l'adtress de ca

    printf("\n *******Represer
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    p = (int*)calloc(n,sizeof(int));

    printf("\n ******* Representation comme pointeur\n*****");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));
    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    free(p);
}
```

```
******* Representation comme pointeur
*****
l'adtress de cas 0 est 3544336 sa valeur : 0
l'adtress de cas 1 est 3544340 sa valeur : 0
l'adtress de cas 2 est 3544344 sa valeur : 0
l'adtress de cas 3 est 3544348 sa valeur : 0
l'adtress de cas 4 est 3544352 sa valeur : 0
*******Representation comme table *******
l'adtress de cas 0 est 3544336 sa valeur : 0
l'adtress de cas 1 est 3544340 sa valeur : 0
l'adtress de cas 2 est 3544344 sa valeur : 0
l'adtress de cas 3 est 3544348 sa valeur : 0
l'adtress de cas 4 est 3544352 sa valeur : 0
```

Pointeurs: Pointeurs et tableaux

Q: Écrire une boucle pour remplir la table

```c
#include <stdio.h>
#include <stdlib.h>

main() {

    int i,n;
    printf("donnez la dimension de la tab :");
    scanf("%d",&n);
    int *p;
    p = (int*)malloc(n*sizeof(int));

    printf("\n *******Representation comme pointeur*******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));

    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    p = (int*)calloc(n,sizeof(int));

    printf("\n ******* Representation comme pointeur\n*****");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,(p+i),*(p+i));
    printf("\n *******Representation comme table *******\n");
    for(i=0;i<n;i++)
    printf("\n l'adtress de cas %d est %ld sa valeur : %d \n",i,p+i,p[i]);

    free(p);
}
```

Pointeurs: Pointeurs et tableaux

- Les éléments de tab sont manipulés avec l'opérateur d'indexation **[],** exactement comme pour les tableaux.
- Les deux différences principales entre un tableau et un pointeur sont:
  - Un pointeur doit toujours être initialise, soit par une allocation dynamique (**malloc**), soit par affectation d'une expression adresse, (**p = &i**) ;
  - Un tableau n'est pas une **Lvalue** ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas **l'arithmétique** (**on ne peut pas écrire tab++;).**

Pointeurs: Pointeurs et tableaux

- Un tableau à deux dimensions est, par définition, un tableau de tableaux. ou un pointeur vers un pointeur.

int tab[M][N];

  - **tab** est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier.
  - **tab** a une valeur constante égale à l'adresse du premier élément du tableau, &tab[0][0].
- De même tab[i], pour i entre 0 et M-1, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice i.
- tab[i] a donc une valeur constante qui est égale à &tab[i][0].
- Exactement comme pour les tableaux à n une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnes.
- On déclare un pointeur qui pointe sur un objet de type type * (deux dimensions) de la même manière qu'un pointeur:

type **nom-du-pointeur;

Pointeurs: Pointeurs et tableaux

- **Procédure de création un tableau de 2 dimensions:**
  - Avec un pointeur de pointeur une matrice à **k** lignes et **n** colonnes à coefficients entiers:

```
1) int k, n;
2) int **tab;
3) tab = (int**)malloc(k * sizeof(int*));
4) for (i = 0; i < k; i++)
        tab[i] = (int*)malloc(n * sizeof(int));
5)  for (i = 0; i < k; i++)
        free(tab[i]);
6)  free(tab);
```

Pointeurs: Pointeurs et tableaux

- **Exemple:**

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main() {
        int **A ;
        int  N,M,P,i,j;
        printf("entrer le nombre de ligne de la matrice A:");
        scanf("%d",&N);
        printf("entrer le nombre de colonne de la matrice A :");
        scanf("%d",&M);

        A=(int**)malloc(N*sizeof(int*));
        for(i=0;i<N;i++)
                A[i]=(int*)malloc(M*sizeof(int));
        //remplissage de la matrice A
        printf("remplissage de la matrice A.\n");
        printf("\n");
        for(i=0;i<N;i++){
                for(j=0;j<M;j++){
                        printf("entrer la valeur d'indice %d,%d :",i,j);
                        scanf("%d",*(A+i)+j);
                        }
                printf("\n");
                }
        //Affichage de la matrice A
        printf("\nla matrice A est:\n");
        for(i=0;i<N;i++){
                for(j=0;j<M;j++)
                        printf("%6d",*(*(A+i)+j));
                printf("\n");
                }

    }
```

Pointeurs: Pointeurs et tableaux

- **Exemple:**

```
entrer le nombre de ligne de la matrice A:2
entrer le nombre de colonne de la matrice A :2
remplissage de la matrice A.

entrer la valeur d'indice 0,0 :1
entrer la valeur d'indice 0,1 :0

entrer la valeur d'indice 1,0 :0
entrer la valeur d'indice 1,1 :1


la matrice A est:
1       0
0       1
```

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main() {
        int **A ;
        int  N,M,P,i,j;
        printf("entrer le nombre
        scanf("%d",&N);
        printf("entrer le nombre
        scanf("%d",&M);

        A=(int**)malloc(N*sizeof(
        for(i=0;i<N;i++)
                A[i]=(int*)malloc
        //remplissage de la matr
        printf("remplissage de la
        printf("\n");
        for(i=0;i<N;i++){
                for(j=0;j<M;j++){
                        printf("entrer la valeur d'indice %d,%d :",i,j);
                        scanf("%d",*(A+i)+j);
                        }
                printf("\n");
                }
        //Affichage de la matrice A
        printf("\nla matrice A est:\n");
        for(i=0;i<N;i++){
                for(j=0;j<M;j++)
                        printf("%6d",*(*(A+i)+j));
                printf("\n");
                }

    }
```

Plan

- **Chapitre 5 :Pointeurs**
  - Adresse et Valeur d'un objet
  - Notion
  - Arithmétique des pointeurs
  - Allocation dynamique
  - Utilisation dans:
    - Tableaux
    - Fonction passage par Valeur/Adresse
    - Chaines de caractères
    - Structures

Passage des arguments et variables locales

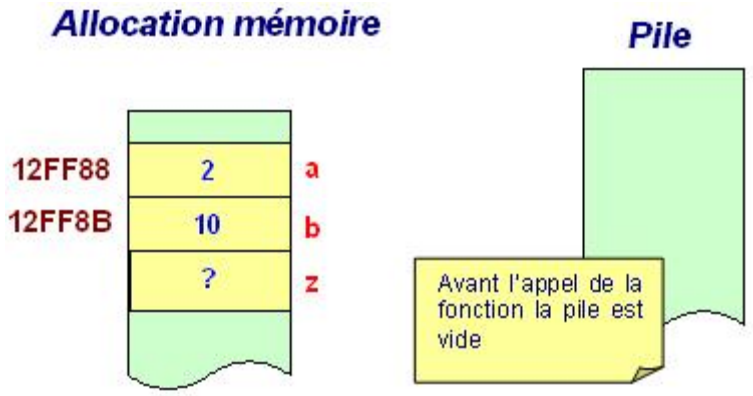
- Les fonctions utilisent un **espace d'allocation** de mémoire située sur la **pile** d'exécution du programme.
- Cet espace d'allocation reste associé à la fonction jusqu'à ce que celle-ci se termine. Dès lors, l'espace devient automatiquement disponible pour être réutilisé.
- Chaque **paramètre (input)** de fonction, ainsi que les **variables internes**, sont stockés sur cet espace d'allocation.
- Ces deux valeurs sont alors appelés, **variables locales**.
- Cette **pile** est différente de l'allocation **mémoire statique**, ce qui sous-entends que les valeurs des arguments passées à la fonction vont être copiées dans les paramètres et se retrouvent donc sur la pile.

Passage des arguments et variables locales

- Les changements effectués sur ces variables locales (donc sur la pile), ne sont pas répercutées sur les valeurs des arguments.
- Chaque entité possède son propre espace mémoire.
- Une fois la fonction terminée, l'espace d'allocation de la pile est supprimée pour cette fonction, et donc, les valeurs locales sont définitivement perdues.
- Les valeurs locales sont donc des variables dynamiques qui possèdent, malgré tout, une identité, c'est-à-dire un nom.

```
//----------------------------------------
double puissance(double y, unsigned x);
//----------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//----------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//----------------------------------------
```

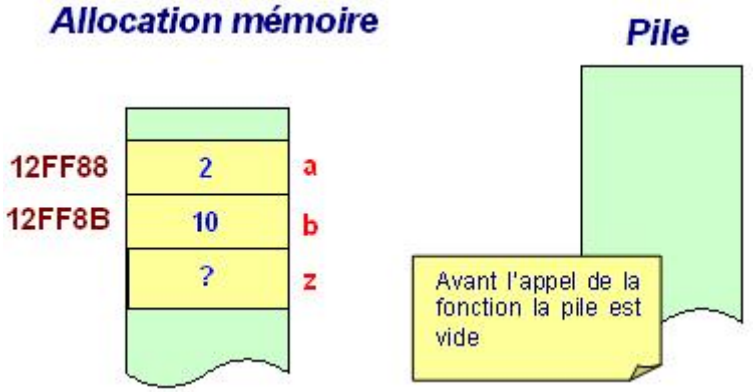
Prochaine ligne d'exécution



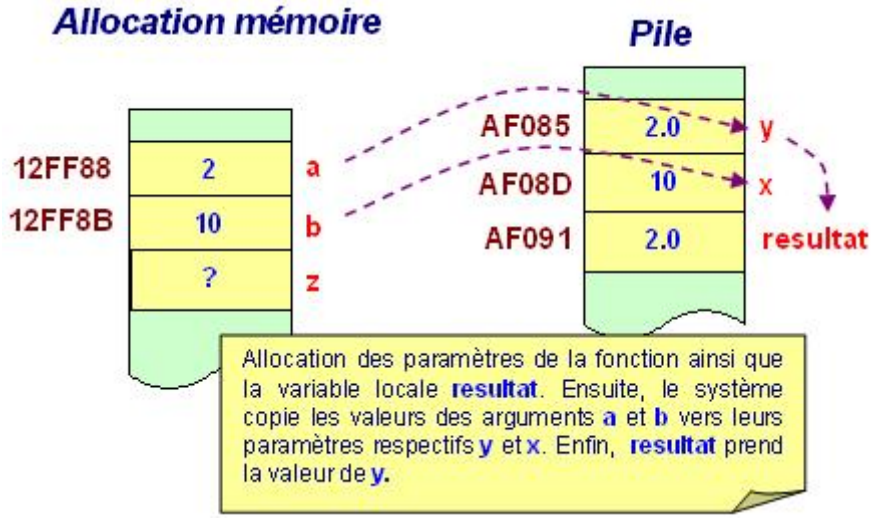
Passage des arguments et variables locales

```
//----------------------------------------
double puissance(double y, unsigned x);
//----------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//----------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//----------------------------------------
```

Prochaine ligne d'exécution

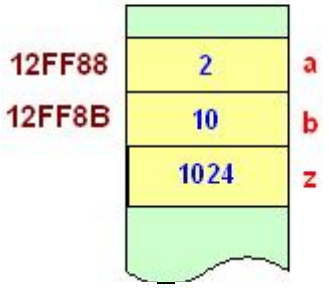


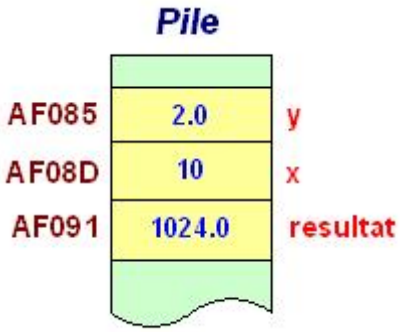

```
//----------------------------------------
double puissance(double y, unsigned x);
//----------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//----------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//----------------------------------------
```



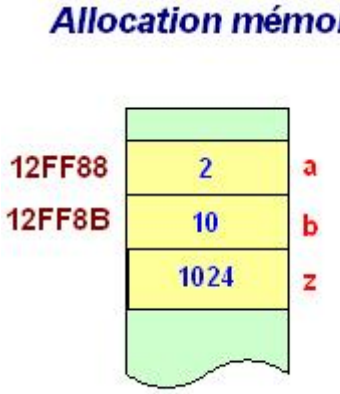
Passage des arguments et variables locales

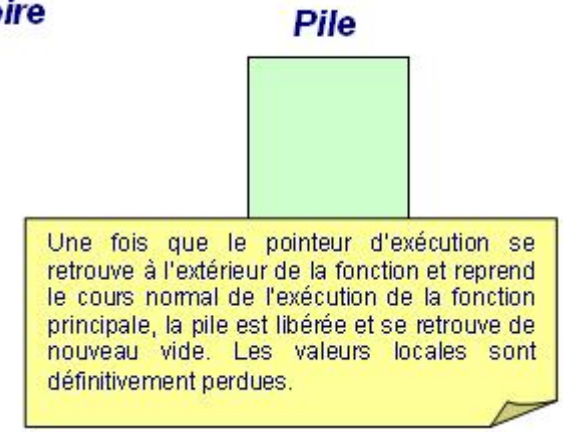
```
//---------------------------------------
double puissance(double y, unsigned x);
//---------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//---------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//---------------------------------------
```






```
//---------------------------------------
double puissance(double y, unsigned x);
//---------------------------------------
int main()
{
  int a = 2, b = 10;
  int z;
  z = puissance(a, b);
  return 0;
}
//---------------------------------------
double puissance(double y, unsigned x)
{
  double resultat = y;
  if (x==0) return 1.0;
  for (int i=1; i<x; i++) resultat *= y;
  return resultat;
}
//---------------------------------------
```





Transmission par Valeur et par Adresse

- **Le passage de paramètres permet à une fonction de pouvoir traiter des données qui ne sont pas définies dans son corps.**
- **Les données sont passées à la fonction lors de son appel.**
- **Il existe globalement deux techniques de passage de paramètres :**
  - ***Soit par valeur, et c'est la technique que nous venons d'utiliser.***
  - ***Soit par adresse , ce qui permet dans ce cas là, de se connecter directement (ou indirectement) aux variables de la fonction principale, c'est-à-dire aux arguments.***

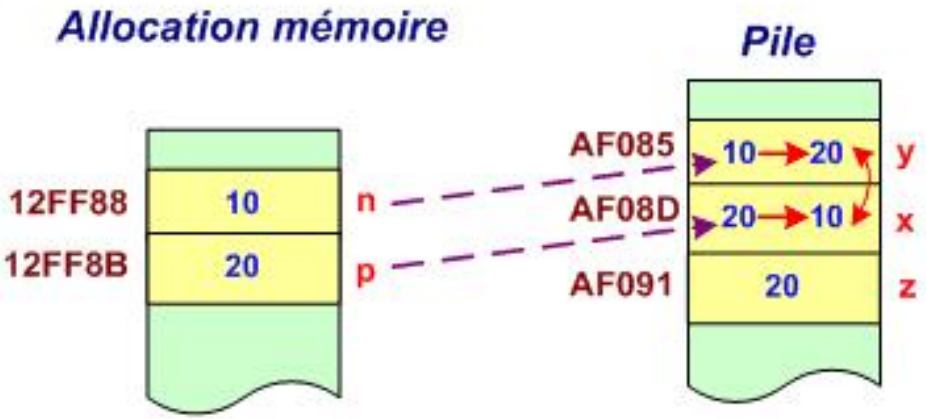
Transmission par valeur

- **Le type de transmission qui est le plus couramment utilisé.**
- **La fonction manipule les copies locales des arguments.**
- **Les fonctions n'obtiennent que les valeurs de leurs paramètres passés et elles n'ont pas accès au contenu des variables elles-mêmes.**
- **Les paramètres d'une fonction sont des variables locales qui sont initialisées automatiquement par les valeurs indiquées par les arguments lors de l'appel.**
- **A l'intérieur de la fonction, il est possible de changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes .**

Transmission par valeur

- **Exemple:**
  - En prenant le passage par valeurs comme c'est le cas ici, les seuls échanges proposés se situent au niveau des variables locales à la fonction, sans qu'il y ait de répercutions sur les arguments n et p .
  - Dans la plupart des cas, c'est très bien, puisque les arguments sont protégés de toute mauvaise utilisation.

```
//-------------------------
void echange(int x, int y)
{
  int z=x;
  x=y; y=z;
}
//-------------------------
int main()
{
  int n=10, p=20;
  echange(n, p);
  return 0;
}
//-------------------------
```



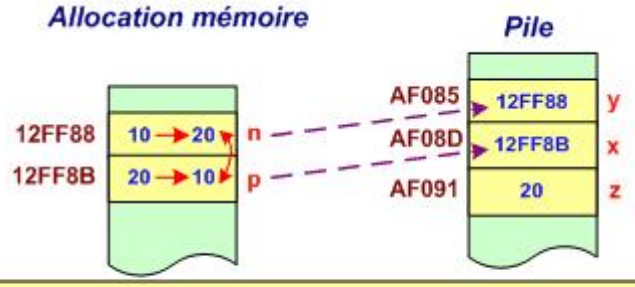
Transmission par adresse

- **Ce passage permet à la fonction appelée de pouvoir modifier le contenu de la variable passée en paramètre.**
- **Il existe deux techniques pour résoudre ce problème :**
  - ***Soit indirectement en utilisant les pointeurs.***
    - ***Cas de C et C++***
  - ***Soit directement en utilisant les références.***
    - ***Cas de C++***

Transmission par adresse

- **Avec des pointeurs comme paramètres (C et C++)**

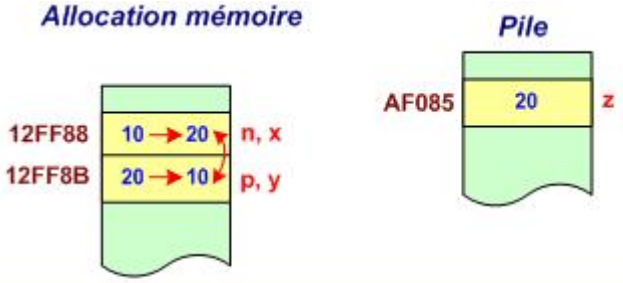
```
//---------------------------
void echange(int *x, int *y)
{
  int z = *x;
  *x = *y; *y = z;
}
//---------------------------
int main()
{
  int n=10, p=20;
  echange(&n, &p);
  return 0;
}
//---------------------------
```



L'échange s'effectue bien sur les bonnes variables, c'est-à-dire sur les arguments de la fonction principale. Toutefois, la syntaxe demeure relativement lourde puisqu'il faut déréférencer systématiquement. Enfin, à l'appel de la fonction, il est nécessaire de donner les adresses des variables à traiter puisqu'il s'agit de pointeurs.

- **Avec des références comme paramètres (C++)**

```
//---------------------------
void echange(int &x, int &y)
{
  int z = x;
  x = y; y = z;
}
//---------------------------
int main()
{
  int n=10, p=20;
  echange(n, p);
  return 0;
}
//---------------------------
```



Ici aussi, l'échange entre les arguments s'effectue bien. De plus, la syntaxe est extrêmement simple. Par ailleurs, les références se connectent directement sur les arguments (alias), ce qui évite d'utiliser des variables locales supplémentaires. Le gain de temps et la souplesse d'emploi sont prépondérants.

Conclusion

- **A l'aide de ces différents scénario, nous pouvons presque conclure que :**
  - **Lorsque nous devons récupérer une valeur sans changer le contenu de l'argument, il faut alors proposer une transmission par valeur, et il suffit alors de faire une déclaration classique des paramètres.**
  - **Lorsque nous devons modifier directement le contenu de l'argument, il faut cette fois-ci proposer une transmission par adresse en prenant si possible une référence pour que l'argument soit directement connecté.**

Types composes : Structure

- Une **structure** est une suite finie d'objets de types différents. Contrairement aux tableaux.
- Les différents éléments d'une structure n'occupent pas nécessairement des zones contigües en mémoire.
- Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur.
- **Syntaxe**:

1) Déclaration:

```
struct modele
{       type-1 membre-1;
        type-2 membre-2;
        .........
         type-n membre-n; }
```

2) Utilisation:

```
struct modele objet;
```

Types composes : Structure

- Lorsqu'une structure est définie, elle crée un type défini par l'utilisateur, mais pas de stockage est allouée:
  - **Déclaration** :
```
struct person { char name[50];
                int city_no;
                float salary; };
```
  - **Utilisation**; Dans la fonction main :
```
struct person p1, p2, p[20];
```

- On accède aux différents membres d'une structure grâce a l'operateur membre de structure, note "**.**".
- Le i-éme membre de objet est désigne par l'expression **objet.membre-i**
- On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type type-i.

Types composes : Structure

```c
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */
   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
   /* print Book1 info */

   printf( "\nBook 1\t\t Title : %s\n", Book1.title);
   printf( "\t \t Author : %s\n", Book1.author);
   printf( "\t \t Subject : %s\n", Book1.subject);
   printf( "\t \t Book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "\nBook 2\t\t Title : %s\n", Book2.title);
   printf( "\t \t Author : %s\n", Book2.author);
   printf( "\t \t Subject : %s\n", Book2.subject);
   printf( "\t \t Book_id : %d\n", Book2.book_id);
}
```

Types composes : Structure

```c
#include <stdio.h>
#include <string.h>

struct Books
{
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
};
int main( )
{
    struct Books Book1;
    struct Books Book2;
    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;
    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;
    /* print Book1 info */

    printf( "\nBook 1\t\t Title : %s\n", Book1.title);
    printf( "\t \t Author : %s\n", Book1.author);
    printf( "\t \t Subject : %s\n", Book1.subject);
    printf( "\t \t Book_id : %d\n", Book1.book_id);

    /* print Book2 info */
    printf( "\nBook 2\t\t Title : %s\n", Book2.title);
    printf( "\t \t Author : %s\n", Book2.author);
    printf( "\t \t Subject : %s\n", Book2.subject);
    printf( "\t \t Book_id : %d\n", Book2.book_id);
}
```

```
Book 1          Title : C Programming
                Author : Nuha Ali
                Subject : C Programming Tutorial
                Book_id : 6495407

Book 2          Title : Telecom Billing
                Author : Zara Ali
                Subject : Telecom Billing Tutorial
                Book_id : 6495700
```

Types composes : Champs de bits

- Il est possible en C de spécifier la longueur des champs d'une structure au bit prés si ce champ est de type entier (int ou unsigned int).
- Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration.
- **Exemple**:

```
struct registre
{
   unsigned int actif : 1;
   unsigned int valeur : 31;
};
```

  - **registre** possède deux membres, ***actif*** qui est code sur un **1bit**, et ***valeur*** qui est code sur **31 bits**.
  - Tout objet de type **struct registre** est donc code sur 32 bits.

Types composes : Champs de bits

```
struct registre
{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

- **L'ordre** dans lequel les champs sont places à l'intérieur de ce mot de 32 bits **dépend de l'implémentation**.
- Le champ actif de la structure ne peut prendre que les valeurs 0 et 1.
  - Si r est un objet de type **struct registre:**

  l'operation **r.actif += 2**; ne modifie pas la valeur du champ.
- La taille d'un champ de bits **doit être inferieure** au nombre de bits d'un entier.
- Un champ de bits **n'a pas d'adresse** ; on ne peut donc pas lui appliquer l'operateur &.

Types composes : Champs de bits

- **Exemple:**

```c
#include <stdio.h>
#include <string.h>

struct
{
  unsigned int age : 3;
} Age;

main( )
{
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );

}
```

```
Sizeof( Age ) : 4
Age.age  : 4
Age.age  : 7
Age.age  : 0

--------------------------------
```

Types composes : Unions

- Une **union** désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire.
- Une **union** permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types.
- Si les membres d'une union sont de longueurs différentes, **la place réservée en mémoire** pour la représenter correspond a la taille **du membre le plus grand**.
- Les **déclarations** et les opérations sur les objets de type **union** sont les mêmes que celles sur les objets de type **struct**.

Types composes : Union

- **Exemple**: la variable jour de type union jour peut être soit un entier, soit un caractère.

```
union jour
{
  char lettre;
  int numero;
};
```

- Les **unions** peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents ″**mais en général de même taille**″.

Types composes : Union Vs Structure

- **Exemple:**

```c
#include <stdio.h>
#include <stdlib.h>
    union job { //defining a union
        char name[32];
        float salary;
        int worker_no;
         }u;
    struct job1 {
        char name[32];
        float salary;
        int worker_no; }s;

main(){
    printf("Taille en memoire d'un union = %d  Octets ",sizeof(u));
    printf("\n Taille en memoire d'une structure = %d  Octets", sizeof(s));
}
```

```
Taille en memoire d'un union       = 32  Octets
Taille en memoire d'une structure  = 40  Octets
--------------------------------
Process exited after 0.9424 seconds with return value 46
```

Types composes : Union Vs Structure

- Il y a une différence dans l'allocation de mémoire entre union et structure .
  - La quantité de mémoire nécessaire pour stocker une structure de variables est la somme de la taille de la mémoire de tous les membres.

  - La mémoire nécessaire pour stocker **une variable union est la mémoire requise pour le plus grand élément** d'une union.

Types composes : Énumérations

- Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre.
- Un objet de type énumération est défini par enum et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

enum modele {constante-1, constante 2,…, constante-n};

- Les objets de type enum sont représentes comme des int.
- Les valeurs possibles **constante-1, constante-2, …, constante-n** sont codées par des entiers de **0** à **n-1**.
- **Exemple**: booléen défini dans le programme suivant associe l'entier 0 à la valeur faux et l'entier 1 à la valeur vrai.

Types composes : Énumérations

- **Exemple:**

```c
#include <stdio.h>

enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
main(){
     enum week today;
     today=wednesday;
     printf("%d day\n",today+1);
     printf("%d day",today-2);

   }

```

```
4 day
1 day
--------------------------------
```

Types composes : typedef

- Définition de types composes avec **typedef**
- Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type compose à l'aide de typedef :

```
typedef type synonyme;
```

- **Exemple :**

```
struct complexe
{
 double reelle;
 double imaginaire;
};
```

- **typedef struct complexe** complexe;

Pointeurs: Pointeurs et Chaine

- Une **chaîne de caractères** était un tableau à une dimension d'objets de type char, se terminant par le caractère nul '**\0**'.
- Toute **chaîne de caractères** est manipulée à l'aide d'un pointeur sur un objet de type **char**.
- Une chaîne est définie par

char *chaine;

- Les affectations sont possibles:

chaine = "ceci est une chaine";

- Toute opération valide sur les pointeurs, comme l'instruction **chaine++.**

Pointeurs: Pointeurs et Chaine

- **Exemple:**

```c
#include <stdio.h>
main()
{
int i;
char *chaine;
chaine = "chaine de caracteres";
for (i = 0; *chaine != '\0'; i++)
chaine++;
printf("nombre de caracteres = %d\n",i);
}
```

```
nombre de caracteres = 20
```

Pointeurs: Pointeurs et Chaine

- **Fonctions de manipulation de chaîne:**
- Ces fonctions travaillent toujours sur des adresses:
  - On transmet jamais à une fonction la valeur d'une chaine, mais seulement son adresse.
  - On transmet ainsi un **pointeur** sur son **premier caractère**.
- Les adresses son toujours de type **char *.**
- Des fonctions de traitement des chaînes de caractères sont disponibles dans les bibliothèques standards:

  scanf, **printf** en utilisant **%s** dans le format
- **scanf** prend une adresse en argument (&x), une chaîne de caractères étant un tableau (ie, l'adresse du premier élément), il n'y a pas de &.
- **puts: puts(MACHAINE);** est équivalent à printf("%s \n",**TXT**);
- **gets: gets(MACHAINE);** lit une ligne jusqu'au retour chariot et remplace le '\n' par '\0' dans l'affectation de la chaîne.

Pointeurs: Pointeurs et Chaine

- **Exemple:**

```
Chaine de Caractere: ABCDEFGH

A   dans l'adress 4210688
B   dans l'adress 4210689
C   dans l'adress 4210690
D   dans l'adress 4210691
E   dans l'adress 4210692
F   dans l'adress 4210693
G   dans l'adress 4210694
H   dans l'adress 4210695
Nombre de Caractere est :8
```

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   main() {
5
6   char *st;
7   int i=0;
8   st="ABCDEFGH";
9   printf("\n Chaine de Caractere: %s \n",st);
10  for (st;*st!='\0';st++)
11  {printf("\n %c  dans l'adress %ld",*st,st);
12  i++;
13  }
14  printf("\n Nombre de Caractere est :%d",i);
15
```

Pointeurs: Pointeurs et Chaine

- **Fonctions de manipulation de chaîne:**
  1) **Fonction Longueur**
  2) **Fonctions de Copie**
  3) **Fonctions de Concaténation**
  4) **Fonctions de Comparaison**
  5) **Fonctions de Recherche**
  6) **Fonctions de Conversion de chaîne en nombre**

Pointeurs: Fonction longueur

- **Fonction strlen :**
  - La fonction **strlen** fournit la longueur de la chaine dont on lui a transmis l'adresse en argument.
  - La longueur correspond au nombre de caractères trouvés depuis l'adresse indiquée jusqu'au premier caractère de code nul.
  - Ce dernier n'étant pas pris en compte.
  - **Prototype :**

```
int strlen(char*  but);
```

    - **but**: Adresse de chaine
    - **valeur de retour** : Entier
  - **Exemple :**

1) `strlen("bonjour") ; /* retournera 7 */`
2) `char * adr = "salut" ;`
   `strlen(adr) ; /* retournera 5 */`

Pointeurs: Fonctions de copie

- **Fonction strcpy :**
- La fonction recopie à l'adresse d'une première chaine, l'adresse d'une seconde chaine de caractères avec son caractère de fin de chaine ('**\0**').
  - **Prototype :**

    char * strcpy(char* but, **const** char * source)
    - **but**: Adresse à laquelle sera recopiée la chaine source
    - **source**: Adresse de la chaine à recopier
    - **valeur de retour** : Adresse but
  - **Exemple:**
    - **strcpy (ch1, "") ;**
    - La chaine se trouvant à l'adresse **ch1** est maintenant une chaine vide

Pointeurs: Fonctions de copie

Fonction strncpy :

- La fonction recopie à l'adresse d'une première chaine, l'adresse d'une seconde chaine de caractères mais en limitant la copie à un nombre de caractères précis.
- Ce nombre est donné en argument. Contrairement à **strcpy**, le caractère de fin de chaine ('\0') n'est pas rajouté automatiquement dans tous les cas.
  - **Prototype :**

```
char * strncpy(char*  but, const char * source, size_t longueur)
```

    - But : Adresse à laquelle sera recopiée la chaine source
    - Source : Adresse de la chaine à recopier
    - longueur: Nombre maximal de caractères recopies, y compris l'éventuel caractère '\0'
    - **valeur de retour :** Adresse but
  - **Exemples :**
    - **char ch [10] ; strcpy (ch, "abc", 10) ; strcpy (ch, "abc", 4) ;**
    - **strcpy (ch, "abc", 3) ; /* ch contiendra seulement a, b et c. */**
    - **strcpy (ch, "abc", 2) ; /* ch contiendra seulement a et b. */**

Pointeurs: Fonctions de concaténation

- **Fonctions de concaténation:**
- Les fonctions de **concaténation** permettent de mettre bout à bout deux chaines de caractères afin de n'en former qu'une seule.
- Il existe en C **deux fonctions** pour réaliser ce travail :
  - **strcat**
  - **strncat**
- Les deux fonctions sont très similaires, la seule différence réside dans le fait que **strncat** possède un troisième argument qui permet de limiter le nombre de caractères à concaténer (identiquement à **strncpy**).

Pointeurs: Fonctions de concaténation

- **Fonction strcat :**
- La fonction recopie à la fin d'une première chaine une seconde chaine.
  - **Prototype :**

    char * strcat(char* but, char * source)
    - **But : Adresse de la chaine réceptrice**
    - **Source : Adresse de la chaine à concaténer**
    - **valeur de retour : Adresse but**
- La fonction recopie la chaine située à l'adresse *source à la fin de la chaine d'adresse but*, *c'est-à-* dire à partir de son zéro de fin; ce dernier se trouve donc remplacé par le premier caractère de la chaine d'adresse source.
  - **Exemple :**
  - **char * ch1, * ch2 = "";**
  - **strcat(ch1, "");** /* la chaine à l'adresse ch1 est inchangée */
  - **strcat(ch1, ch2);** /* la chaine à l'adresse ch1 est inchangée */

Pointeurs: Fonctions de concaténation

- **Fonction strncat :**
- La fonction recopie à la fin d'une première chaine une seconde chaine en limitant cette dernière à un nombre de caractères donnés en argument.
  - **Prototype :**

```
char * strncat(char*  but, char * source, size_t longueur)
```

    - But : Adresse de la chaine réceptrice
    - Source : Adresse de la chaine à concaténer
    - longueur: Nombre maximal de caractères concaténés
    - valeur de retour : Adresse but
- La fonction recopie la chaine située à *l'adresse source à la fin de la chaine d'adresse but*, *c'est-à-* dire à partir de son zéro de fin.
- **\0** si se trouve donc sera remplacé par le premier caractère de la chaine d'adresse source.
- Le nombre de caractères recopiés est limité à *longueur*.

Pointeurs: Fonctions de concaténation

- **Fonction strncat :**
- A la différence de strncpy, strncat place un caractère de fin de chaine même si aucun caractère de fin de chaine n'a été trouvé parmi les caractères à concaténer.
  - **Exemple :**

```
char * ch1 = "salut";
char * ch2 = " Jack";
strncat(ch1, "", 10);
/* la chaine à l'adresse ch1 est inchangée */
strncat(ch1, ch2, 10);
/* la chaine à l'adresse ch1 est inchangée */
strncat(ch1, " Ben", 0);
/* la chaine à l'adresse ch1 est inchangée */
```

Pointeurs: Fonctions de comparaison

- **Fonctions de comparaison**
- Pour comparer des chaines de caractères, il faut utiliser des fonctions spécifiques.
- Contrairement aux données de type de base les opérateurs relationnels classiques**(>,<,=)** ne peuvent être employés.
- Les fonctions permettent de comparer des chaines de caractères selon leur ordre alphabétique (ordre de la table ASCII).
- Ces fonctions sont les suivantes :
  - **strcmp**
  - **strncmp**

Pointeurs: Fonctions de comparaison

- **Fonction strcmp :**
- Cette fonction compare la première chaine à la seconde et retourne une valeur entière en retour qui indique l'ordre des deux chaines.
  - **Prototype :**

```
char * strcmp(const char*  chaine1, const char * chaine2 )
```

  - **chaine1** : Adresse de la première chaine
  - **chaine2** : Adresse de la deuxième chaine
  - **Valeur de retour:**

    < 0 : si la chaine d'adresse **chaine1** arrive **avant** la **chaine2**
    > 0 : si la chaine d'adresse **chaine1** arrive **après** la **chaine2**
    =0 : si les deux chaines sont identiques

Pointeurs: Fonctions de comparaison

Fonction strcmp :

- Cette fonction compare la première chaine à la seconde et retourne une valeur entière en retour qui indique l'ordre des deux chaines.
  - **Prototype :**

```
char * strcmp(const char*  Chaine1, const char * chaine2 )
```

  - **Exemple :**

```
strcmp("bonjour", "monsieur");        /* négatif */
strcmp("paris2", "paris10");  /* négatif, 1 vient avant le 2 */
strcmp("bonjour", "bonjour"); /* nul */
strcmp("Paris", "paris");    //negatif, les minuscules se placent  apres les
                             //majuscules
strcmp("PARIS", "paris");     /* idem*/
strcmp("ré", "rat");          /* positif */
strcmp("ré", "rue");          /* positif   */
```

Pointeurs: Fonctions de comparaison

Fonction strcnmp :

- Même comportement que **strncmp** mais en limitant la comparaison des deux chaines à un nombre maximal de caractères.
  - **Prototype :**

```
int strncmp (const char * chaine1, const char * chaine2, size_t longueur)
```

- chaine1 :Adresse de la première chaine
- chaine2 : Adresse de la deuxième chaine
- Longueur : Nombre maximal de caractères soumis à la comparaison
- **Valeur de retour**:

  < 0 : si la chaine d'adresse **chaine1** arrive **avant** la **chaine2**

  > 0 : si la chaine d'adresse **chaine1** arrive **après** la **chaine2**

  =0 : si les deux chaines sont identiques

Pointeurs: Fonctions de comparaison

- **Fonction strcnmp :**
- Même comportement que strncmp mais en limitant la comparaison des deux chaines à un nombre maximal de caractères.
  - **Prototype :**

```
int strncmp (const char * chaine1, const
  char * chaine2, size_t longueur)
```

  - **Exemples :**

```
strcmp("bonjour", "bon", 12);
 /* positif "bon" avant "bonjour"  */
strcmp("bonjour", "bon", 4);   /* idem                    */
strcmp("bonjour", "bon", 2);   /* idem                    */
```

Pointeurs: Fonctions de Recherche

- **Fonctions de recherche**
- La bibliothèque standard propose aussi des fonctions de recherche de la première occurrence d'un caractère ou sous-chaine dans une chaine de caractères et qui sont :
  - Recherche d'un caractère : **strchr** et **strrchr**
  - Recherche d'une sous-chaine : **strstr**
  - Recherche d'un des caractères appartenant à un ensemble de caractères : **strpbrk**
- Toutes ces fonctions retournent en résultat *l'adresse* de l'information cherchée si elle est trouvée, le pointeur *nul* autrement.
- Ou encore des fonctions qui permettent d'éclater une chaine en plusieurs parties. Exemple : **strtok.**

Pointeurs: Fonctions de Recherche

- **Fonction strchr :**
- Recherche la première occurrence d'un caractère dans une chaine de caractères.
  - **Prototype :**

```
char * strchr (const char * chaine, int c)
```

- **chaine : Adresse de la première chaine**
- **c : Caractère recherché après conversion en unsigned char**
- **Valeur de retour: Adresse du premier caractère c trouvé s'il existe, pointeur NULL sinon**
  - **Exemple :**

```
strchr("bonjour", 'o');
 /* adresse du premier 'o' de la chaine "bonjour" */
strchr("bonjour", 'a'); /* fournit la valeur NULL   */
```

Pointeurs: Fonctions de Recherche

- **Fonction strrchr :**
- Même comportement que **strrchr** mais en effectuant la recherche à partir de la fin de la chaine et non à partir du début de la chaine comme **strchr**.
- **Prototype :**

char * strrchr (const char * chaine, int c)

- **chaine : Adresse de la première chaine**
- **c : Caractère recherché après conversion en unsigned char**
- **Valeur de retour: Adresse du premier caractère c trouvé s'il existe, pointeur NULL sinon**

Pointeurs: Fonctions de Recherche

- **Fonction strstr :**
- Recherche la première occurrence d'une sous chaine dans une chaine de caractères.
  - **Prototype :**

```
char * strstr (const char * chaine1, const char * chaine2)
```

- chaine1 : Adresse de la première chaine
- chaine2 : Adresse de la sous-chaine recherchée
- **Valeur de retour**: Adresse de la première occurrence complète de la sous-chaine cherchée, si elle existe, sinon le pointeur NULL
  - **Exemple :**

```
strstr("recherche", 'ch');
/* retourne l'adresse du 3 ème caractère de la chaine */
strstr("recherche", 'cha');  /* retourne la valeur NULL
```

Pointeurs: Fonctions de Recherche

- **Fonction strtok :**
- Eclate une chaine en plusieurs sous-chaines, sachant que ces dernières sont séparées par un ou plusieurs délimiteurs.
  - **Prototype :**

char * strtok (const char * chaine, const char * delimiteurs)

- **chaine1 :** Adresse de la première chaine
- **delimiteurs** Adresse d'une chaine contenant les caractères délimiteurs
- **Valeur de retour :** Adresse de la première sous-chaine de la chaine délimitée (avant et après) par des caractères délimiteurs si elle existe, le pointeur NULL sinon
- Cette fonction recherche donc à partir de l'adresse chaine, le premier caractère différent des caractères appartenant à la chaine délimiteurs.
- Si aucun caractère n'est trouvé la fonction retourne NULL.

Pointeurs: Fonctions de Recherche

- **Fonction strtok :**
- Eclate une chaine en plusieurs sous-chaines, sachant que ces dernières sont séparées par un ou plusieurs délimiteurs.
  - **Prototype :**

```
char * strtok (const char * chaine, const char * delimiteurs)
```

- **Exemple:**

```
char message1[] = "Voici un simple texte";
char message2[] = "/usr/lib/cron/";
printf("message1=%s\n",strtok(message1," "));
printf("message2=%s\n",strtok(message2,"/"));
```

message1=Voici **message2**=usr

Pointeurs: Fonctions de Conversion

- **Fonctions de conversion de chaîne en nombre**
  - Ces fonctions permettent de convertir une chaine ou une partie de la chaine en un nombre.
  - Dans les deux cas les caractères qui ne peuvent pas être convertis sont ignorés.
- **Les fonctions de conversion sont :**
  - **Bibliothèque <string.h> :**
    - **strtol** → chaine en long
    - **strtoul** → chaine en unsigned long
    - **strtod** → chaine en double
  - **Bibliothèque <strlib.h> :**
    - **atoi** → chaine en entier
    - **atol** → chaine en long
    - **atof** → chaine en flottant

Pointeurs: Fonctions de Conversion

- **Prototypes :**
- **double atof(const char * chaine)**
- **chaine :** Chaine à convertir
- **Valeur de retour:** Résultat de la conversion de la chaine en double

- **long atol(const char * chaine)**
- **chaine:** Chaine à convertir
- **Valeur de retour:** Résultat de la conversion de la chaine en long

- **int atoi(const char * chaine)**
- **chaine:** Chaine à convertir
- **Valeur de retour:** Résultat de la conversion de la chaine en int

Plan

- **Chapitre5 :Fonctions**
  - Modularisation de programmes
  - Notion de blocs et la portée des identificateurs
  - Déclaration et définition de fonctions
  - Renvoyer un résultat
  - Paramètres d'une fonction
  - Passage de variables :par valeur, par adresse
  - Récursivité

Fonctions

- Dès qu'un groupe de lignes revient plusieurs fois on les regroupe dans une fonction
- Une fonction se reconnaît à ses ()
- Une fonction en C est assez proche de la notion mathématique de fonction:
- Exemples : y = sqrt(x) ; Z = pgcd(A,B) ;

Fonctions

- Intérêt des fonctions
- Lisibilité du code
- Réutilisation de la fonction
- Tests facilités
- Évolutivité du code
- Plus tard : les fonctions fonctions dans des fichiers fichiers séparés séparés du main.c
- Nb : une fonction fonction peut faire appel à d autres ’autres fonctions fonctions
- dans son code
- dans ses arguments