

## Programmation II en Langage C

### Plan du cours

Semestre I (Programmation I)	Semestre II (Programmation II)
<p><b>Chap1.</b> Introduction / Composants d'un programme C</p> <p><b>Chap2.</b> Types de base / Lecture et écriture formatée des données</p> <p><b>Chap3.</b> Opérateurs et expressions logiques</p> <ul style="list-style-type: none"><li>- Les opérateurs standard du C, affectation incrémentation, priorité des opérateurs</li><li>- Les expressions arithmétiques, relationnelles et logiques</li></ul> <p><b>Chap4.</b> Structures alternatives (conditionnelles) et structures répétitives (boucles)</p> <p><b>Chap5.</b> Tableaux à une dimension</p> <p><b>Chap6.</b> Fonctions et procédures :</p> <ul style="list-style-type: none"><li>- Déclaration des arguments</li><li>- Variables locales et variables globales</li><li>- Passages par valeur et par adresse</li></ul>	<p><b>Chap7.</b> Les pointeurs</p> <ul style="list-style-type: none"><li>- arithmétique des pointeurs</li><li>- allocation dynamique de la mémoire</li><li>- passage par adresse (pour les tableaux dynamiques)</li></ul> <p><b>Chap8.</b> Les structures complexes de données</p> <ul style="list-style-type: none"><li>- Section I - Le type « structure »</li><li>- Section II - Les listes chaînées</li><li>- Section III - Les Files, les piles et les arbres</li></ul> <p><b>Chap9.</b> Les algorithmes de tri</p> <ul style="list-style-type: none"><li>- Tri classique (par sélection), tri par insertion, tri à bulles (par propagation), tri par tas (heap sort).</li></ul> <p><b>Chap10.</b> Les fichiers (Type FILE)</p>

## Chapitre 6 : Fonctions et procédures

### 1- Déclaration des fonctions

A la suite de l'entête d'un programme C, on déclare les fonctions et les procédures en séquence.

Attention : les fonctions doivent être déclarées avant d'être appelées par d'autres fonctions.

Une fonction est généralement déclarée de la manière suivante :

```
<type> Nom_fonction(<type1> var1, <type2> var2, , <typen> varn)
{
    Déclaration des variables locales
    Instruction1;
    Instruction2;
    InstructionN;
    return resultat;
}
```

- **Nom\_fonction** : est le nom de la fonction, appelée aussi « identificateur »
- **<type>** : est le type de données retournée par la fonction (exemple int) : c'est le type de la variable appelée « resultat » retournée par la fonction.
- **var1, var2, ..., varn** : sont les arguments de la fonction ou encore ses variables / données d'entrée.
- **<type1>** : est le type de données de la variable **var1**.
- **<typen>** : est le type de données de la variable **varn**.

### 2- Quelques règles syntaxiques et d'utilisation

- Une fonction se termine généralement par l'instruction de la fonction « return ». Si on ajoute des instructions après cette dernière, elles ne seront pas prises en compte. L'instruction return est le point de sortie de la fonction.
- La valeur retournée par la fonction varie en fonction des valeurs prises par ses variables arguments.
- Les variables d'entrée (arguments) de la fonction sont indiquées entre parenthèses après le nom de la fonction. Elles sont précédées par leurs types respectifs et séparées entre elles par des virgules « , » .
- Le contenu de chaque fonction est délimité par deux accolades { }.
- Une fonction est composée de plusieurs instructions, Le nombre des instructions dans une fonction n'est pas limité,
- Chaque instruction est terminée par un point-virgule « ; » qui joue le rôle d'un séparateur qui marque la fin de l'instruction.
- Une instruction peut être écrite sur plusieurs lignes,
- Plusieurs instructions peuvent être écrites sur la même ligne à condition d'utiliser le séparateur (le point-virgule),
- Des commentaires peuvent être insérés dans le texte du programme à la suite du symbole « // » sur la même ligne ou entre les délimiteurs « /\* » et « \*/ » (pas nécessairement sur la même ligne). Ils n'affectent pas le code puisqu'ils sont inactifs.
- Une fonction peut être appelée un nombre illimité de fois par une autre fonction,

### 3- Notion de variables locales / globales

#### Locales :

- On peut aussi définir des variables à l'intérieur de la fonction.
- Dans ce cas, elles sont dites locales car elles ne seront utilisées que par les instructions de la fonction,
- La déclaration des variables devrait être terminée par un point-virgule,

Les variables déclarées dans la fonction main() sont aussi des variables locales à cette fonction (il s'agit souvent d'un sujet de confusion). La zone de déclaration des variables locales dans la fonction n'est pas forcément située avant le bloc des instructions (ce n'est qu'une question de présentation). Une variable peut être déclarée au milieu des instructions à condition qu'elle soit déclarée avant son utilisation.

Une variable peut être à la fois déclarée et initialisée.

```
#include <stdio.h>
int factoriel(int a)
{
    int k, fac=1; // déclaration variables locales
    for(k = 2; k<=a; k++)
        fac *= k;
    return fac;
}
```

- Une variable locale utilisée par une fonction, perd sa valeur après la fin de cette fonction
- Une variable locale utilisée par une fonction n'a aucune relation avec une autre variable locale qui porte le même nom dans une autre fonction. Chaque variable est localement utilisée par une fonction. On peut utiliser la même appellation pour nommer d'autres variables dans d'autres fonctions.

### Globales :

Lorsqu'une variable est déclarée dans le code même, c'est-à-dire à l'extérieur de toute fonction ou de tout bloc d'instruction, elle est accessible de partout dans le code (n'importe quelle fonction du programme peut faire appel à cette variable). On parle alors de **variable globale**.

```
#include <stdio.h>
int i = 3; // i est une variable globale
void main()
{
    int j=2, k = 12; // j et k sont deux variables locales
    k = i + j + k;
    printf("%d", k);
}
```

## 4- Les arguments d'une fonction

- Situées entre parenthèses après le nom de la fonction
- Ils représentent les variables d'entrée d'une fonction. Cette dernière admet une seule valeur de sortie qui est la valeur retournée.
- Au moment de la déclaration d'une fonction, les arguments sont délimités par des virgules précédées par leurs types. Ex: `int somme(int a, int b)`
- Au moment de l'appel d'une fonction, les arguments sont délimités par des virgules sans indications de types. Ex: `z=somme(x,y);`

### **Attention à la syntaxe :**

Quand on a besoin de déclarer deux variables du même type, on peut utiliser deux déclarations: `int i; int j;` ou bien une seule déclaration: `int i, j.` Aucune de ces deux possibilités n'est acceptable pour une liste de paramètres formels.

## 5- Appel d'une fonction

- Pour exécuter une fonction, il suffit de faire appel à elle, dans le corps d'un programme, en écrivant son nom suivi de parenthèses qui contiennent les valeurs des arguments. Ex: `y=2; z=somme(5,y+1);`

- une fonction peut être appelée plusieurs fois et par plusieurs fonctions qui sont définies à des positions qui viennent après sa position dans le programme.

## 6- Renvoi d'une valeur par une fonction

- la fonction peut renvoyer une valeur et donc se terminer grâce au mot clé `return`. La fonction peut contenir plusieurs instructions `return`. Elle se termine au premier `return` exécuté. Les instructions qui viennent après ce `return` ne seront pas lues par le processeur du programme. Ex :

```
int min(int a,int b)
{
```

```
        if(a<b) return a; else return b;
    }
```

- return permet d'interrompre même une boucle dans une fonction. Ex plus petit diviseur >1 d'un entier A:

```
int ppdiviseur(int a)
{
    int i;
    for(i=2;i<sqrt(a);i++)
        if(a%i==0) return i;
    return a;
}
```

## 7- Absence d'arguments dans une fonction

- Il est possible mais rare de ne pas avoir d'arguments pour une fonction. Ex :

```
double pi_()
{
    return 3.14159;
}
```

## 8- fonctions avec un argument du type « tableau »

Exercice : donner une fonction permet de calculer la somme des éléments d'un tableau d'entier.

**- Avec int A[]:**

```
#include <stdio.h>
int somme(int A[], int n)
{
    int i,resultat =0;
    for(i=0;i<n;i++)
    {
        resultat += A[i];
    }
    return resultat;
}
int somme2(int A[100], int n)
{
    int i,resultat =0;
    for(i=0;i<n;i++)
    {
        resultat += A[i];
    }
    return resultat;
}
int somme3(int A[5], int n)
{
    int i,resultat =0;
    for(i=0;i<n;i++)
    {
        resultat += A[i];
    }
    return resultat;
}
void main()
{
    int x[10],i;
    for(i=0;i<10;i++) x[i] = 1;
    printf("moy x : %d\n",somme(x,10));
    printf("moy x : %d\n",somme2(x,10));
    printf("moy x : %d\n",somme3(x,10));
}
```

**- Avec int \*A: (A est un tableau via un pointeur)**

```
#include <stdio.h>
int somme(int *A, int n)
{
    int i,resultat =0;
    for(i=0;i<n;i++)
    {
        resultat += A[i];
    }
    return resultat;
}

void main()
{
    int x[10],i;
    for(i=0;i<10;i++) x[i] = 1;
    printf("moy x : %d\n",somme(x,10));
}
```

## **9- Les procédures**

- Si la fonction ne retourne pas de valeur, il s'agit dans ce cas d'une procédure (un ensemble d'instructions à exécuter).
- Dans ce cas, le type de la donnée retournée par la procédure est noté « **void** ».
- La fonction main() est généralement prise comme procédure (elle n'est pas conçue pour retourner un seul résultat).

**Exemple :** Procédure affichage de la somme.

```
#include <stdio.h>

void affichage_sum(int x, int y) //procédure affichage
{
    printf("la somme de %d et %d = %d", x,y, x+y);
}
main()//fonction principale
{
    affichage_sum(2,5);
}
```

**Exemple :** affichage d'un élément d'un tableau

```
void imprime_tab (int *tab, int nb_elements)
{
    int i;
    for (i = 0; i < nb_elements; i++)
        printf("%d \t",tab[i]);
    printf("\n");
}
```

## **11- Passage d'arguments par valeur / adresse**

Les paramètres d'une fonction sont traités de la même manière que les variables locales : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, 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. On dit que les paramètres d'une fonction sont *transmis par valeurs*. Par exemple, le programme suivant :

**Exercice 1:** donner une fonction qui permet de calculer la longueur d'un entier.

```
#include <stdio.h>
int longueur (int a)
{
    int t=0;
    do
    {
        t++;
        a = a/10;
    }while(a !=0);
    return t ;
}
void main()
{
    int x; scanf ("%d",&x);
    printf ("%d\n",longueur(x));
}
```

Malgré les modifications réalisées à l'intérieur de la fonction de l'argument a, sa valeur n'a pas changé au niveau de la fonction main().

→ Le passage de l'argument a est fait par valeur. Seulement la valeur de a qui a été transférée à la fonction. Cette dernière a opéré sur une copie de la variable a.

Exercice : donner une procédure qui permet de permuter deux entiers. On a besoin de modifier les valeurs des arguments de la procédure. Le passage d'argument doit être réalisé par adresses. Dans ce cas la fonction opérera sur les variables elles mêmes.

Réponse 1 :

```
void echange (int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}

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

imprime
a = 2    b = 5
```

La modification n'est pas réalisée. Ici le passage d'arguments était fait par valeur.

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

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

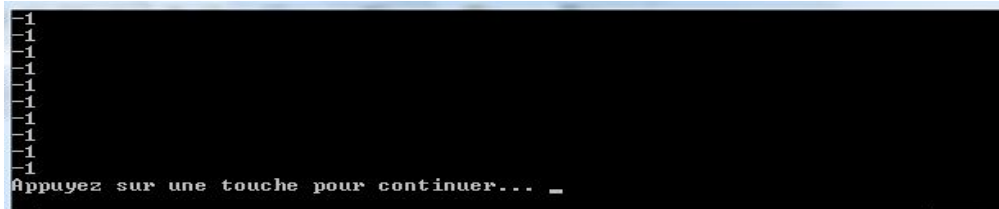
## **11- Passage d'argument par valeur / adresse : cas d'un tableau (pointeur)**

Exercice : donner une procédure qui permet de modifier à l'opposé les éléments d'un tableau d'entier. Avec l'utilisation de pointeur le passage d'arguments est automatiquement fait par adresse :

### **Modification réalisée :**

```
#include <stdio.h>
void modif_tab(int *A, int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        A[i] *=-1;
    }
}
void main()
{
    int x[10],i;
    for(i=0;i<10;i++) x[i] = 1;
    modif_tab(x,10);
    for(i=0;i<10;i++)
        printf("%d\n",x[i]);
}
```

la console résultante :



Ceci reste vrai si on utilise void modif\_tab(int A[], int n). En effet, int A[] et int \*A sont équivalentes.

## **13- Retourner un tableau par une fonction**

On ne retourne jamais un tableau « statique », du type int tab[3]; comme on le fait pour une variable simple. Pour faire équivalent, on peut retourner un pointeur sur une zone mémoire allouée dynamiquement pour contenir les éléments du tableau. Ceci implique une bonne connaissance sur le fonctionnement des pointeurs et sur l'allocation dynamique de la mémoire. Sinon, ce qui est mieux et simple, on peut passer le tableau, que nous voulons « retourner » comme résultat, en argument de la fonction. Sa modification sera prise en compte à la fin de la fonction (ou de la procédure).

Exemple :

```
void fonction (int *i)
{
    i[0] = 1;
    i[1] = 2;
    i[2] = 3;
}
```

## **12- Récursivité**

### **Exercice 1:** a<sup>b</sup>

```
int puissance (int a, int n)
{
    if (n == 0)
        return(1);
    return(a * puissance(a, n-1));
}

main()
{
    int a = 2, b = 5;
```

```
    printf("%d\n", puissance(a,b)); // a^b  
}
```

### **Exercice 2** : factoriel

```
int facto(int n)  
{  
    if (n == 1) return(1);  
    else return(n * facto(n-1));  
}
```

## **13- Exercice**

---

### **Exercice** : pgcd

```
int pgcd(int * a, int b) { // noter l'opérateur * indiquant le passage par adresse  
    int c;  
    if (*a<b) { // noter l'opérateur * a chaque occurrence de a  
        c=*a; *a=b; b=c;  
    }  
    do {  
        c=*a%b; *a=b; b=c;  
    } while (c!=0);  
    return *a;  
}  
int main(){  
    int x,y;  
    printf("donner deux entiers :");  
    scanf("%d %d",&x,&y);  
    printf("\n le pgcd de %d et %d est %d\n",x,y,pgcd(&x,y)); // noter l'operateur &  
    printf("\n le pgcd de 50 et 30 est %d\n",pgcd(50,30));  
}
```



## Chapitre 7 : Pointeurs et allocation dynamique de la mémoire

### Contenu

- arithmétique des pointeurs
- allocation dynamique de la mémoire
- passage d'arguments par adresse (pointeurs simples, pointeurs doubles)

### 1 - Définition d'un pointeur

Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné. Par ailleurs, la notion de pointeur est une technique de programmation permettant de définir des structures dynamiques, c'est-à-dire qui évolue au cours du temps (par opposition aux tableaux par exemple qui sont des structures de données statiques, dont la taille est figée à la définition).

### 2- Déclaration d'un pointeur

Un pointeur est une variable qui doit être définie en précisant le type de variable pointée, de la façon suivante :

```
type * Nom_du_pointeur ;
```

Le type de variable pointée peut être aussi bien un type primaire (tel que *int*, *char*...) qu'un type complexe (tel que *struct*...).

Un pointeur doit OBLIGATOIREMENT être typé !

Grâce au symbole '\*' le compilateur sait qu'il s'agit d'une variable de type *pointeur* et non d'une variable ordinaire, de plus, étant donné que vous précisez (obligatoirement) le type de variable, le compilateur saura combien de blocs suivent le bloc situé à l'adresse pointée.

### 3- La notion d'adresse

En réalité la mémoire est constituée de plein de petites cases de 8 bits (un octet). Une variable, selon son type (donc sa taille), va ainsi occuper une ou plusieurs de ces cases (une variable de type *char* occupera une seule case, tandis qu'une variable de type *long* occupera 4 cases consécutives).

Chacune de ces « cases » (appelées **blocs**) est identifiée par un numéro. Ce numéro s'appelle **adresse**.

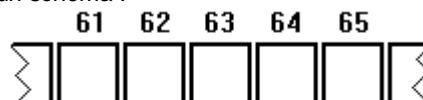
On peut donc accéder à une variable de 2 façons :

- grâce à son nom
- grâce à l'adresse du premier bloc alloué à la variable

#### Explication simplifiée :

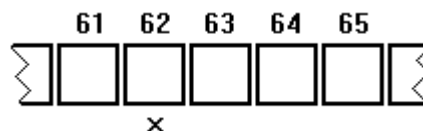
Nous allons représenter la mémoire de l'ordinateur par des cases numérotées en ordre croissant. On considérera qu'une variable utilise une case, même si dans la réalité elles ne prennent pas la même quantité de mémoire selon leur type.

Voyons-en une partie sous forme d'un schéma :



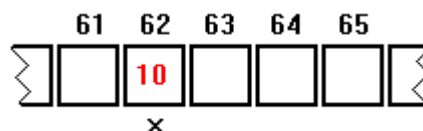
#### - Avec une variable ordinaire :

Quand j'écris : `int x;`



Je réserve une case pour la variable x dans la mémoire, case numéro 62 dans le cas du schéma.

Quand j'écris : `x=10;`



J'écris la valeur 10 dans l'emplacement réservé pour x. On dit que x a pour valeur 10. Cette valeur est située physiquement à l'emplacement &x (adresse de x) dans la mémoire (62 dans le contexte du schéma).

Pour obtenir l'adresse d'une variable on fait précéder son nom avec l'opérateur '&' (adresse de) :

```
printf("%p",&x);
```

Ce qui dans le cas du schéma ci-dessus, afficherait 62.

Remarque : pour afficher l'adresse : on utilise `printf("%p",&x);` ou encore `printf("%d",&x);` pour convertir l'adresse de son format hexadécimal (parfois composé) au format entier.

**- Avec les pointeurs:**

Un pointeur est aussi une variable, il est destiné à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire.

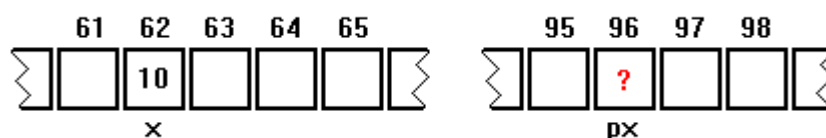
Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom du signe '\*' lors de sa déclaration.

Poursuivons notre exemple :

```
int *px; // Réserve un emplacement pour stocker une adresse mémoire.
px = &x; // Ecrit l'adresse de x dans cet emplacement.
```

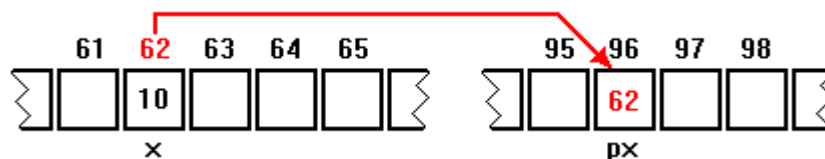
Décomposons :

Quand j'écris : `int *px;`



Je réserve un emplacement en mémoire pour le pointeur px (case numéro 96 dans le cas du schéma). Jusqu'a ici il n'y a donc pas de différence avec une variable ordinaire.

Quand j'écris : `px = &x;`



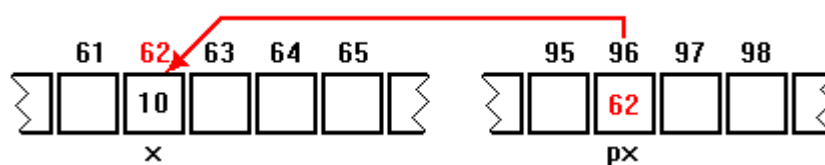
J'écris l'adresse de x à l'emplacement réservé pour le pointeur px. Je rappelle qu'un pointeur est destiné à mémoriser une adresse mémoire.

A l'emplacement réservé pour le pointeur px, nous avons maintenant l'adresse de x. Ce pointeur ayant comme valeur cette adresse, nous pouvons donc utiliser ce pointeur pour aller chercher (lire ou écrire) la valeur de x. Pour cela on fait précéder le nom du pointeur de l'opérateur de déréférencement '\*'.

Donc l'instruction suivante :

```
printf("%d",*px);
```

Affiche la valeur de x par pointeur déréférencé (10 dans le cas du schéma).



On peut donc de la même façon modifier la valeur de x :

```
*px = 20; // Maintenant x est égal à 20.
```

## 4- Arithmétique des pointeurs

Exemple 1 :

```
int a = 2;
char b;
int *p1;
char *p2;
p1 = &a;
p2 = &b;
*p1 = 10;
*p2 = 'a';
```

Si vous désirez utiliser cette notation dans une expression plus complexe, il sera nécessaire d'employer des parenthèses. Par exemple :

```
a = (*p1) + 2;
```

Exemple 2 :

```
int a = 2;
int *p1;
p1 = &a;
Si on écrit (*p1)++;
```

La valeur de a sera incrémentée

Exemple 3:

```
int a = 2;
int *p;
p = &a;
printf("valeur de p : %p - %d\n",p,p);
printf("adresse de a : %p - %d\n",&a,&a);
```

```
p : 0022FF44 - 2293572
adresse a : 0022FF44 - 2293572
Appuyez sur une touche pour continuer... _
```

Si on écrit par la suite

```
p++;
printf("\nvaleur de p : %p - %d\n",p,p);
printf("adresse de a : %p - %d\n",&a,&a);
```

```
p : 0022FF44 - 2293572
adresse a : 0022FF44 - 2293572
valeur de p : 0022FF48 - 2293576
adresse de a : 0022FF44 - 2293572
Appuyez sur une touche pour continuer... _
```

p pointera sur la première case mémoire disponible après la case mémoire de la variable a. Comme a est un entier la case de a contient 4 cellules (4 octets) l'adresse suivante à 2293572 est 2293576. Ici p ne pointera plus sur a. Cette dernière garde son adresse initiale.

Exemple 4:

```
int a =2, b=20;
int *p;
p = &a;
p = &b;
printf("adresse de a : %p - %d\n",&a,&a);
printf("adresse de b : %p - %d\n",&b,&b);
printf("valeur de p : %p - %d\n",p,p);
```

```
adresse de a : 0022FF44 - 2293572
adresse de b : 0022FF40 - 2293568
valeur de p : 0022FF40 - 2293568
Appuyez sur une touche pour continuer... _
```

## A retenir

Opérateur *	Opérateur &
<ul style="list-style-type: none"> <li>- L'opérateur « * » permet de savoir qu'il s'agit d'un pointeur lors d'une déclaration (ex : <code>int *x ;</code>).</li> <li>- L'opérateur « * » permet aussi d'accéder au contenu d'une case mémoire pointée par un pointeur (ex : <code>int *p,a; p=&amp;a; *p=2;</code>) : *p est le contenu de la case mémoire pointée par p.</li> <li>- L'opérateur « * » est utilisée pour rendre un argument d'une fonction modifiable (passage par</li> </ul>	<ul style="list-style-type: none"> <li>- L'opérateur permet d'accéder à l'adresse d'une variable ordinaire (ex : <code>int *p,a; p=&amp;a;</code>)</li> <li>- Il est utilisé lors de l'appel d'une procédure qui utilise des arguments pointeurs (ex : <code>void procedure(int *a, int *b)// déclaration</code> <code>procedure(&amp;a, &amp;b)// appel</code>) → passage d'argument par adresse</li> </ul>

adresse par l'utilisation d'arguments pointeurs → modification prise en compte au niveau du programme appelant).  
(ex : `void procedure (int *a, int *b)`)

Affichage d'une adresse (valeur d'un pointeur) :  
on utilise `printf("%p",&x);` ou encore `printf("%d",&x);` pour convertir l'adresse de son format hexadécimal (parfois composé) au format entier.

	Variable ordinaire <code>int x;</code>	Pointeur <code>int *ptr;</code>
Contenu	<code>x</code>	<code>*ptr</code>
Adresse	<code>&amp;x</code>	<code>ptr</code>

## 5- Passage d'argument par adresse (cas d'une variable simple en argument)

Exemple 1 :

<p>Corriger ce code :</p> <pre>#include&lt;stdio.h&gt; void Ajout2(int a) {     a +=2; } void main() {     int b = 3;     Ajout2(b); }</pre>	<p>Solution (passage par adresse) :</p> <pre>#include&lt;stdio.h&gt; void Ajout2(int *a) {     *a +=2; } void main() {     int b = 3;     Ajout2(&amp;b); }</pre>
--	---

Exemple 2:

```
void minmax(int i, int j, int *min, int *max)
{
    if(i<j) { *min=i; *max=j; }
    else { *min=j; *max=i; }
}
```

## 6- Allocation dynamique de la mémoire : Pointeurs et tableaux

Une utilisation très courante des pointeurs est l'allocation dynamique de la mémoire. Quand on fait une allocation dynamique de mémoire on obtient en retour un pointeur sur la zone mémoire allouée.

### malloc et free

Il s'agit de 2 appels système standards :

- La fonction malloc, avec la syntaxe (`int *p ; p = (int*) malloc(*sizeof(int));`) demande au système de fournir une zone mémoire de t cases et renvoie un pointeur vers cette zone (ou le pointeur NULL s'il n'y a pas assez de mémoire).
- La fonction free(p) libère la zone mémoire pointée par p.

## → Tableaux de taille variable

Grâce à malloc et free, on peut gérer des tableaux dont la taille est variable : un tableau peut s'allonger ou se réduire en fonction des besoins du programmeur. On appelle cela de l'allocation dynamique de mémoire. Ne pas oublier de libérer la mémoire.

Exemple 1 :

- tableau statique d'entiers de taille 10

```
int p[10];
p[0] = 5;
printf("%d",p[0]);
```
- tableau dynamique d'entiers de taille 10

```
int *p;
p = malloc(10*sizeof(int));
if(p) // si allocation reussie
{
    p[0] = 5;
    printf("%d",p[0]);
    free(p);
}
```

Exemple 2 :

```
#include<stdio.h>
#include<stdlib.h> // bibliothèque nécessaire pour l'allocation dynamique
void main()
{
    int *t;
    int i;
    t = (int *) malloc( 5 * sizeof(int) );
    if (t==NULL) printf("pas assez de mémoire\n");
    else
    {
        for(i=0 ; i<5 ; i++)
            t[i] = i * i;
        free(t);
    }
}
```

### Explications

- o Dans cet exemple, t est un pointeur vers un entier.
- o Après l'appel à la fonction malloc dans l'instruction t=(int \*)malloc(5\*sizeof(int)), t contient l'adresse d'une zone mémoire dont la taille est 5 fois la taille d'un entier. La variable t devient ainsi un tableau de 5 entiers qu'on peut utiliser comme n'importe quel tableau d'entiers.
- o Si t n'est pas NULL, ce qui signifie qu'il y avait assez de mémoire disponible, alors on peut accéder à n'importe quelle élément du tableau en écrivant t[i] (i étant bien sûr compris entre 0 et 4).
- o Dans ce programme, on remplit les 5 cases du tableau t en mettant i\*i dans la case i et on affiche ce tableau.
- o Ensuite, on libère l'espace occupé par le tableau en appelant la fonction free(t). t devient alors un pointeur non initialisé et on a plus le droit d'accéder aux différentes cases du tableau qui a été détruit.

Exemple 3 :

Création et lecture d'un tableau dynamique de taille 5

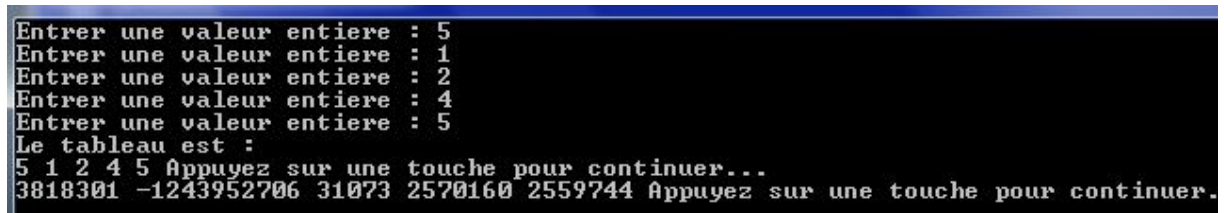
```
#include <stdio.h>
#include <stdlib.h>
void main ()
{
    int i=0, taille=5;
    int * tab;
    tab = (int*)malloc(taille*sizeof(int));
    do {
        printf ("Entrer une valeur entiere : ");
        scanf ("%d", &(tab[i]));
    }
```

```

        i++;
    } while (i<taille);
    printf ("Le tableau est : \n");
    for (i=0;i<taille;i++) printf ("%d ",tab[i]);
    system("PAUSE");
    for (i=taille;i<10;i++) printf ("%d ",tab[i]);
    free(tab);
    system("PAUSE");
}

```

On obtient :



```

Entrer une valeur entiere : 5
Entrer une valeur entiere : 1
Entrer une valeur entiere : 2
Entrer une valeur entiere : 4
Entrer une valeur entiere : 5
Le tableau est :
5 1 2 4 5 Appuyez sur une touche pour continuer...
3818301 -1243952706 31073 2570160 2559744 Appuyez sur une touche pour continuer.

```

Les valeurs qui sont en dehors de la taille du tableau peuvent être affichée sans déclaration d'erreur par le compilateur. Ce dernier lit les cases mémoires qui suivent celles du tableau. Ceci est aussi possible pour le cas d'utilisation d'un tableau statique.

## realloc

La fonction, avec la syntaxe (`int *p; p=(int*)realloc(p,nouvelleTaille*sizeof(int);`) change la taille de l'objet pointé par p (taille que nous appelons ancienneTaille) pour que cette taille devienne égale à nouvelleTaille. Si nouvelleTaille est plus petite que ancienneTaille, seul le début de l'objet pointé par p est conservé. La fonction retourne l'adresse du nouvel espace mémoire alloué ou bien NULL en cas d'échec.

```

/* realloc exemple */
#include <stdio.h>
#include <stdlib.h>
void main ()
{
    int i=0, taille;
    int * tab = NULL;
    do {
        tab = (int*)realloc(tab,(i+1)*sizeof(int));
        if (tab==NULL) { puts ("Error (re)allocating memory"); exit (1); }
        printf ("Entrer une valeur entiere : ");
        scanf ("%d", &(tab[i]));
        i++;
    } while (tab[i-1]!=0);
    taille = i;
    printf ("Le tableau est : \n");
    for (i=0;i<taille;i++) printf ("%d ",tab[i]);
    free(tab);
}

```

Le programme invite l'utilisateur à entrer des numéros dans un « tableau dynamique » jusqu'à l'entrée de la valeur 0 qui sera la dernière valeur du tableau. Chaque fois qu'une nouvelle valeur est introduite, le bloc de mémoire pointé (le tableau) est augmenté de la taille d'1 entier.

### Remarque :

Avant l'utilisation de realloc, il est indispensable d'initialiser le pointeur à la valeur NULL (ou bien il faut disposer d'une première allocation avec la fonction malloc par exemple). Avec malloc, on peut commencer avec un pointeur libre.

## 7- Passage d'argument par adresse (cas d'un tableau en arguments d'une procédure)

---

### 7.1- Cas d'un tableau statique :

Avec un pointeur simple, le passage d'argument est automatiquement fait par adresse

### 7.2- Cas d'un tableau dynamique (avec malloc) :

- CAS ou malloc est dans la fonction appelante → avec un pointeur simple, le passage d'argument est automatiquement fait par adresse

```
#include <stdio.h>
#include <stdlib.h>
void remplissage (int *tab, int taille)
{
    int i;
    printf ("Remplissage du tableau : \n");

    for(i=0;i<taille;i++)
    {
        printf ("Entrer valeur %d : ",i);
        scanf ("%d",&(tab[i]));
    }
}
void main ()
{
    int i, taille=5, *tab ;
    tab = (int*)malloc(taille*sizeof(int));
    if (tab==NULL)
        { puts ("Error (re)allocating memory"); exit (1); }
    remplissage(tab,taille);
    printf ("Le tableau est : \n");
    for (i=0;i<taille;i++)
    {
        printf ("%d\n",tab[i]);
    }
    free(tab);
}
```

- CAS ou malloc est dans la fonction appelée → **pointeur double**

```
#include <stdio.h>
#include <stdlib.h>
void remplissage (int **tab, int taille)
{
    int i;
    printf ("Remplissage du tableau : \n");
    *tab = (int*)malloc(taille*sizeof(int));
    if (*tab==NULL)
        { puts ("Error (re)allocating memory"); exit (1); }

    for(i=0;i<taille;i++)
    {
        printf ("Entrer valeur %d : ",i);
        scanf ("%d",&((*tab)[i]));
    }
}
void main ()
{
    int i, taille=5, *tab ;
    remplissage(&tab,taille);
    printf ("Le tableau est : \n");
    for (i=0;i<taille;i++)
    {
        printf ("%d\n",tab[i]);
    }
    free(tab);
}
```

### 7.3- Cas d'un tableau dynamique (avec realloc) :

On utilise un **pointeur double** (l'utilisation d'un pointeur simple ne marche pas !); On reprend l'exemple descriptif introduit pour la fonction realloc. On utilise une fonction de remplissage du tableau.

```
#include <stdio.h>
#include <stdlib.h>
void remplissage_dyn (int **tab, int *taille)
{
    int i=0;
    *tab = NULL;
    printf ("Remplissage du tableau : \n");
    do {
        *tab = (int*)realloc(*tab,(i+1)*sizeof(int));
        if (*tab==NULL)
            { puts ("Error (re)allocating memory"); exit (1); }
        printf ("Entrer valeur %d : ",i);
        scanf ("%d",&((*tab)[i]));
        i++;
    } while ((*tab)[i-1]!=0);
    *taille = i;
}
void main ()
{
    int i, taille, *tab ;
    remplissage_dyn(&tab,&taille);
    printf ("taille : %d\n",taille);
    if (tab==NULL)
        { puts ("Error (re)allocating memory"); getchar(); exit(1);}
    printf ("Le tableau est : \n");
    for (i=0;i<taille;i++)
    {
        printf ("%d\n",tab[i]);
    }
    free(tab);
}
```

## 8- Autre utilisation des pointeurs pour les tableaux

```
int Tab[10]={5,8,4,3,9};
```

L'instruction suivante affiche bien la valeur du premier élément du tableau par pointeur :

```
printf ("%d", *Tab);
```

Par ailleurs, `printf("%d",Tab[i]);` est équivalente à : `printf("%d",*(Tab+i));`

Ceci est vrai aussi après une allocation dynamique :

```
int *Tab, i;
Tab =(int*)malloc(5*sizeof(int));
*Tab = 5; *(Tab+1)=8; *(Tab+2)=4; *(Tab+3)=3; *(Tab+4)=9;
for(i=0 ;i<5 ;i++) printf ("%d\n",Tab[i]);
```

Tab est l'adresse du premier élément du tableau. Tab +i est l'adresse du i ème élément du tableau

C'est-à-dire `*(Tab+i)` et `Tab[i]` désignent la même chose.

## 9- Variable pointeur au lieu d'une variable ordinaire

```
int *x;
x =(int*)malloc(sizeof(int));
*x = 5; printf ("%d\n",*x);
free(x);
```

```
int x;
x = 5; printf ("%d\n",x);
```

→ Meilleure gestion de la mémoire.