1. Les bases

1.1 La structure d'un programme

Un programme simple se compose de plusieurs parties :
– des directives de précompilation
– une ou plusieurs fonctions dont l'une s'appelle obligatoirement main(), celle-ci constitue le programme principal et comporte 2 parties :
– la déclaration de toutes les variables et fonctions utilisées
– des instructions y compris les appels des autres fonctions
Les commentaires débutent par /* et finissent par */, ils peuvent s'étendre sur plusieurs lignes.

1.2 Les directives de précompilation

Elles commencent toutes par un #.

Commande	signification
#include <stdio.h>	*permet d'utiliser les fonctions* printf() *et* scanf()
#include <math.h>	*permet d'utiliser les fonctions mathématiques*
#define PI 3.14159	*définit la constante* PI
#undef PI	*à partir de cet endroit, la constante* PI *n'est plus définie*

```
#ifdef PI      // si la constante PI est définie, on compile les instructions1, sinon, les instructions2
    instructions 1 ...
#else
    instructions 2 ...
#endif
```

Parmi ces directives, une seule est obligatoire pour le bon fonctionnement d'un programme : #include <stdio.h>. En effet, sans elle, on ne peut pas utiliser les fonctions utiles pour l'affichage à l'écran : printf() et la lecture des données au clavier : scanf(). Nous verrons le fonctionnement de ces fonctions plus tard.

1.3 La fonction main()

Elle commence par une accolade ouvrante {et se termine par une accolade fermante }. À l'intérieur, chaque instruction se termine par un point-virgule. Toute variable doit être déclarée.

```
main(){
    int i ;          /* declaration des variables */
    instruction;
    ...
    }
```

Exemple de programme simple :

```
#include <stdio.h>
#include <stdlib.h>
/* Mon 1er programme en C */
main(){
    printf("Bonjour mes amis\n") ;
    system(''pause'') ; return 0 ;
    }
```

2. Variables et constantes

2.1 Les constantes

Constantes entières 1, 2, 3,... cons tint n=10 ;

Constantes caractères 'a','A',... const char cara='a' ;

Constantes chaînes de caractères "Bonjour" ; const char ch[20]="Bonjour" ;

Exemple d'utilisation :

```
#include <stdio.h>     #include <stdlib.h>
const int n=10;
const char cara='a';
const char ch[20]="bonjour";
main(){
     printf("%d   %c  %s \n",n, cara,ch);
     system("pause");       return 0;       }
```

```
10   a  bonjour
Appuyez sur une touche pour continuer...
```

Pas de constantes logiques Pour faire des tests, on utilise un entier. 0 est équivalent a faux et 1 à vrai.

2.2 Les variables

2.2.1 Noms des variables

Le C fait la différence entres les MAJUSCULES et les minuscules. Donc pour éviter les confusions, on écrit les noms des variables en minuscule et on réserve les majuscules pour les constantes symboliques définies par un #define. Les noms **doivent** commencer par une **lettre** et ne contenir **aucun blanc**. Le seul caractère spécial admis est le soulignement (_). Il existe un certain nombre de noms réservés (while, if, case, ...), dont on ne doit pas se servir pour nommer les variables. De plus, on ne doit pas utiliser les noms des fonctions pour des variables.

Déclaration des variables :

Pour déclarer une variable, on fait précéder son nom par son type. Il existe 6 types de variables :

Type	Signification	Valeur min	Valeur max
char	**caractère codé sur un octet**	**-2^7 (-128)**	**2^7 (+127)**
short	**entier court codé sur 2 octets**	**-2^{15} (-32768) à**	**$2^{15}-1$ (+32767)**
unsigned short	**entier court codé sur 2 octets**	**0 à**	**$2^{16}-1$ (+65535)**
long	**entier long codé sur 4 octets**	**-2^{31}**	**$+2^{32}-1$**
int	**entier codé sur 4 octets**	**-2^{31}**	**$+2^{31}-1$**
float	**réel codé sur 4 octets**	**-2^{31}**	**$+2^{31}-1$**
double	**réel double codé sur 8 octets**	**-2^{63}**	**$+2^{63}-1$**

On peut faire précéder chaque type par le préfixe unsigned, ce qui force les variables à prendre des valeurs uniquement positives.

Exemples de déclarations :

déclaration	
int a ;	*a est un entier*
int z=4 ;	*z est un entier =4*
unsigned int x ;	x *est un entier positif (non signé)*
float zx, zy ;	zx *et* zy *sont de type réel*
float zx=15.15 ;	zx *est de type réel et vaut 15.15*
double z ;	z *est un réel en double précision*
char zz ;	zz *est une variable caractère*
char yz='a' ;	yz *est une variable caractère et vaut* 'a'

Il n'y a pas de type complexe.

2.3 Les opérateurs

Le premier opérateur à connaître est l'**affectation "="**. Exemple : {a=b++" ;} Il sert à mettre dans la variable de **gauche** la valeur de ce qui est à droite. Le membre de droite est d'abord évalué, et ensuite, on affecte cette valeur à la variable de gauche. Ainsi l'instruction i=i+1 a un sens.

Pour les opérations dites naturelles, on utilise les opérateurs +, -, *, /, %. % est l'opération modulo : 5%2 est le reste de la division de 5 par 2. 5%2 est donc égal à 1.

Le résultat d'une opération entre types différents se fait dans le type le plus haut. Les types sont classés ainsi :

char < short < int < float < double

Par ailleurs, l'opération 'a'+1 a un sens, elle a pour résultat le caractère suivant à a dans le code ASCII.

En C, il existe un certain nombre d'opérateurs spécifiques, qu'il faut utiliser prudemment sous peine d'erreurs.

++ incrémente la variable d'une unité.

- - décrémente la variable d'une unité.

Ces 2 opérateurs ne s'utilisent pas avec des réels. Exemples d'utilisation :

```
i++ ; /* effectue i=i+1 */
i- - ; /* effectue i=i-1 */
```

Leur utilisation devient délicate quand on les utilise avec d'autres opérateurs. Exemple :

```
int  i=1 , j ;     /* effectue d'abord j=i et ensuite i=i+1 */
j=i++ ;            /* on a alors j=1 et i=2 */
j=++i ;            /* effectue d'abord i=i+1 et ensuite j=i */
                   /* on a alors j=3 et i=3 */
```

Quand l'opérateur ++ est placé avant une variable, l'incrémentation est effectuée en premier. L'incrémentation est faite en dernier quand ++ est placé aprés la variable. Le comportement est similaire pour - -.

D'autres opérateurs sont définis dans le tableau qui suit :

```
i+=5 ; /* i=i+5 */
i-=3 ; /* i=i-3 */
i*=4 ; /* i=i*4 */
i/=2 ; /* i=i/2 */
i%=3 ; /* i=i%3 */
```

Pour finir, ajoutons que les opérateurs qui servent à comparer 2 variables sont :

==	égal à
<	inférieur
>	supérieur
!=	différent de
<=	inférieur ou égal
>=	supérieur ou égal
&&	'et' logique
||	'ou' logique

ATTENTION !

Ne pas confondre l'opérateur d'affectation = et l'opérateur de comparaison ==.

3. Quelques fonctions indispensables

- ✓ **La fonction** printf()

Elle sert à afficher à l'écran la chaîne de caractère donnée en argument, c'est-à-dire entre parenthèses.

printf("Bonjour\n") ; affichera Bonjour à l'écran.

Certains caractères ont un comportement spécial :

\n	retour à la ligne	\"	"
\b	n'imprime pas la lettre précédente	\'	'
\r	n'imprime pas tout ce qui est avant	\ ?	?
\t	tabulation horizontale	\ !	!
\v	tabulation verticale	\\	\

Mais printf() permet surtout d'afficher à l'écran la valeur d'une variable :

```
main(){
    int n=3, m=4 ;
```

```
        printf("%d",n) ;                /* affiche la valeur de n au format d (decimal) */
        printf("n=%d",n) ; /            * affiche 'n=3' */
        printf("n=%d, m=%d",n,m) ; /* affiche 'n=3, m=4' */
}       printf("n=%5d",n) ; /* affiche la valeur de n sur 5 caracteres : 'n=          3' */
```

Le caractère % indique le format d'écriture à l'écran. Dès qu'un format est rencontré dans la chaîne de caractère entre " ", le programme affiche la valeur de l'argument correspondant.

printf("n=%d, m=%d , k= %f " ,n, m, 25.5);

ATTENTION ! le compilateur n'empêche pas d'écrire un char sous le format d'un réel ⇒ affichage de valeurs délirantes. Et si on écrit un char avec un format décimal, on affiche la valeur du code ASCII du caractère.

Tableau des formats utilisables

%d	int	*entier (décimal)*
%u	unsigned	*entier non signé (positif)*
%hd	short	*entier court*
%ld	long	*entier long*
%f	float	*réel, notation avec le point décimal (ex. 123.15)*
%e	float	*réel, notation exponentielle (ex. 0.12315e+03)*
%lf	double	*réel en double précision, notation avec le point décimal*
%le	double	*réel en double précision, notation exponentielle*
%c	char	*caractère*
%s	char	*chaîne de caractères*

Remarque :une chaîne de caractères est un tableau de caractères. Elle se déclare de la façon suivante : char p[10] ;. Mais nous reviendrons sur la notion de tableau plus tard.

- ✓ **La fonction** scanf()

Dans un programme, on peut vouloir qu'une variable n'ait pas la même valeur à chaque exécution.
La fonction scanf() est faite pour cela. Elle permet de lire la valeur que l'utilisateur rentre au clavier et de la stocker dans la variable donnée en argument. Elle s'utilise ainsi :

```
main(){
    int a ;
    scanf("%d",&a) ;   // se trouve dans la librerie stdio.h
}
```

On retrouve les formats de lecture précisés entre " " utilisés pour printf(). Pour éviter tout risque d'erreur, on lit et on écrit une même variable avec le même format.

Le **&** est indispensable pour le bon fonctionnement de la fonction. Il indique l'adresse de la variable, mais nous reviendrons sur cette notion d'adresse quand nous aborderons les pointeurs.

- ✓ **La librairie** string.h

La librairie string.h fournit un certain nombres de fonctions très utiles pour manipuler des chaînes de caractères en C. En effet, le C ne sait faire que des affectations et des comparaisons pour 1 seul caractère.

```
char p, q ;                          /* p et q sont des caractères */
char chaine1[10], chaine2[10] ;      /* chaine1 et chaine2 sont des chaînes de caractères */
 p = 'A' ;                           /* instruction valide */
chaine1 = "Bonjour" ;                /* instruction NON valide (1) */
chaine2 = "Hello" ;                  /* instruction NON valide (2) */
if (p == q) ;                        /* instruction valide */
if (chaine1 == chaine2)              /* instruction NON valide (3) */
```

Pour faire les affectations (1) et (2), et la comparaison (3), il faudrait donc procéder caractère par caractère. Ces opérations étant longues et sans intérêt, on utilise les fonctions déjà faites dans string.h. Pour les opérations d'affectation (1) et (2), il faut utiliser la fonction strcpy (string copy), et pour une comparaison (3), la fonction strcmp (string compare).

```
#include<stdio.h>
#include<string.h>
main(){
    char chaine1[10], chaine2[10] ;
    int a ;
    strcpy(chaine1,"Bonjour") ; /* met "Bonjour" dans chaine1 */
    strcpy(chaine2,"Hello") ;          /* met "Hello" dans chaine2 */
    a=strcmp(chaine1,chaine2) ;
    /* a reçoit la différence chaine1 et chaine2 */
    /* si chaine1 est classé alphabétiquement avant chaine2, alors a<0 */
    /* si chaine1 est classé après chaine2, alors a>0 */
    /* si chaine1 = chaine2, alors a = 0 */
    /* Ici, "Bonjour" est alphabétiquement avant "Hello", */
    /* donc chaine1 est plus petite que chaine2, et a < 0 */
}
```

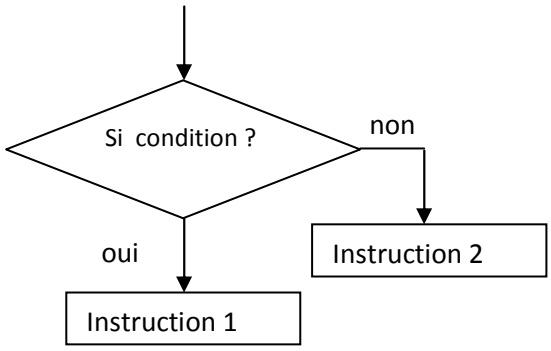
4. Les instructions de contrôle

Ce sont des instructions qui permettent notamment de faire des tests et des boucles dans un programme. Leur rôle est donc essentiel. Pour chaque type d'instruction de contrôle, on trouvera à la fin de la partie 4 les organigrammes correspondant aux exemples donnés.

4.1 Les instructions conditionnelles

✓ **Les tests** if

L'opérateur de test s'utilise comme suit :




```
if (condition) {instruction ;}
/* Si condition est vraie alors instruction est exécutée */
if (condition) {
    instruction 1 ;
    } else {
    instruction 2 ;}
}
/* Si expression est vraie alors l'instruction 1 est exécutée */
/* Sinon, c'est l'instruction 2 qui est exécutée */
if (condition 1) {
    instruction 1 ;
} else if (condition 2){
    instruction 2 ;
} else if (condition3){
    instruction 3 ;
} else { instruction 4 ;          }
/* Souvent, on imbrique les tests les uns dans les autres */
```

Remarque : les instructions à exécuter peuvent être des instructions simples {a=b ;} ou un bloc d'instructions {a=b ; c=d ; m=pow(4,2),...}.

Comme nous l'avons déjà vu, une expression est vraie si la valeur qu'elle renvoie est non nulle.
ATTENTION ! les expressions (a=b) et (a==b) sont différentes :

```
if (a==b)      vrai si a et b sont égaux
int b=1 ;
if (a=b)       on met la valeur de b dans a. a vaut alors 1. l'expression est donc vraie
```

✓ **Les tables de branchement :** switch

Cette instruction s'utilise quand un **entier** ou un caractère prend un nombre fini de valeurs et que chaque valeur implique une instruction différente.

```
switch(i) {
        case valeur1 : instruction 1 ;      /* si i=1 on exécute l'instruction 1 */
                break ;              /* et on sort du switch */
        case valeur2 : instruction 2 ;      /* si i=2 ... */
                break ;
        case valeur10 : instruction 3 ;     /* si i=10 ... */
                break ;
        default : instruction 4 ;     /* pour les autres valeurs de i */
                break ;
    }
```

ATTENTION ! On peut ne pas mettre les break ; dans les blocs d'instructions. Mais alors on ne sort pas du switch, et on exécute toutes les instructions des case suivants, jusqu'à rencontrer un break ;.

Si on reprend l'exemple précédent en enlevant tous les break, alors :

? si i=1 on exécute les instructions 1, 2, 3 et 4
? si i=2 on exécute les instructions 2, 3 et 4
? si i=10 on exécute les instructions 3 et 4
? pour toutes les autres valeurs de i, on n'exécute que l'instruction 4.

4.2 Les boucles

✓ **La boucle** for

Elle permet d'exécuter des instructions plusieurs fois sans avoir à écrire toutes les itérations. Dans ce genre de boucle, on doit savoir le nombre d'itérations avant d'entrer dans la boucle. Elle s'utilise ainsi :

```
for (compteur=valeur_initiale ; condition_d'arret ; incrémentation){
                ❶                      ❷              ❸
    Bloc instructions ... ;
                                    }
```

```
for ( i=0 ;  i<20 ;   i++) {
        ❶     ❷      ❸
    printf(‘’ i=%d \n ‘’,i);
                    }
```

Dans cette boucle, i est le **compteur**. Il ne doit pas être modifié dans les instuctions, sous peine de sortie de boucle et donc d'erreur.

- La 1ere ` instruction entre parenthèses est l'initialisation de la boucle (*compteur=valeur_initiale*).
- La 2eme est la condition de sortie de la boucle : tant qu'elle est vraie, on continue la boucle. ? Et la 3eme ` est l'instruction d'itération : sans elle, le compteur reste à la valeur initiale et on ne sort jamais de la boucle.

✓ **La boucle** while

Contrairement à la boucke for, on n'est pas obligés ici de connaître le nombre d'itérations. Il n'y a pas de compteur.

```
compteur=valeur_initiale
        ❶
while (condition_d'arret ) {
            ❷
    bloc  instructions ... ;
    incrémentation  du compteur      }
        ❸
```

```
❶  i=0  ;
        ❷
while  (i<20)  {
    printf(‘’ i= %d \n’’,i) ;
    i++ ;
        ❸
}
```

L'expression est évaluée à chaque itération. Tant qu'elle est vraie, les instructions sont exécutées.
Si dès le début elle est fausse, les instructions ne seront jamais exécutées.
ATTENTION ! Si rien ne vient de modifier l'expression dans les instructions, on a alors fait une boucle infinie :
while (1) { instructions }. ⟹ Boucle infinie

Exemple d'utilisation de la boucle while:

```
#include <stdio.h>
main(){
    float x=1.0, R=1.e5 ;
    while (x < R) {
        x = x+0.1*x ;
        printf("x=%f",x) ;
            }
        }
```

- ✓ **La boucle *do ... while***

À la différence d'une boucle while, les instructions sont exécutées au moins une fois : l'expression est évaluée en fin d'itération.

```
compteur=valeur_initiale
            ❶
    do {
        instructions ... ;
        incrémentation du compteur ;
                    ❸
    } while (condition_d'arret) ;
                ❷
```

Les risques des faire une boucle infinie sont les mêmes que pour une boucle while.

- ✓ **Les instructions** break , continue **et goto.**

break fait sortir de la boucle.

continue fait passer la boucle à l'itération suivante.

Goto fait sauter à une adresse (référence dans le programme)

5. Les fonctions

Créer une fonction est utile quand vous avez à faire le même type de calcul plusieurs fois dans le programme, mais avec des valeurs différentes. Typiquement, pour le calcul de l'intégrale d'une fonction mathématique *f* .

Comme en mathématique, une fonction prend un ou plusieurs arguments entre parenthèses et renvoie une valeur.

5.1 Déclaration

On déclare une fonction ainsi :

```
Type   nom_de_la_fonction( type1 arg1 , type2  arg2, ... , typen  argn) { */ prototype */
        déclaration variables locales ;
        instructions ;
        return (expression) ;
    }
```

Type est le type de la valeur renvoyée par la fonction. type1 est le type du 1erargument arg1 ... . Les variables locales sont des variables qui ne sont connues qu'à l'intérieur de la fonction.
expression est évaluée lors de l'instruction return (expression) ; c'est la valeur que renvoie la fonction quand elle est appelée depuis main().
La 1^{ere} ligne de la déclaration est appelée le **prototype** de la fonction.

Exemple de fonction :

```
float affine( float x ) {      /* la fonction 'affine' prend 1 argument réel x et renvoie un argument réel (ax+b)*/
```

```
    int a,b ;
    a=3 ;
    b=5 ;
    return (a*x+b) ;       /* valeur renvoyée par la fonction */
}
```

On n'est pas obligés de déclarer des variables locales :

```
float norme( int x, int y ){ /* la fonction 'norme' prend 2 arguments entiers et renvoie un résultat réel */
    return (sqrt(x*x+y*y)) ;          /* sqrt est la fonction racine carree */
    }
```

On peut mettre plusieurs instructions return dans la fonction :

```
    float val_absolue( float x ) {
    if (x < 0) {
    return (-x) ;
    } else {
    return (x) ;
    }
    }
```

Une fonction peut ne pas prendre d'argument, dans ce cas-là, on met à la place de la déclaration des arguments, le mot-clé void :

```
    double pi ( void ) { /* pas d'arguments */
    return(3.14159) ;
    }
```

Une fonction peut aussi ne pas renvoyer de valeur, son type sera alors void :

```
  void mess_err ( void ) {
        printf("Vous n'avez fait aucune erreur \n") ;
        return ; /* pas d'expression après le return */
    }
```

- ✓ **Appel de la fonction**

Une fonction f() peut être appelée depuis le programme principal main() ou bien depuis une autre fonction g() à la condition de rappeler le prototype de f() après la déclaration des variables de main() ou g() :

```
  #include <stdio.h>
   main(){
        int   x ,y ,r ;
        int   plus( int x, int y ) ;    /* déclaration du prototype de la fonction plus(x,y)*/
        x=5 ;
        y=235 ;
        r=plus(x,y) ; /* appel d'une fonction avec arguments */
    }
        int plus( int x, int y ){      /* définition de la fonction plus(x,y)*/
        void mess( void ) ;
        mess_err() ; /* appel d'une fonction sans arguments */
        return (x+y)) ;
    }
    void mess( void ) {
        printf ("Vous n'avez fait aucune erreur\n") ;
        return ; }
```

Quand le programme rencontre l'instruction return, l'appel de la fonction est terminé. Toute instruction située après lui sera ignorée.

Exercices :

1. *On désire calculer l'exponentielle de x en utilisant la série :* $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$,

Ecrire une fonction serie_expon(float x, int n) qui développe la série jusqu'à un indice n; elle reçoit en paramètre les valeurs de x et de n.

Ecrire une fonction expon(float x,int n) qui retourne la valeur de l'exponentielle d'un nombre x passé en paramètre en développant jusqu'à l'indice n.

Ecrire un programme principal qui saisit un nombre réel et un entier n et affiche son exponentielle à l'ordre n.

2. *Ecrire la fonction NombreChiffre du type int qui obtient une valeur entière N (positive ou négative) du type long comme paramètre et qui fournit le nombre de chiffres de N comme résultat.*

Ecrire un petit programme qui teste la fonction NombreChiffre:
Exemple d'exécution:
Introduire un nombre: 123
Le nombre 123 a 3 chiffres.
Introduire un nombre: 580499
Le nombre 580499 a 6 chiffres.

6. Les tableaux

6.1 Déclaration

Comme une variable, on déclare son type, puis son nom. On rajoute le nombre d'éléments du tableau entre crochets [ ] :

Syntaxe float tab[5] ; est un tableau de 5 flottants.
int tablo[3] ; est un tableau de 3 entiers.

float tab[5];

	tab
tab[0]=	2.2
tab[1]=	4.5
tab[2]=	11
tab[3]=	7.9
tab[4]=	25.7

float tablo[5];

	tablo
tablo[0]=	22
tablo[1]=	45
tablo[2]=	11

ATTENTION !

Les numéros des éléments d'un tableau de *n* éléments vont de 0 à $n - 1$.

La taille du tableau doit être une constante (par opposition à variable), donc int t1[n] ; où n serait une variable déjà déclarée est une mauvaise déclaration. Par contre si on a défini #define N 100 en directive, on peut déclarer int t1[N] ; car N est alors une constante.

L'utilisation des accolades {}

```
int tableau[4] = {15,2,14,23};
```

L'utilisation des accolades {} avec un tableau dont la taille n'est pas spécifiée :

```
int tableau[] = {15,2,14,23};
int t[10] = {[indice]=5,6}; exemple : int t[10] = {[2]=5,6};
int t[10] = {2,3,[4]=5,6,9}; résultat : 2  3  0  0  5  6  9  0  0  0
int t[10] = {[4]=5,6,[9]=9}; résultat :
```

0	0	0	0	5	6	0	0	0	9

6.2 Les tableaux de tableaux ("tableaux à plusieurs dimensions")

```
int tableau1[2][3] = {{1,8,9},{0,6,4}};
int tableau1[2][3][2] = {{ {1,8}  , {0,6} , {0,0} },
                         { {31,52} , {4,8} , {11,5}}
                        };
int tableau1[][3] = {{1,8,9},{0,6,4},{5,3,7},{2,2,2}};
int t[4][5] = {{1,[3]=5,6},   [2]={[3]1},{2,4,[4]=10}};
int tableau[taille1][taille2][taille3];
```

```
int i,j,k;

for(i=0 ; i < taille1 ; i++){            //Première dimension
    for(j=0 ; j < taille2 ; j++){        //Deuxième dimension
        for(k=0 ; k < taille3 ; k++){  //troisième dimension
            tableau[i][j][k] = 0;
        }
    }
}
```

✓ **Cas d'un tableau de caractères**

Un tableau de caractères est en fait une **chaîne de caractères**. Son initialisation peut se faire de plusieurs façons :

```
char p1[10]='B','o','n','j','o','u','r' ;
char p2[10]="Bonjour" ; /* initialisé par une chaîne littérale */
char p3[ ]="Bonjour" ; /* p3 aura alors 8 éléments */
```

ATTENTION ! Le compilateur rajoute toujours un caractère *'\0'* à la fin d'une chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus que la chaîne littérale.

7. Les pointeurs

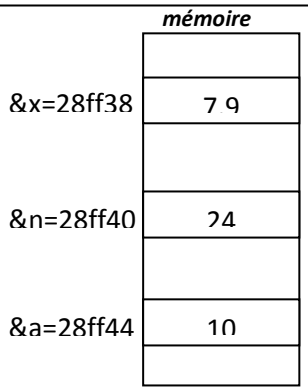
7.1 Stockage des variables en mémoire

Lors de la compilation d'un programme, l'ordinateur réserve dans sa mémoire une place pour chaque variable déclarée. C'est à cet endroit que la valeur de la variable est stockée. Il associe alors au nom de la variable l'adresse de stockage. Ainsi, pendant le déroulement du programme, quand il rencontre un nom de variable, il va chercher à l'adresse correspondante la valeur en mémoire.
Pour les déclarations de variables suivantes on aura les résultats suivants :

```
main(){
    int a=10;
    long n=24;
    float x=77.9;
    printf(" valuer a=%d  adresse de a &a=%x  \n",a,&a);
    printf(" valeur de n=%ld  adresse de n &n=%x  \n",n,&n);
    printf(" valeur de x=%.2f  adresse de   &x=%x  \n",x,&x);
    system("pause");
    return 0;     }
```

```
valeur de a=10 adresse de a &a=28ff44
valeur de n=24 adresse de n &n=28ff40
valeur de x=77.90 adresse de x &x=28ff3c

Appuyez sur une touche pour continuer...
```




```
main(){
    int a=10;      long n=24;      float x=77.9;
    printf("\n\n");
    printf(" valeur de a=%d adresse de a &a=%x  \n",a,&a);
    printf(" (adresse de a)+1: &a+1=%x  \n",&a+1);      printf("\n");
    printf(" valeur de n=%ld adresse de n &n=%x  \n",n,&n);
    printf(" (adresse de n)+1 &n+1=%x  \n",&n+1);      printf("\n");
    printf(" valeur de x=%.2f adresse de x &x=%x  \n",x,&x);
    printf(" (adresse de x)+1 &n+1=%x  \n",&n+1);      printf("\n");
    //printf("%d  %c  %s \n",i,cara,ch);
    system("pause");      return 0;      }
```

```
valeur de a=10 adresse de a &a=28ff44
(adresse de a)+1: &a+1=28ff48

valeur de n=24 adresse de n &n=28ff40
(adresse de n)+1 &n+1=28ff44

valeur de x=77.90 adresse de x &x=28ff3c
(adresse de x)+1 &x+1=28ff40

Appuyez sur une touche pour continuer...
```

Explication du schéma et résultat obtenu:

- Chaque case supporte une donnée codée sur 8 bits (1 octet)
- x est codé sur 4 octets et son adresse est (28ff3c), donc l'adresse de n sera (28ff3c)+(4)=(28ff40).
- n est codé sur 4 octets et son adresse est (28ff40), donc l'adresse de a sera (28ff40)+(4)=(28ff44).

7.2 Définition et déclaration d'un pointeur

Un pointeur est une variable qui a pour valeur l'adresse d'une autre variable : celle sur laquelle elle pointe ! Un pointeur est toujours associé à un type de variable et un seul. Au moment de la déclaration, on détermine le type de variable pointé par le pointeur, en écrivant le type concerné, puis le nom du pointeur avec une * devant :

```
int *pta ; /* la variable pta est un pointeur sur un entier*/
int a ; /* la variable a est un entier*/
```

l'écriture ptr=&a signifie que le pointeur ptr pointe vers l'adresse de a.

7.3 Opérateur d'adresse : &

Pour affecter l'adresse de la variable a au pointeur pta, on écrit l'instruction : pta=&a ;
Cet opérateur signifie donc **adresse de**.

7.4 Opérateur d'indirection : *

Cet opérateur, mis en préfixe d'un nom de pointeur signifie **valeur de la variable pointée** ou, plus simplement, valeur pointée.

```
int a=10 ;
int *ptint ;          /* declaration d'un pointeur sur un entier */
ptint=&a ;            /* ptint pointe sur a */
*ptint=12 ;           /* la variable pointée par ptint reçoit 12*/
printf("a=%d \n",a) ; /* affiche "a=12" */
```

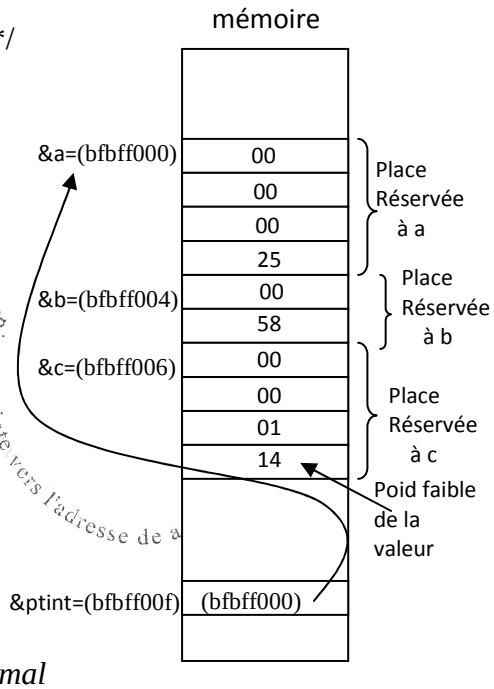
En fait, manipuler ptint revient à manipuler a.

7.5 Mémoire et pointeurs

On reprend l'exemple de la partie 7.1 en ajoutant un pointeur d'entier :

```
int a=25 ;
short b=58 ;
int c=114 ;
int *ptint ;
ptint=&a ;
```

Etat de la mémoire :



nom	*adresse*	*valeur en décimal*
a	bfbff000	25
b	bfbff004	58
c	bfbff006	114
ptint	bfbff00f	bfbff000

Exercices :

1. Ecrire un programme en C qui permet d'éliminer un caractère dans une chaine de caractères.
2. Saisir 10 réels, les ranger dans un tableau. Calculer et afficher la moyenne et l'Écart-type.
3. Saisir une matrice d'entiers 2x2, calculer et afficher son déterminant.

7.6 Tableaux et pointeurs :

La déclaration d'un pointeur sur tableau s'effectue en utilisant des parenthèses, l'astérisque '*' et en définissant la taille du tableau sur lequel on souhaite pointer. Notez que les parenthèses sont très importantes car en leur absence, ce sera un tableau de pointeurs qu'on aura déclaré, ce qui n'est pas la même chose.

La déclaration d'un tableau de pointeurs se fait comme pour un tableau de variables quelconques :
le type puis le nom du tableau avec :

- le nombre d'éléments entre crochets derrière le nom et une * devant.

int *pttab[4] ; / * pttab est un tableau de 4 pointeurs d'entiers */

Exemple :

```
int (*ptrtableau)[4];
```

Dans cet exemple, il s'agit d'une déclaration d'un pointeur sur tableaux de 4 entiers.

L'erreur à ne pas faire :

```
int * ptrtableau[4];
int (*tableau)[4];
tableau = malloc(5 * sizeof(*tableau));
```

Ainsi on aura déclaré un tableau de 5 tableaux de 4 entiers chacun (équivalent à `int tableau[5][4];`).

```
int * tableauDePtr[5];
int i;

for(i=0 ; i < 5 ; i++){
        tableauDePtr[i] = malloc(4 * sizeof(tableau[0]));}
```

Ainsi on aura créé un tableau de 5 tableaux de 4 entiers chacun (équivalent à `int tableau[5][4];`).

✓ *Pointeurs et tableaux à plusieurs dimensions*

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

Ainsi, le tableau défini par int tab[4][5] ; contient 4 tableaux de 5 entiers chacuns. tab donne l'adresse du 1er sous-tableau, tab+1 celle du 2ème sous-tableau et ainsi de suite :

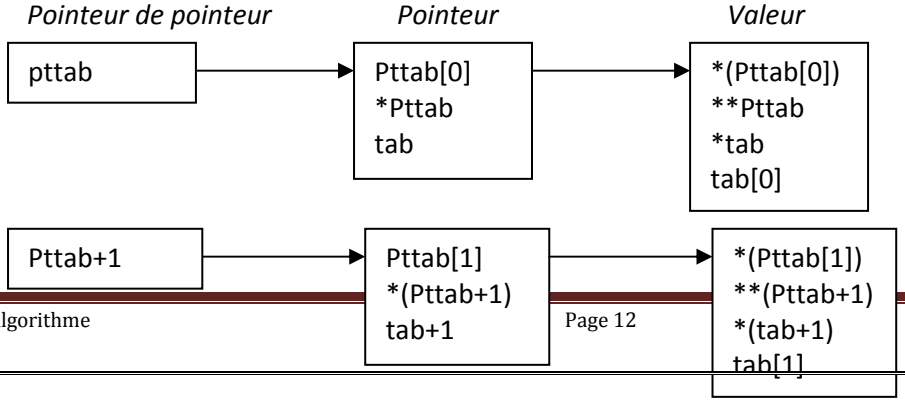
tab ⟶	tab+1 ⟶	tab+2 ⟶	tab+3 ⟶
tab[0][0]	tab[1][0]	tab[2][0]	tab[3][0]
tab[0][1]	tab[1][1]	tab[2][1]	tab[3][1]
tab[0][2]	tab[1][2]	tab[2][2]	tab[3][2]
tab[0][3]	tab[1][3]	tab[2][3]	tab[3][3]
tab[0][4]	tab[1][4]	tab[2][4]	tab[3][4]

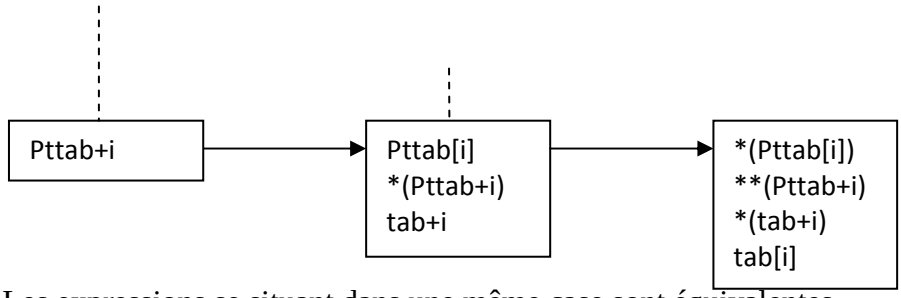
Ici, l'opération tab+2 n'ajoute pas 2 à la valeur de tab mais ajoute 2 fois le nombre d'octets correspondant à un tableau de 5 entiers, à savoir 5*4 = 20 octets.

1er exemple d'utilisation d'un tableau de pointeurs

int tab[4] ; /* tab est un tableau de 4 entiers */
int *pttab[4] = { tab , tab+1 , tab+2 , tab+3 } ;

Les valeurs du tableau pttab sont des adresses de données. On dit alors que pttab est un **pointeur de pointeurs** car (pttab) pointe sur l'adresse de son 1er élément, qui est lui-même une adresse. Voici les relations qu'il y a entre pttab et tab :





Note : Les expressions se situant dans une même case sont équivalentes.

Au niveau de la mémoire, les éléments d'une colonne contiennent les adresses des éléments de la colonne qui est juste à sa droite.Voici l'état de la mémoire pour un tel exemple :

nom	***Adresse***	***valeur***
tab	(bfbff000)	00 00 00 25
	(bfbff004)	00 00 00 47
	(bfbff008)	00 00 01 75
	(bfbff00c)	00 00 21 74
pttab	(bfbff010)	bf bf f0 00
	(bfbff014)	bf bf f0 04
	(bfbff018)	bf bf f0 08
	(bfbff01c)	bf bf f0 0c

Exemple2

```
int l1[4] = { 11 , 22 , 33 , 44 } ;
int l2[3] = { 55 , 66 , 77 } ;
int l3[1] = { 88 } ;
int *pt[3] = { l1 , l2 , l3 } ;
```

Les éléments de tab sont les adresses de tableaux ne comportant pas le même nombre d'éléments.
pt est en fait un tableau dont les 3 lignes n'ont pas la même longueur.

Pointeur de pointeur	***pointeur***	***Equivalence***
pt	pt[0]≡l1	l1[0] ↔ pt[0][0]
		l1[1] ↔ pt[0][1]
		l1[2] ↔ pt[0][2]
		l1[3] ↔ pt[0][3]
pt+1	pt[1]≡l2	l2[0] ↔ pt[1][0]
		l2[1] ↔ pt[1][1]
		l2[2] ↔ pt[1][2]
pt+2	pt[2]≡l3	l3[0] ↔ pt[2][0]

Voici l'état de la mémoire pour cet exemple :

nom	***Adresse***	***valeur***
L1	(bfbff000)	00 00 00 11
	(bfbff004)	00 00 00 22
	(bfbff008)	00 00 00 33
	(bfbff00c)	00 00 00 44
L2	(bfbff010)	00 00 00 55
	(bfbff014)	00 00 00 66
	(bfbff018)	00 00 00 77
L3	(bfbff01c)	00 00 00 88
pt	(bfbff020)	(bf bf f0 00)
	(bfbff024)	(bf bf f0 10)
	(bfbff028)	(bf bf f0 1c)

Explication : Les cases mémoire des variables a, b et c contiennent leur valeur. Celles de la variable ***ptint*** contiennent l'adresse de la valeur pointée. En effet, la valeur stockée est (bfbff000), ce qui est bien l'adresse de a.

8.6 Exemple

Voici un petit exemple d'illustration :

```
#include <stdio.h>
main(){
float *px ;
float x=3.5 ;
px=&x ;
printf ("adresse de x : 0x%lx \n",&x) ;
printf ("valeur de px : 0x%lx \n",px) ;
printf ("valeur de x : %3.1f \n",x) ;
printf ("valeur pointee par px : %3.1f \n",*px) ;
return 0 ;
}
```

Le programme précédent affichera ceci à l'écran :

- ✓ L'adresse de x : 0xbfbffa3c
- ✓ La valeur de px : 0xbfbffa3c
- ✓ La valeur de x : 3.5
- ✓ La valeur pointée par px : 3.5

Exercices :

1. Ecrire un programme en C qui permet d'éliminer un nombre x dans un tableau des entiers.
2. On peut représenter un vecteur de l'espace vectoriel R_n à l'aide d'un tableau de n réels. Écrire un programme qui lit deux vecteurs de R_{10}, calcule leur produit scalaire et affiche les deux vecteurs et leur produit scalaire.
3. Ecrire une fonction **somme_div** qui retourne la somme des diviseurs d'un nombre passé en paramètre.
4. Deux nombres M et N sont appelés nombres_amis si la somme des diviseurs de M est égale à N et la somme des diviseurs de N est égale à M.
   Ecrire une fonction **amis** qui retourne le nombre_amis (s'il existe) d'un nombre passé en paramètre, cette fonction utilise la fonction **somme_div** de l'exercice 3.
   Ecrire un programme principal qui affiche tous les nombres_amis inférieurs à une certaine limite.

Pointeurs et fonctions

Un variable globale est une variable connue dans toutes les fonctions d'un programme (main comme les autres).

Une variable locale n'est connue qu'à l'intérieur d'une fonction (main ou une autre).

Les variables locales d'une fonction sont regroupées dans une partie de la mémoire, et celles d'une autre fonction, dans un autre endroit. Ainsi, il peut exister un float a ; dans une fonction et un int a dans une autre sans qu'il y ait de conflit.

Une conscéquence de cette propriété est que la fonction suivante ne marchera pas :

```
void permute (int a , int b){
int tempo;
tempo= a ;
a = b ;
b = tempo ;
return ;
}
```

car lors de l'appel de cette fonction depuis main, les valeurs des arguments vont être copiés dans les variables de permute et ce sont ces variables locales qui vont être modifiées, pas celles de main.

Ainsi, dans main, un appel du type permute(i,j) laissera i et j inchangés. On dit que le C passe ses arguments **par valeur**.

Pour que permute fonctionne, il faut que ses arguments soient les adresses des variables a et b et utiliser des pointeurs.

```
void  permute (int *a , int *b){
int tempo ;
tempo= *a ;
*a = *b ;
*b = tempo;
return ;
}
```

Lors de l'appel de la fonction, les pointeurs locaux vont recevoir l'adresse des variables a et b.

Donc travailler sur ces pointeurs revient à travailler sur les variables a et b de la fonction main.

L'appel d'une telle fonction se fait ainsi : ***permute (&a ,&b)***

De façon générale, on utilise des pointeurs avec les fonctions quand on veut qu'une fonction modifie des variables du programme principal.

Séries d'Exercices :

1. Ecrire un programme en c qui permet de:

- Lire un tableau de 20 réels
- Afficher les éléments de ce tableau
- Chercher et afficher un élément x dans ce tableau
- Calculer et afficher la moyenne et l'écart type des éléments de ce tableau.
- Trier ce tableau par ordre croissant
- Ré-afficher ce tableau trié

2. Refaire l'exercice1 en utilisant des fonctions (Lire_Tableau, Afficher_Tableau, Moyenne, Ecart_type, Rechercher et Trier_Tableau).
3.