Programmation en langage C

FACULTE DES SCIENCES BEN M'SICK
UNIVERSITÉ HASSAN II DE CASABLANCA

Depuis 2017

—

Licence SMI

—

ELFILALI Sanaa

Université Hassan II Casablanca
Faculté de Sciences de Ben M'Sik

Programmation en langage C

SMI_ S3

Pr S.ELFIALI

2017 - 2023

Ce document présente les bases du langage C. Ce sera votre document de référence pendant les séances de travaux dirigés et de travaux pratiques.

Table des matières

Chapitre 1. Introduction

1. Préambule

1.1. Un programme informatique

Un programme informatique est une liste d'instructions écrites dans un ou plusieurs fichiers, destiné à être exécuté par l'ordinateur afin de réaliser une ou plusieurs tâche(s), de résoudre un problème, de manipuler des données.

1.2. Un langage informatique

On appelle langage informatique ou langage de programmation, un langage destiné à décrire l'ensemble des actions consécutives qu'un ordinateur ou plus précisément qu'un processeur doit exécuter.

Le langage utilisé par le processeur, c'est-à-dire les données telles qu'elles lui arrivent, est appelé *langage machine*. Il s'agit d'une suite de 0 et de 1 (du binaire). Toutefois le langage machine n'est pas compréhensible facilement par l'humain.

Ainsi il est plus pratique de trouver un langage intermédiaire, compréhensible par l'homme, qui sera ensuite transformé en langage machine pour être exploitable par le processeur.

2. Historique

Langage C a été créé en 1972 par Dennie Ritchie aux Laboratoires Bell/AT&T avec un objectif de développer une version portable du système d'exploitation UNIX.

Il provient de deux langages : BCPL développé en 1967 par Martin Richards et B développé en 1970 chez AT&T par Ken Thompson.

Il fut limité à l'usage interne de Bell jusqu'en 1978, date à laquelle Brian Kernighan et Dennie Ritchie publièrent la définition classique du langage C (connue sous le nom de *standard **K&R-C***) dans un livre intitulé « The C Programming Language ».

Le succès des années qui suivaient et le développement de compilateurs C par d'autres maisons ont rendu nécessaire la définition d'un standard actualisé et plus précis. En 1983, l'American National Standards Institute' (ANSI) chargeait une commission de mettre au point une définition explicite et indépendante de la machine pour le langage C. Le résultat était le *standard ANSI-C*. La seconde édition du livre « The C Programming Language », parue en 1988, respecte tout à fait le standard ANSI-C et elle est devenue par la suite, la 'bible' des programmeurs en C.

3. Intérêts du langage C

- polyvalent : il permet le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- Il a donné naissance à de nombreux langages dérivés, comme le C++, l'Objective C, le Java et le C#.
- Vaste ensemble de bibliothèques prédéfinies, fournies avec le compilateur.
- Portabilité : En respectant la norme ANSI-C, il est possible d'utiliser le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant.
- Permet un accès à toutes les ressources de la machine (Mémoire, processeur, périphérique, etc.).
- Le C a une grande popularité.

4. De l’édition à l’exécution

Les étapes menant de l’édition à l’exécution d’un programme en C sont : l’édition, la compilation et l’édition de liens.

4.1 L’Edition du programme

L’édition, c’est la rédaction du programme à l’aide de l’éditeur de texte de CodeBlocks ou d’un autre traitement de texte : on parle alors de « programme source ».

En générale, ce texte sera conservé dans un fichier que l’on nommera « fichier source » dont l’extension est « **.C** ».

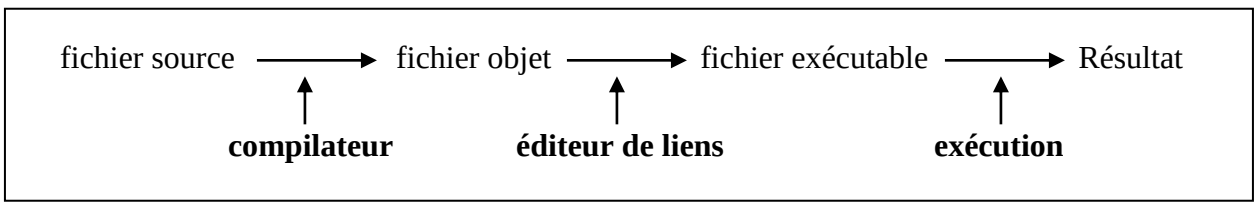
4.1 La compilation

Elle consiste à traduire le programme source en langage machine, en faisant appel à un programme nommé compilateur. Le fichier généré, appelé fichier objet, possède le même nom que le fichier source, mais son extension est « .OBJ ».

4.1 L’édition de liens

Il permet d'intégrer dans le fichier final tous les éléments annexes (fonctions ou librairies) auquel le programme fait référence mais qui ne sont pas stockés dans le fichier source. Le fichier généré est un **fichier exécutable** qui contient tout ce dont il a besoin pour fonctionner de façon autonome.

Ces étapes sont illustrées dans la figure suivante :



Exercice : *Editer, compiler et exécuter le programme suivant:*

```
#include <stdio.h>

void main()
{
        printf("Soyez les bienvenues dans le monde du développement informatique  a la FSB SMI S3!");
}
```

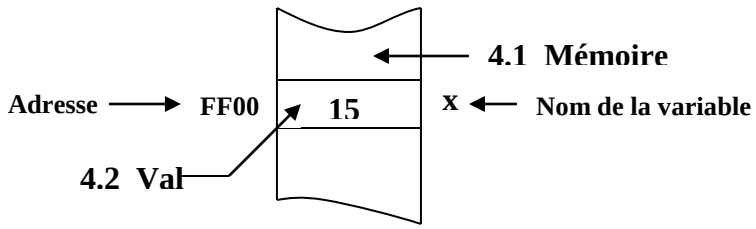
Chapitre 2. Manipulation de données en C

1. Les variables

1.1 Qu'est-ce qu'une variable ?

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

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



1.2 Classification des types de données simples

Le tableau suivant présente tous les types de données simples qui nous permettront de définir les variables.

Type de donnée	Signification	Taille (en octets)	Valeurs limites
Char	caractère	1	-128 à 127
unsigned char	caractère non signé	1	0 à 255
short int	entier court	2	-32 768 à +32 767
unsigned short int	entier court non signé	2	0 à 65 535
Int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à +32 767 -2 147 483 648 à +2 147 483 647
unsigned int	entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	entier long	4	-2 147 483 648 à +2 147 483 647
unsigned long int	entier long non signé	4	0 à 4 294 967 295
Float	flottant (réel)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
Double	flottant double	8	$1.7*10^{-308}$ à $1.7*10^{308}$
long double	flottant double long	10	$3.4*10^{-4932}$ à $3.4*10^{4932}$

1.3 Identificateur d'une variable

Il existe un certain nombre de limites pour choisir l'identificateur d'une variable :

- un identificateur peut contenir des lettres minuscules ou majuscules, des chiffres, ou le caractère spécial de soulignement « _ ». Par contre, il ne doit pas commencer par un chiffre ou posséder des lettres accentuées.
- les espaces ne sont pas admis dans l'identificateur
- un identificateur ne doit pas comporter plus de 32 caractères ;
- les majuscules sont distinguées des minuscules, ainsi : Montant et montant désignent deux noms différents.
- un identificateur ne peut pas être un mot réservé du langage :

Auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
Default	goto	sizeof	volatile
do	if	static	while

Identificateurs valides :

x x1 _x

Identificateurs non valides :

2eme	commence par un chiffre
a#b	caractère non autorisé (#)
num-employe	caractère non autorisé (-)
racine carree	caractère non autorisé (espace)

1.4 Déclaration des variables

Avant d'utiliser une variable dans un programme C, il faut la déclarer c'est-à-dire réserver son emplacement mémoire.

Pour déclarer une variable, on doit :

1. spécifier le type de donnée;
2. indiquer l'identificateur de la variable.

Une variable peut être initialisée lors de sa déclaration.

Syntaxe : Type **identificateur** [= valeur_initiale];

Exemples :

```
int x=-10, y, z=20 ;
char touche = 'A' ;
float hauteur, largeur;
```

2. Constantes littérales

Quand une valeur comme **1** apparaît dans un programme, elle est appelée constante littérale : littérale car on ne peut parler d'elle qu'à travers sa valeur, et constante car sa valeur ne peut être changée. Chaque littéral possède un type associé. Par exemple, **0** est de type entier, **3.1459** est une constante littérale de type double.

Le tableau suivant présente les quatre types de constantes littérales du langage C : les constantes entières, réelles, caractères et chaîne de caractères.

Type	**Description**	**Exemples**
Entière	Décimale	-12, 20
	Octale	015,027
	Hexadécimale	0x1A5, 0xFFFF
Réelle	Décimale	15.25, 1.53**e**2, 314**E**-2
Caractère	Imprimable	'a' ..... '2'.....',' ..... ' ' (espace)
	Non-imprimable	• **\n** : saut de ligne (Line Feed) • **\t** : tabulation horizontale (Horizontal Tab) • **\v** : tabulation verticale (Vertical Tab) • **\b** : retour arrière (Backspace) • **\r** : retour chariot (Carriage Return) • **\f** : saut de page (Form Feed) • **\a** : sonnerie ou bip (Alert)
Chaîne de caractère		"Bonjour"

3. Constantes symboliques : const

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

Syntaxe : **const** type **identificateur** = valeur_initiale;

Exemples : **const** double PI = 22.0 / 7.0;
const float TVA = 0.2;
const float remise ;
remise = 0.4; ⟹ **Erreur !**

4. Opérateurs et expressions

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

Exemples :

- b*b – 4*a*c
- 'A' + 1
- (-b + sqrt(b*b – 4*a*c)) / (2*a)

4.1 Classification et description

Les tableaux suivants présentent la liste de tous les opérateurs disponibles en C.

Opérateurs arithmétiques		
Fonction	**Symbole**	**Remarques**
Addition	+	
Soustraction	-	
Multiplication	*	
Division	/	
Modulo	%	Reste d'une division entre entiers : **7/5 = 1**, reste **2**; donc **7%5 = 2**

Opérateurs de relation		
Fonction	**Symbole**	**Remarques**
Plus grand que	>	
Plus petit que	<	
Plus grand ou égal à	>=	
Plus petit ou égal à	<=	
Egal à	==	Attention! Trop souvent confondu avec l'opérateur d'affectation =
Différent de	!=	
Note : Ces opérateurs servent à comparer des expressions. Le résultat des opérations est VRAI ou FAUX.		

Opérateurs logiques		
Fonction	**Symbole**	**Remarques**
ET	&&	**((a>2) && (a<10))** → **VRAI** si **a = 6**, **FAUX** si **a = 16**
OU	||	**((a<2) || (a>10))** → **FAUX** si **a = 6**, **VRAI** si **a = 16**
Négation	!	**!((a<2) || (a>10))** → **VRAI** si **a = 6**, **FAUX** si **a = 16**
Note : Ces opérateurs servent à tester l'état logique d'une expression : **VRAI** ou **FAUX**, **OUI** ou **NON**, **0** ou **1**. **Une valeur numérique peut également être testée : VRAI** si différente de **0, FAUX** si égale à **0**.		

Opérateurs de manipulation de bits		
Fonction	**Symbole**	**Remarques**
ET	&	**c = a & b**; → si **a = 1001 0011** et **b = 0101 0110**, **c = 0001 0010**
OU	|	**c = a | b**; → si **a = 1001 0011** et **b = 0101 0110**, **c = 1101 0111**
OU exclusif	^	**c = a ^ b**; → si **a = 1001 0011** et **b = 0101 0110**, **c = 1100 0101**
Décalage à droite	>>	**(a>>2)** décale la valeur de a de **2** bits vers la droite
Décalage à gauche	<<	**(a<<2)** décale la valeur de a de **2** bits vers la gauche
Complément à un	~	**c = ~a** → si **a = 1001 0011**, **c = 0110 1100**
Note : Ces opérateurs ne modifient pas la valeur de la variable, sauf si le résultat de l'opérateur affecte celle-ci : **a >> 2** ne change pas la valeur de **a**, **a = a >> 2** change la valeur de **a**.		

Opérateurs d'affectation		
Fonction	**Symbole**	**Remarques**
Affectation simple	=	
Ajoute et affecte	+=	**a += b**; équivaut à **a = a + b** ;
Soustrait et affecte	-=	**a -= b**; équivaut à **a = a - b** ;
Multiplie et affecte	*=	**a *= b**; équivaut à **a = a * b** ;
Divise et affecte	/=	**a /= b**; équivaut à **a = a / b** ;
Modulo et affecte	%=	**a %= b**; équivaut à **a = a % b** ;
OU (bit) et affecte	|=	**a |= b**; équivaut à **a = a | b** ;

OU exclusif (bit) et affecte	^=	**a ^= b**; équivaut à **a = a ^ b** ;
ET (bit) et affecte	**&=**	**a &= b**; équivaut à **a = a & b** ;
Décale à droite et affecte	>>=	**a >>= b**; équivaut à **a = a >> b** ;
Décale à gauche et affecte	<<=	**a <<= b**; équivaut à **a = a << b** ;
Note : Ces opérateurs modifient la valeur des variables.		

Opérateur d'évaluation séquentielle		
Fonction	**Symbole**	**Remarques**
Evaluation séquentielle	**,**	**int a, b; b = 3, a = 94;**

Opérateur conditionnel		
Fonction	**Symbole**	**Remarques**
Evaluation conditionnelle	**?:**	**min = (a<b) ?a :b;** → si **(a<b) est VRAI, min=a ;.**sinon **min=b ;**
Note : L'utilisation de cet opérateur fait partie des thèmes traités au chapitre 4.		

Opérateurs unaires		
Fonction	**Symbole**	**Remarques**
Incrémentation	++	**c=a + (b++)** ;→ **b=b + 1 après** exécution de l'instruction **c=a + (++b)** ;→ **b=b + 1 avant** exécution de l'instruction
Décrémentation	**- -**	**c=a + (b--)** ;→ **b=b - 1 après** exécution de l'instruction **c=a + (--b)** ;→ **b=b - 1 avant** exécution de l'instruction
Conversion explicite	**(nom Type)**	**char a; int b; b=(int)a**;→ une copie de la valeur de **a** est transformée en **int** avant d'être déposé dans **b**.
Adressage indirect	*	***ptr=a** ; → copie la valeur de **a** à l'adresse pointée par **ptr.**
Adresse de	**&**	**&a** est l'adresse de **a.**
Evalue grandeur mémoire	**sizeof**	**sizeof(expr). sizeof(nom Type).**
Plus unaire	+	force l'évaluation d'une expression avant une autre à cause de son niveau de priorité.
Moins unaire	-	**-a** est le **complément à 2** de **a**.
Note : Un opérateur unaire est un opérateur qui nécessite 1 seul opérande : **a++.** Un opérateur binaire est un opérateur qui nécessite 2 opérandes : **a + b**.		

Opérateurs primaires		
Fonction	**Symbole**	**Remarques**
Parenthèse	**( )**	
Expression d'indice	**[ ]**	**char tableau[8] ; tableau[3] = a + tableau[5]** ;
Sélecteur de membre	->	Sélection par pointeur : **ptrstruct->mois = 12 ;**
Sélecteur de membre	**.**	Sélection par la structure : **date.mois = 12 ;**

4.2 Niveau de priorité des opérateurs

Le niveau de priorité des opérateurs détermine dans quel ordre ceux-ci seront appliqués lors de l'évaluation d'une expression : les opérateurs de niveau plus élevé seront toujours évalués en premier.

Lorsque plusieurs opérateurs de même niveau de priorité se trouvent dans une expression, c'est l'associativité (gauche à droite *ou* droite à gauche), qui détermine si les opérations seront effectuées de droite à gauche ou de gauche à droite.

Priorité des opérateurs			
Niveau de priorité	**Type d'opérateur**	**Opérateurs**	**Associativité**
15	Primaire	**( ) [ ] . ->**	gauche à droite
14	Unaire	**++ -- (nom Type) * & sizeof + - ! ~**	**droite à gauche**
13	Arithmétique	*** / %**	gauche à droite
12	Arithmétique	**+ -**	gauche à droite
11	De manipulation de bits	**>> <<**	gauche à droite
10	De relation	**> < >= <=**	gauche à droite
9	De relation	**== !=**	gauche à droite
8	De manipulation de bits	**&**	gauche à droite
7	De manipulation de bits	^	gauche à droite
6	De manipulation de bits	|	gauche à droite
5	Logique	**&&**	gauche à droite
4	Logique	||	gauche à droite
3	Conditionnel	**?:**	**droite à gauche**
2	D'affectation	**= += -= *= /= |= ^= &= >>= <<=**	**droite à gauche**
1	D'évaluation séquentielle	**,**	gauche à droite

5. Les Instructions

- Une instruction simple est soit une expression terminée par *un point virgule*, soit une instruction de contrôle (traitée au chapitre 4) soit un bloc d'instructions délimité par **{** et **}**.

  Exemple : a=a +1**;**

- Les instructions composées (ou blocs) qui permettent de considérer une succession d'instructions comme étant une seule instruction :
  - elles commencent par "{" et finissent par "}"
  - l'accolade ouvrante est l'équivalente du début en algorithme
  - l'accolade fermante est l'équivalente de la fin en algorithme

  Exemple :

```
{
        float tauxConversion=6.55957, valeurEnFranc;
        int valeurEnEuro=50;
        valeurEnFranc = valeurEnEuro * tauxConversion;
}
```

Chapitre 3. Les entrées-sorties conversationnelles

La bibliothèque standard contient un ensemble de fonctions d'entrées-sorties conversationnelles (lecture au clavier, affichage à l'écran).

1. Fonction printf()

La fonction **printf()** permet d'afficher du texte, des valeurs de variables ou des résultats d'expressions à l'écran.

Syntaxe : printf(format[,arg1, arg2,…,argn]).

- Le paramètre **format** désigne une chaîne de caractères comprenant :

  1. des caractères à afficher tels quels,
  2. des caractères non imprimables (séquences d'échappement),
  3. des spécificateurs de format : suite de caractères précédée du symbole % précisant la manière dont les valeurs des arguments <arg1..n> sont affichées.

Une spécification de format est de la forme :

% [drapeau] [largeur] [.précision] [modificateur] code_conversion.

- Les arguments à afficher peuvent être des constantes littérales ou des variables ou des expressions.

Note : Il doit y avoir autant de spécificateurs de format que d'arguments.

1.1 Le champ code de conversion

Le format minimum d'impression est constitué du caractère % suivi d'un code de conversion qui spécifie le type de donnée de l'argument à afficher, dont voici les principaux :

Code de conversion	Type	Remarque
d ou i	int	
U	unsigned int	

O	int	affiché en octal
x ou X	int	affiché en hexadécimale
C	char	
F	float ou double	
e ou E	float ou double	affiché en notation scientifique
S	char*	

Exemple 1 :

```
int q = 100;
float p = 60.50;
char t =’A’;

printf("Quantite = %d\n",q);
printf("Prix = %f\n", p) ;
printf("Taille = %c\n",t) ;
printf("Prix total = %f", q*p);
```

ou bien

```
printf("Quantite = %d\nPrix = %f\nTaille = %c\nPrix total = %f",q,p,t,q*p);
```

Imprime

```
Quantite = 100
Prix =    60.500000
Taille = A
Prix total = 6050.000000
```

1.2 Le champ modificateur

Le champ modificateur est optionnel, il est constitué d’une simple lettre (l, h ou L) placée devant le champ code de conversion qui spécifie **short** ou **long.**

Exemples des combinaisons typiques des champs modificateurs et codes de conversion :

ld ou li : **long int**.
lu : **unsigned long int.**
hd : **short int**.
Lf ou Le : **long double.**

1.3 Spécificateur de précision

Par défaut, les flottants sont affichés avec six chiffres après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle).Il est possible de modifier cette représentation en utilisant un point décimal et un spécificateur de précision.

Exemple 2 :

```
float val = 25.1234;

printf("%f",val);          /* affiche : 25.123400 */
printf("%.0f",val);        /* affiche : 25 */
printf("%.1f",val);        /* affiche : 25.1*/
printf("%.2f",val);        /* affiche : 25.12*/
printf("%e",val);          /* affiche : 2.512340e+01*/
printf("%.3e",val);        /* affiche : 2.512e+01*/
```

1.4 Le champ largeur

Le champ *largeur* permet de spécifier le nombre minimal de caractères à afficher.

Exemple 3 :

```
int val1 = 125;
float val2 = 4.5;

printf("|%d|",val1);          /* affiche : |125| */
printf("|%2d|",val1);         /* affiche : |125| */
printf("|%5d|",val1);         /* affiche : |--125| */
printf("|%05d|",val1);        /* affiche : |00125| */

printf("|%8.2f|\n",val2);     /* affiche : |----4.50| */
```

1.5 Le champ drapeau

- "-" : Cadrage à gauche (sinon à droite par défaut).
- "+" : affichage du signe pour les nombres positifs.

Exemple 4 :

```
int val1 = 125;
float val2 = 4.5;

printf("|%-5d|",val1);        /*affiche : |125--|*/
printf("|%+05d|",val1);       /*affiche : |+0125|*/

printf("|%-8.2f|\n",val2);    /*affiche : |4.50----|*/
```

2. Fonction scanf()

La fonction *scanf* permet de lire des données au clavier et de les affecter à des variables.

Syntaxe : scanf(format,arg1, arg2,…,argn).

Une spécification de format est de la forme :

%[largeur][modificateur] code_conversion.

Remarques :

- Chaque nom d'argument qui correspond à un scalaire doit être précédé de & (opérateur d'adresse), ceci indique que l'information saisie va être placée à l'adresse de la variable.
- La chaîne de format ne doit comporter que des spécifications de format, sinon tout autre caractère indiqué dans la chaîne de format doit être saisi par l'utilisateur.
- Le code de conversion **"f"** s'applique aux données de type **float et double** dans le cas de la fonction printf(). Par contre, pour la fonction scanf(), une distinction s'impose : **"f"** ne s'applique qu'aux données de type **float** ; pour les données de type **double**, il faut utiliser le code de conversion **"lf"** qui signifie **long float.**
- Le champ largeur change par rapport à printf. Il permet de spécifier le nombre maximal de caractères à lire.

Exemples :

- Saisie d'une variable

```
{
        float rayon, perimetre;
        printf("Donnez le Rayon : ");
        /* Saisie du Rayon */
        scanf("%f",&rayon);
        /* Calcul du périmètre */
        perimetre = 2*3.14*rayon;
        /* Affichage de périmètre */
        printf("Le périmètre = %.2f", perimetre);
}
```

- Saisie de plusieurs variables.

```
{
        float a, b, c, det ;
        printf("Donnez les valeurs de a,b et c : ");
        /* Saisie de a, b, c */
        scanf("%f %f %f",&a,&b,&c);
        /* Calcul du déterminant */
        det = b*b- 4*a*c;
        printf("Le déterminant = %.2f", det);
}
```

Tampon de scanf

Les informations tapées au clavier sont d'abord mémorisées dans un emplacement mémoire appelé *tampon* (buffer). Elles sont mises à la disposition de scanf après l'activation de la touche <Entrée>

- Pour les nombres : scanf avance jusqu'au premier caractère différent d'un séparateur (espace, tabulation, retour à la ligne) puis scanf prend en compte tous les caractères jusqu'à la rencontre d'un séparateur ou d'un caractère invalide ou en fonction du largeur.
- Pour les caractères %c : scanf prend le caractère courant séparateur ou non séparateur.

Exemple (^ = espace, @ = fin de ligne) :

```
scanf("%d%d",&a,&b);
123^45@          ==>      a = 123  b = 45
10@@20@          ==>      a = 10    b = 20

scanf("%d%c",&a,&b);
200^z            ==>       a = 200 b = ' '

scanf("%d^ %c",&a,&b);
200^z            ==>       a = 200 b = 'z '

scanf("%d",&a);
15@              ==>       a = 15
scanf("%c",&b);  ==>      b = '\n'
```

4.2 3. Exemple de programme complet

Le programme suivant calcule la moyenne de deux nombres entrés au clavier et l'affiche :

```
#include <stdio.h>

void main()        // Point d'entré du programme
{          /*Début du programme*/
           float x, y;

           printf("\t\t\tCalcul de la moyenne\n\n");      /*Affiche le titre*/
           printf("Entrez le premier nombre : ");
           scanf("%f", &x);                              /*Lecture du premier nombre*/
           printf("Entrez le deuxieme nombre : ");
           scanf("%f", &y);                              /*Lecture du deuxième nombre*/
           printf("La valeur moyenne de %.2f et de %.2f est %.2f\n",x, y,(x+y)/2);
}          // Fin du programme
```

Commentaire :

Inclusion des fichiers : La première ligne contient la directive **#include** suivi d'un nom de fichier a pour effet d'insérer le fichier spécifié entre < et >, ici stdio.h, dans le fichier source à l'endroit où la directive est placée. Le fichier d'en-tête **stdio.h** contenant les déclarations nécessaires à l'utilisation des fonctions d'entrées-sorties standard. Le compilateur dispose ainsi des informations nécessaires pour vérifier si l'appel de la fonction (en l'occurrence printf et scanf) est correct.

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

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

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

Chapitre 4. Les instructions de contrôle

Les instructions de contrôle servent à contrôler le déroulement de l'enchaînement des instructions à l'intérieur d'un programme, ces instructions peuvent être des instructions conditionnelles ou itératives.

1. Les instructions conditionnelles

Les instructions conditionnelles permettent de réaliser des tests, et suivant le résultat de ces tests, d'exécuter des parties de code différentes.

Le langage C offre deux types d'instructions conditionnelles :

- l'instruction **if**
- l'instruction **switch**

1.1 Instruction conditionnelle if

A) l'instruction if-else

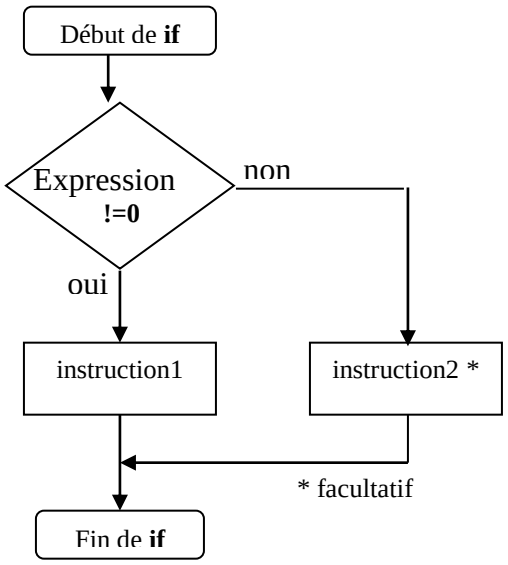
Syntaxe :

if (<*expression*>) <*instruction1*>
if (<*expression*>) <*instruction1*> **else** <*instruction2*>

Description :

- Instruction1 et instruction2 : sont des instructions quelconques, c'est-à-dire :
  - simple,
  - bloc,
  - instruction structurée.
- Si le résultat de l'expression est « vrai », c'est-à-dire non nul, on exécute l'instruction <*instruction1*>, dans le cas contraire on exécute l'instruction <*instruction2*> si elle existe. Puis, le programme passe à l'exécution de l'instruction suivante.

Diagramme syntaxique du if :



Exemples :

```
int qte_cmd ;
float prix, remise=0 ;

printf("La quantité commandée : ") ;scanf("%d",&qte_cmd) ;
printf("Prix unitaire : ") ;scanf("%f",&prix) ;
if(qte_cmd > 100) remise = 0.1;
printf("Prix à payer : %.2f",prix*qte_cmd*(1 – remise) ) ;
```

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

```
if (qte_cmd <= qte_stock)
{
    printf("Prix total = %.2f", qte_cmd*prix);
    qte_stock -= qte_cmd;
}
else
    printf("\aLa quantité commandée est sup. à la quantité en stock ! ");
```

Exercice 1 :

L'utilisateur saisit un caractère, le programme teste s'il s'agit d'une lettre majuscule, si oui il affiche cette lettre en minuscule, sinon il affiche un message d'erreur.

Solution :

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Exercice 2 :

Quel résultat affiche le programme suivant :

```
#include<stdio.h>
void main()
{
        int x = 3 ;
        if (x < 0) ;
        {
                x = -x ;
                printf("x = %d ", x);
        }
        if (x = 4) printf("x = %d", x);
}
```

Solution :

.....

B) Imbrication d'instructions if

Il est possible d'imbriquer une instruction **if** à l'intérieur d'une autre.

Syntaxe :

if (<*expression1*>) **if** (<*expression2*>) <*instruction1*> **else** <*instruction2*>

Exemple :

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

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

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

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

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

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

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

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

C)L'instruction if-else if

Il est fréquemment utile d'utiliser une série d'instructions **if-else if** pour répondre à une situation dans laquelle les choix sont multiples.

Syntaxe :

```
if (<expression1>) <instruction1>
else if (<expression2>) <instruction2>
else if (<expression3>) <instruction3>
...
[else   <instruction_n+1>]
```

Chaque instruction est associée à une condition et le dernier else, s'il existe, correspond au cas où aucune condition n'est vraie.

Exemples :

```
if ( qte < 1000)
    taux_remise = 5;
else if ( qte < 10000)
        taux_remise = 10;
     else
        taux_remise = 15;
```

```
#include<stdio.h>
void main()
{
        char  c;
        printf("Entrez une lettre non accentuée : ");
        scanf("%c", &c);
        if (c >='A' && c<='Z')
                printf("Cette lettre en minuscule est : %c",c+32);
        else if (c >='a' && c<='z')
                printf("Cette lettre en majuscule est : %c",c-32) ;
             else
                printf("Erreur ! Ce n'est pas une lettre non accentuée") ;
}
```

Exercice 3 :

Ecrivez un programme qui permet de saisir une valeur de type entière et indiquez à l'utilisateur si celle-ci est positive, négative ou nulle.

Solution :

……
……
……
……
……
……
……
……
……
……
……
……
……

Remarque :

Le langage C offre la possibilité de remplacer l'instruction conditionnelle **if-else** dans le cas d'une affectation conditionnelle de variable par l'opérateur conditionnel « **?:** » et qui a l'avantage de pouvoir être intégré dans une expression.

Syntaxe :

```
expr1 ? expr2 : expr3
```

On évalue **expr1**:

- Si elle n'est pas nulle (donc elle est vraie), alors la valeur de <**expr2**> est fournie comme résultat.
- Sinon, la valeur de <**expr3**> est fournie comme résultat.

Exemples :

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

⟺

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

⟺

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

1.2 Instruction conditionnelle switch

L'instruction conditionnelle **swtich** permet de remplacer plusieurs if-else imbriqués lorsqu'il s'agit d'effectuer un choix multiple.

Sa syntaxe est la suivante :

```
switch (expression)
{
        case expression_const_1 :
                        instruction_1;
                        [break;]
        case expression_const_2 :
                        instruction_2;
                        [break;]
        ….

        case expression_const_n :
                        instruction_n;
                        [break;]
        [default:
                        instruction_par_défaut;]
}
```

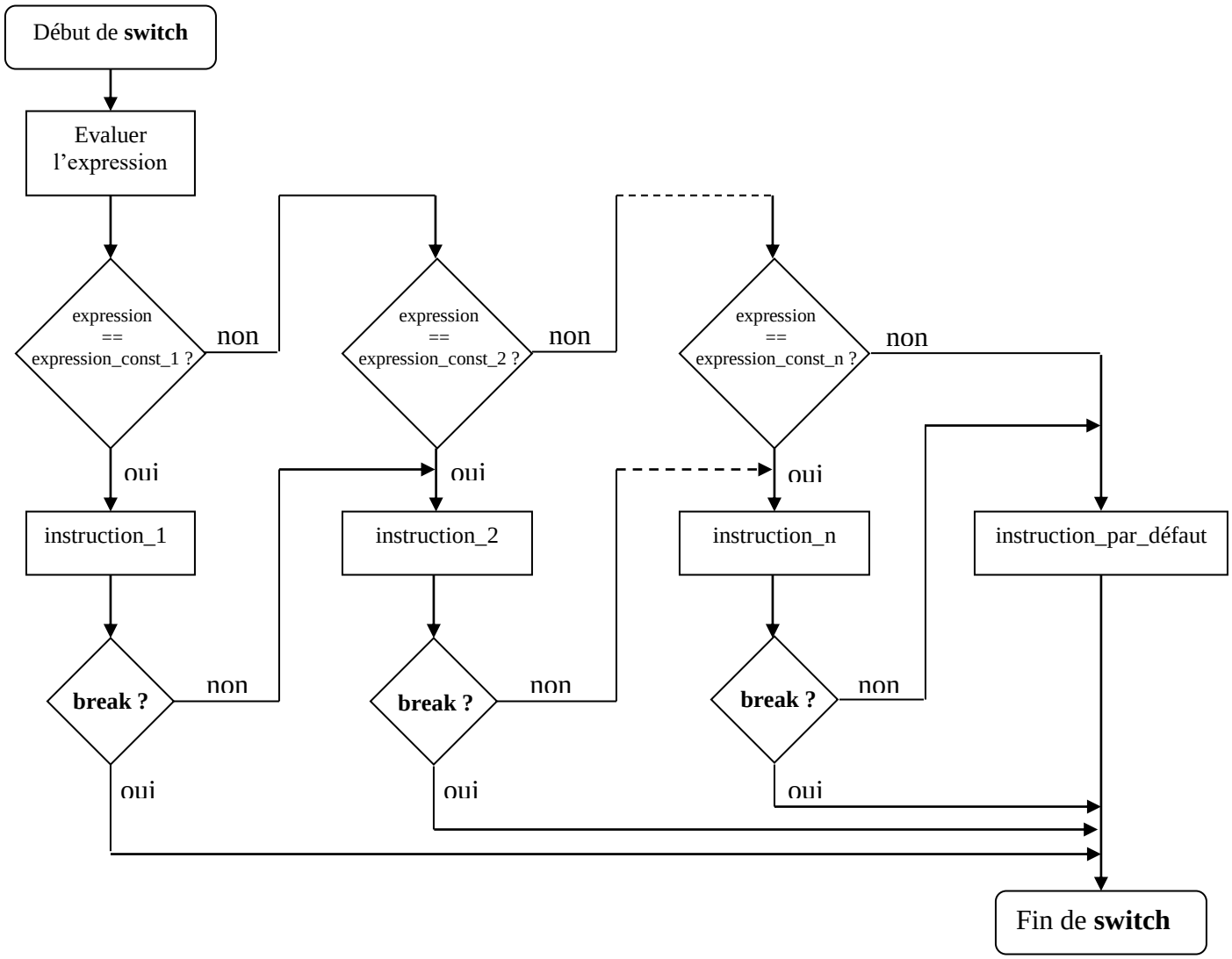
Notes :

- **expression** et **expression_const_i** ne peuvent être ni de type réel ni de type chaîne de caractère.
- **expression_const_1,expression_const_2,…,expression_const_n** doivent obligatoirement être distinctes.
- **break** et **default** sont optionnelles.

<u>Description</u> :

L'<expression> est évaluée puis le résultat est comparé à chacune des expressions constantes spécifiées après les différentes cases. Si l'<expression> vaut l'<expression_const_i>, l'exécution se poursuit par l'<instruction_i> jusqu'à la fin du switch à moins de rencontrer une instruction break qui permet de la terminer. Si l'<expression> ne correspond à aucune <expression constante> alors l'<instruction_par_défaut> qui sera exécuter.

Diagramme syntaxique de switch :



Exemples :

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

⇔

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

2. Instructions itératives

Les instructions itératives ou boucles sont réalisées à l'aide d'une des trois instructions de contrôle suivantes : **do-while, while et for**.

2.1. Instruction do-while <faire-tant que> :

La boucle *do-while* permet de répéter une instruction ou un bloc d'instructions, un certain nombre de fois, tant qu'une condition est vérifiée.

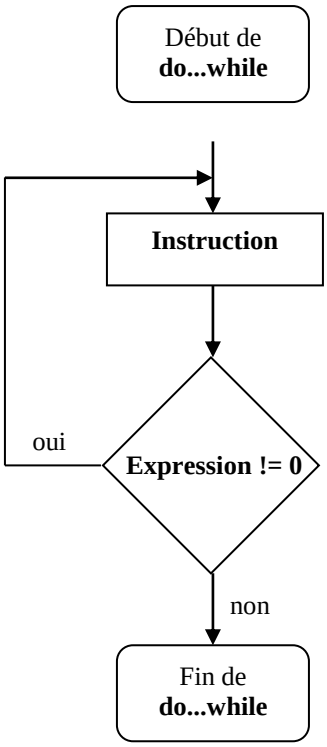
Syntaxe :

```
do
        <instruction>   /*ou <bloc d'instructions> */
while (<expression>);
```

Description :

La boucle **do-while** débute par l'exécution de l'<instruction>, puis l'<expression> est évaluée. Tant qu'elle est VRAIE, l'exécution de l'<instruction> est répétée ; lorsqu'elle devient fausse, le programme quitte la boucle et passe aux instructions suivantes.

Diagramme syntaxique :



Remarques :

- L'<instruction> est exécutée au moins une fois, car le test de l'<expression> s'effectue à la fin de la boucle.
- Il faut s'assurer que l'<instruction> modifie la valeur de l'<expression> si on veut éviter d'être pris dans une boucle sans fin.

```
void main ()
{
        int compteur = 1;
        do
        {
                printf("La valeur du compteur vaut : %d\n", compteur);
                compteur++ ;
        }while (compteur <=10);
        printf("La valeur finale du compteur vaut : %d",compteur);
}
```

Exemples :

```
#define PI 22/7.0
void main ()
{
        float rayon ;
        do
        {
                printf("Donnez le rayon : ") ;scanf("%f",&rayon);
                if(rayon < 0) printf("\aLe rayon doit être positif\n") ;
        }while (rayon < 0);
        printf("Aire = %.2f",rayon*rayon*PI);
}
```

```
#include <stdio.h>
void main ()
{
        char rep ;
        do
        {
                ….
                printf("Vous-voulez continuer (o/n) : ") ;
                scanf(" %c",&rep);  /* rep = getche(); */
        }while (rep == 'o' || rep == 'O');
}
```

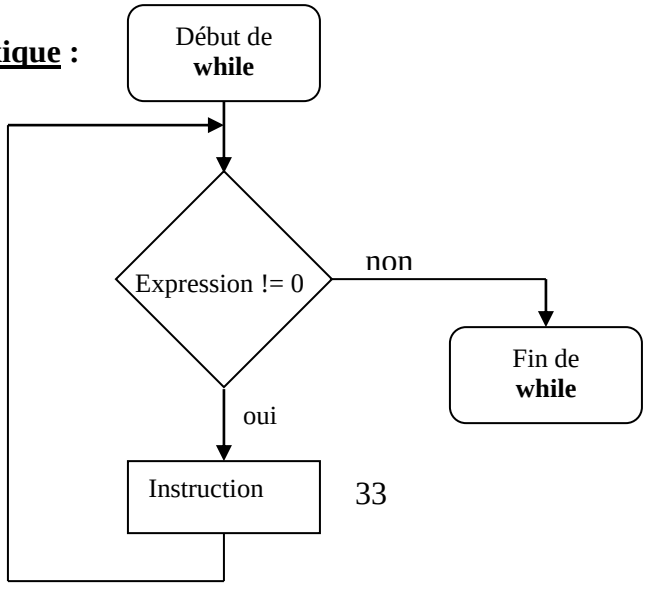
2.2 L'instruction while <tant que> :

L'instruction itérative **while** se comporte comme la boucle **do-while**, à la différence près que l'<expression> est évaluée en début de boucle. Il y a donc possibilité que l'<instruction> à répéter ne soit jamais exécutée, si le premier test n'est pas vérifié.

Syntaxe :

```
while (<expression>)
        <instruction>; /*ou <bloc d'instructions>*/
```

Diagramme syntaxique :



Exemple :

```
#include <conio.h>
void main ()
{
        int color = 1 ;
        while (color <= 15)
        {
                textcolor(color) ;
                cprintf("Couleur N° %d\n\r", color++) ;
        }
}
```

<u>Exercice 4</u> :

Ecrire un programme qui demande un entier positif n et qui affiche la somme des n premiers entiers.

Solution :

……
……
……
……
……
……
……
……
……
……
……
……
……
……
……

Exercice 5 :

Ecrire un programme qui calcule la somme des nombres positifs donnés par l'utilisateur, le programme lira des nombres tant qu'ils seront positifs.

Solution :

……
……
……
……
……
……
……
……
……
……
……
……
……
……

2.3 Instruction itérative for <Pour> :

La syntaxe de la boucle for en C est la suivante :

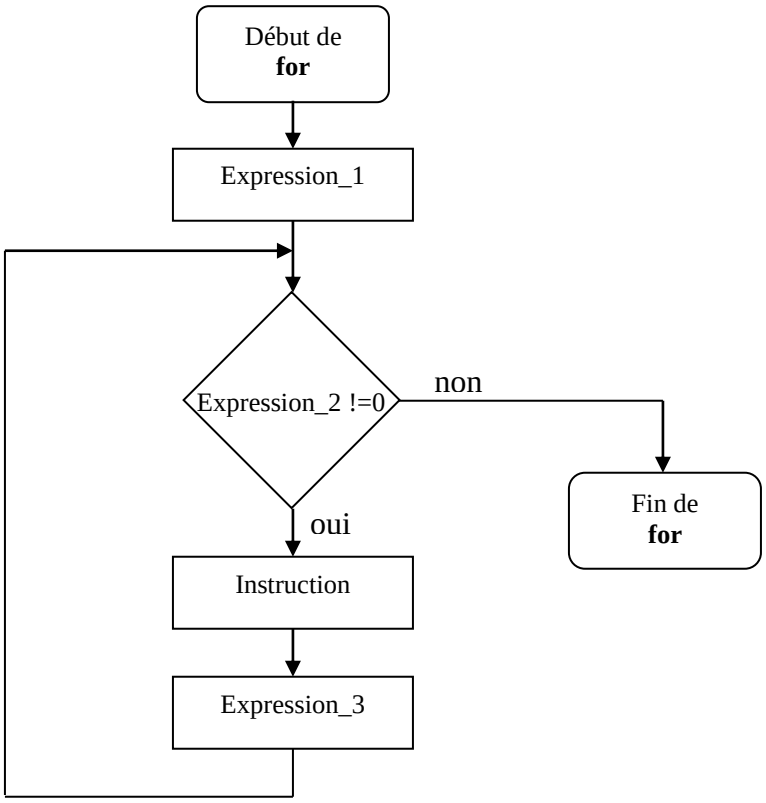
```
for ([expression_1];[expression_2];[expression_3])
        <instruction>; /*ou <bloc d'instructions>*/
```

Description :

La boucle for s'utilise avec trois expressions, séparées par des points virgules :

- expression_1 : est évaluée une seule fois, au début de l'exécution de la boucle . Elle sert à initialiser les données de la boucle.
- expression_2 : est la condition d'exécution de l'<instruction> à répéter. Elle est évaluée et testée avant chaque parcours de la boucle. Si son résultat est VRAI alors l'<instruction> est exécutée sinon la boucle est terminée.
- expression_3 : est évaluée après l'exécution de l'<instruction> à répéter. Elle utilisée pour mettre à jour les données de la boucle.

Diagramme syntaxique :



Remarques :

- La boucle for est équivalente à la structure suivante :

```
expression_1;
while (expression_2)
{
        instruction ;
        expression_3;
}
```

  - Par rapport à la boucle while, la boucle for met en valeur l'<expression_1> et l'<expression_3>.

  - Expression_1, expression_2 et expression_3 sont facultatives. Si l'<expression_2> est omise alors la boucle est infinie.

  - Quand tout le traitement peut être spécifié dans les trois expressions du for, l'instruction à répéter peut se réduire à une instruction vide.

Exemples :

```
#include <stdio.h>
void main ()
{
        int m,i ;
        printf("Donnez le multiplicande  compris entre 1 et 9 : ") ;
        scanf("%d",&m);
        printf("\n\t\t\tTable de multiplication par %d\n",m) ;
        for (i=1; i <= 10 ; i++)
                printf("%2d x %2d = %2d\n",m,i,m*i) ;
}
```

Exercice 6 :

Ecrire un programme qui affiche la somme des **n** premiers termes d'une progression arithmétique de raison **r** et de premier terme **m**.

Par exemple, si **n** = 7, **r** = 3 et **m** = 23 alors **somme** = 23 + 26 + 29 + 32 + 35 + 38 + 41 **= 224**

Solution :

……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……

3. Remarque sur les instructions itératives

Les différentes boucles peuvent être imbriquées.

Exemple :

```
#include <stdio.h>
#include <conio.h>
void main ()
{
        int m,n ;
        for (m=1; m <= 9 ; m++)
        {
                clrscr();
                printf("\t\t\tTable de multiplication par %d\n\n",m) ;
                for (n=1; n <= 10 ; n++)
                        printf("\t\t\t\t%2d x %2d = %2d\n",m,n,m*n);
                getch();
        }
}
```

Lorsque l'on imbrique des instructions for, il faut veiller à ne pas utiliser le même compteur pour chacune des instructions.

4. Les instructions de rupture de séquences

Ces instructions permettent de rompre le déroulement séquentiel d'une suite d'instructions.

4.1 L'instruction break

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

Exemples :

```
#include <stdio.h>
void main()
{
    int i;
    for(i=0;  ;i++)
    {
        printf("La valeur du compteur vaut : %d\n", i);
        if(i == 10) break;
    }
}
```

⟺

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

Remarque :

- L'instruction break ne peut être utilisée que dans le corps d'une boucle ou d'un switch.
- L'action de break ne s'applique qu'à la boucle la plus intérieure dans laquelle elle se trouve. Elle ne permet pas de sortir de plusieurs boucles imbriquées.

4.2 L'instruction continue

Elle relance immédiatement la boucle **while, do, for**, dans laquelle elle se trouve.

- pour les boucles **while et do**, la condition d'arrêt est immédiatement réévaluée.
- pour la boucle **for**, on passe à l'évaluation de l'<expression_3>. C'est pour cette raison que l'équivalence entre le for et le while n'est pas totale.

Exemple :

```
#include <stdio.h>
void main()
{
    int i;
    for(i=0; i<=20; i++)
    {
        if(i%2 == 0) continue;
        printf("La valeur du compteur vaut : %d\n", i);
    }
}
```

4.3 L'instruction goto

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

Sa syntaxe est la suivante :

```
goto étiquette ;
```

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

```
étiquette :
        instruction ;
```

Exemple :

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

Remarque :

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

Chapitre 5. Les tableaux

Il est courant d'avoir besoin de gérer plusieurs éléments qui appartiennent au même type de données. A titre d'exemple, supposez que nous souhaitions déterminer, à partir de 30 notes fournies en donnée, combien d'élèves ont une note inférieure à la moyenne de la classe. Pour parvenir à un tel résultat, nous devons :

- Déterminer la moyenne des 30 notes, ce qui demande de les lire toutes,
- Déterminer combien parmi ces 30 notes, sont inférieures à la moyenne précédemment obtenue.

Vous constatez que si nous ne voulons pas être obligé de demander deux fois les notes à l'utilisateur, il nous faut les conserver en mémoire. Pour ce faire, il paraît peu raisonnable de prévoir 30 variables différentes. Le tableau va nous offrir une solution convenable à ce problème.

1. Définition :

Un tableau représente un ensemble d'emplacements mémoire regroupés d'une façon séquentielle, qui portent le même nom et contiennent le même type de données. Chacun de ces emplacements est appelé élément du tableau.

Un tableau peut être à une ou à plusieurs dimensions. Il n'y a pas de limite au nombre de dimensions.

2. Les tableaux à une dimension :

Un tableau à une dimension à **N** éléments est représenté en mémoire comme suit :

donnée	**donnée**	**donnée**	**……**	**donnée**
Élément **1**	Élément 2	Élément 3		Élément **N**

2.1 Déclaration d'un tableau

La forme générale de la déclaration d'un tableau à une dimension est :

Type Nom_du_tableau **[***nb_elements***]***;*

Cette déclaration réserve en mémoire nb_elements emplacements consécutifs en mémoire permettant de stocker des éléments du type demandé.

Exemple : float notes[30];
Remarques :

- Le nombre d'éléments d'un tableau ne peut être une variable.
- Pour avoir un programme plus **évolutif** on utilise une constante symbolique pour la taille d'un tableau.

Exemple :

```
#define NB_ETUDIANTS 30
…
float notes[NB_ETUDIANTS];
```

2.2 Initialisation d'un tableau

Tout comme pour les variables simples, il est possible d'**initialiser** un tableau lors de sa déclaration en mettant entre accolades les valeurs, séparées par des virgules :

int Tab1[5]={10, 20, 13, 4, 5}; ⇨ Tab1

10	**20**	**13**	**4**	**5**

float Tab2[5]={1.5,6.75}; ⇨ Tab2

1.5	**6.75**	**0**	**0**	**0**

int Tab3[]={10, 2 , 75}; ⇨ Tab3

10	**2**	**75**

int Tab4[3]={10,20,30,40,50}; **/* Erreur ! */**
Remarques:

- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau
- Les valeurs entre accolades doivent être des constantes (l'utilisation de variables provoquera une erreur du compilateur)
- Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments sont initialisés à 0
- Il doit y avoir au moins une valeur entre accolades
- Si le nombre d'éléments n'est pas indiqué explicitement lors de l'initialisation, alors le compilateur réserve automatiquement le nombre d'octets nécessaires.

2.3 Accès aux éléments d'un tableau

Pour accéder à un élément du tableau, le nom que l'on a donné à celui-ci ne suffit pas car il comporte plusieurs éléments. Ainsi, Un élément du tableau est identifié par le nom du tableau et, entre crochets, l'indice représentant sa position dans le tableau :

Nom_du_tableau [indice]

- L'indice du premier élément du tableau est **0.**
- L'indice du dernier élément du tableau est égal au nombre d'éléments **– 1.**
- L'indice i désigne le i+1ème élément du tableau.
- L'indice peut être n'importe quelle expression arithmétique entière.
- L'accès hors tableau n'est pas détecté.

Ainsi, on accédera au 6ème élément du tableau en écrivant : Nom_du_tableau[5].

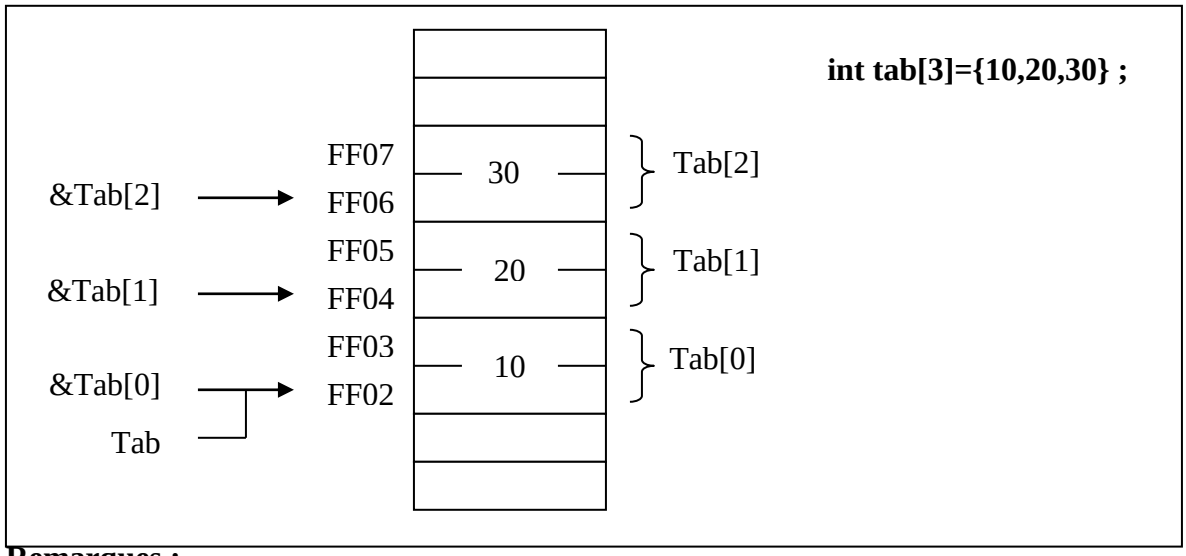
Exemple 1 :

```
#define Nmax 10

void main()
{
      /* Déclaration d'un tableau d'entiers de 10 éléments*/
      int tab[Nmax] , indice;

      /* Initialisation du tableau tab*/
      for(indice=0; indice<Nmax; indice++)
            tab[indice] = indice*2;

      /* Affichage du tableau tab*/
      for(indice=0; indice<Nmax; indice++)
            printf("Tab[%d]= %d\n",indice,tab[indice]);
}
```

2.4 Adressage des éléments d'un tableau

Deux méthodes différentes sont utilisées pour obtenir l'adresse d'un élément : la méthode de l'**indice** et la méthode de l'**offset (**fait partie des thèmes traités au chapitre 6**)**.

La méthode de l'indice est illustrée à la figure suivante :



Remarques :

- La notation **&**Tab[i] désigne l'adresse de l'élément Tab[i].
- L'identificateur du tableau correspond à l'adresse du premier élément du tableau :

Tab = &Tab[0] = FF02

Exemple 2 :

```
#define NB_ELEVES 30

void main()
{
      float notes[NB_ELEVES], som, moy;
      int i, nbr;

      printf("Donnez vos %d notes : \n\n", NB_ELEVES);

      for(i=0, som=0; i<NB_ELEVES; i++)
      {
             printf("Note N°%d : ",i+1);
             scanf("%f",&notes[i]);
             som += notes[i];
      }

      for(nbr=0, moy=som/NB_ELEVES, i=0; i<NB_ELEVES; i++)
            if(notes[i] < moy) nbr++;

      printf("La moyenne de la classe est %.2f\n",moy);
      printf("%d élèves ont moins de cette moyenne",nbr);
}
```

Exercice 1 :

Soit un tableau T ayant N éléments. Transférer les éléments positifs de T dans un tableau TPositif et les éléments négatifs ou nuls de T dans un tableau TNegatif. Afficher les tableaux TPositif et Tnegatif.

2.5 Recherche séquentielle

Soit une donnée introduite dans une variable que l'on appellera **cible**, il est courant de devoir rechercher la présence et éventuellement la position de cette donnée dans un tableau non trié. Dans ce cas la recherche doit se faire de manière séquentielle, c'est à dire en comparant la cible avec les données successives.

Nous présentons ici, deux méthodes de recherche séquentielle :

1/

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

2/

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

Voici le programme qui réalise la recherche séquentielle avec la 1ère méthode :

```
#define Nmax 100

void main()
{
      int tab[Nmax] , n, i, cible;

      /* Saisie des données */
      do
      {
        printf("Donnez le nombre d'éléments (max. %d) : ",Nmax);
        scanf("%d",&n);
      }while(n<=0 || n>Nmax);

      for(i=0; i<n; i++)
      {
            printf("tab[%d] = ",i);
            scanf("%d",&tab[i]);
      }

      printf("Donnez la valeur à rechercher : ");
      scanf("%d",&cible);

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

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

}
```

Les méthodes de tri :

On a très souvent besoin de traiter des données en ordre croissant (de la plus petite à la plus grande) ou en ordre décroissant (de la plus grande à la plus petite).

1. Tri par extraction (selection)

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

Programme :

```
#define Nmax 100

void main()
{
      int tab[Nmax], n, i, j, temp;

      do
      {
        printf("Donnez le nombre d'éléments (max. %d) : ",Nmax);
        scanf("%d",&n);
      }while(n<=0 || n>Nmax);

      for(i=0; i<n; i++)
      {
            printf("Tab[%d] = ",i);
            scanf("%d",&tab[i]);
      }
      for(i=0;i<n-1;i++)
            for(j=i+1;j<n;j++)
                  if(tab[i]>tab[j])
                  {
                        temp = tab[i];
                        tab[i] = tab[j];
                        tab[j] = temp;
                  }
      printf("Tableau Trié par ordre croissant : \n");
      for(i=0;i<n;i++)
            printf("Tab[%d] = %d\n",i,tab[i]);
}
```

2. Tri par insertion :

2.1. Le principe :

Consiste à parcourir la liste , On prend deux éléments qu'on trie dans le bon ordre, puis un 3e qu'on insère à sa place parmi les 2 autres, puis un 4e qu'on insère à sa place parmi les 3 autres, etc.

2.2. Le Programme :

```
#include<stdio.h>
#include<conio.h>

 main()
{  /* Declaration des variables*/
    int i,j,k,n;
    int aux,v[i];
  /* Saisie des donnes*/
     printf(" donner la valeur de n:\t "); scanf(" %d",&n);
   for ( i=1 ; i<=n ; i++ )
     {
      printf(" v[%d]:\t ",i); scanf(" %f",&v[i]);
     }
   /* Traitement*/
   for ( i=2 ; i<=n ; i++ )
     { aux=v[i];
     for ( j=1 ; j<=i-1 ; j++ )
       {   if (aux> v[j])
            {  for ( k=i ; k<=j+1 ; k-- )
               {
                v[k] = v[k-1];
                }
             v[j]=aux;
            }
       }
     }
     /* Affichage du  du résultat */
    printf("Tableau trié :\n");
    for (i=1; i<=n; i++)
     {    printf("%d ", v[i]);
          printf("\n");
     getch();
}
```

3. Tri a bull :

3.1. Le principe :

Ce tri permet de faire remonter petit à petit un élément trop grand vers la fin du tableau en comparant les éléments deux a deux. Si un élément d'indice i est supérieur à un élément d'indice i+1 on les échange et on continue avec le suivant. Lorsqu' on atteint la fin du tableau on repart du début. On s'arrête lorsqu'on n'aura aucun changement d'éléments du tableau.

Commentaires:

Plus le nombre d'éléments à trier est grand, plus c'est lent car il y a beaucoup de permutations à effectuer.

3.2. Le programme

```
#include<stdio.h>
#include<conio.h>
int main()
 {
 int t[10];
 int i,aux,j,n;
 printf("donner la taille du tableau:");
 scanf("%d",&n);
 printf("entrer les elements dans le tableau:");
 for(i=1;i<n;i++)
   {
    for( j=n;j>i;j--)
{
     if(t[j-1]>t[j])
     { aux=t[j-1];
     t[j-1]=t[j];
      t[j]=aux;
      }
      }
 for(i=1;i<=n+1;i++)
 printf("le tableau est:v[%d]=%d\n",i,v[i]);
 getch();
 return 0;
 }
```

4. Tri rapide :

4.1.Le principe :

On choisit un élément particulier de la liste de la liste de clés, appelé pivot, il construit ensuite deux sous-liste gauche et droite contenant respectivement les clés inferieures et supérieures au pivot. Ainsi pour tirer au sous tableau A [p,…, r] du tableau initial A [1,…, n] on retrouve les trois phrase suivantes :

+ A [p,…, q-1] contient les clés inferieure a A[q].
+ A[q] le pivot.
+ A [q+1,...., r] contient les clés superieures a A[q].

4.2.Le programme:

```
#include<stdio.h>
#include<conio.h>
int main()
 {
 int t[10];
 int i,gauche,droite,x;
 printf("donner la taille du tableau:");
 scanf("%d",&n);
 printf("entrer les elements dans le tableau:");
 i=gauche;j= droite; pivot=t[i+j/2]
   while(i<=j)
    { while(t[i]<pivot); i++;
       while ( t[j]>pivot);j--;
     if(i<=j)
     { x=t[i];
     t[i]=t[j];
      t[j]=x;
      i++;j--;
      }if (gauche<j)tri rapide(t,x,gauche);
      if( droite>i)tri rapide(t,i,droite);
      }
 for(i=1;i<=n+1;i++)
 printf("le tableau est:v[%d]=%d\n",i,v[i]);
 getch();
 return 0;
 }
```

2.7 Affectations globales de tableaux

Pour deux tableaux tab1 et tab2 de même taille, on ne peut pas copier l'un dans l'autre avec tab1=tab2. Il faut faire une boucle et faire la copie élément par élément :

```
int tab1[Nmax], tab2[Nmax];
…
for(i=0; i<Nmax; i++)
    tab1[i] = tab2[i] ;
```

3.Tableau à deux dimensions

3.1 Déclaration

Un tableau à deux dimensions, ou matrice, se déclare selon le schéma suivant :

type nomTableau[nbLignes][nbColonnes];

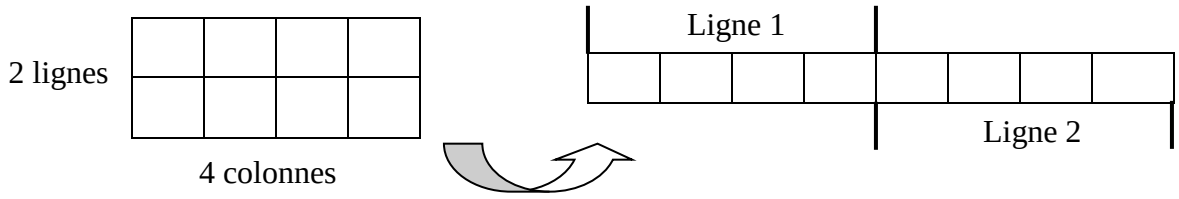
Cette déclaration réserve en mémoire : nbLignes * nbColonnes * sizeof(type) octets.

3.2 Représentation mémoire

La mémoire de l'ordinateur est une succession d'adresses. C'est donc une structure à une seule dimension. La représentation d'un tableau à deux dimensions dans une structure à une dimension est obtenue en stockant les lignes les unes à la suite des autres.

Exemple :

int mat[2][4] : 2 lignes de 4 colonnes



Représentation mémoire

3.3 Initialisation d'une matrice

Une matrice peut être initialisée à sa création en fournissant la liste des éléments ligne par ligne.

Exemples :

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

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

1	2	3	4
5	6	7	8

MatA

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

1	2	3	4
5	6	7	8

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

1	0	0
0	1	0
0	0	1

MatB

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

MatC

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

1	2	3
4	0	0
0	0	0

MatD

3.4 Accès aux éléments d'une matrice

L'accès se fait en utilisant la notation `Mat[i][j]`, `i` désignent le numéro de ligne, `j` le numéro de colonne.

Exemple :

Mat[0][0] = 1 ;
Mat[0][1] = 2 ;
Mat[1][0] = 3 ;
Mat[1][1] = 4 ;

1	2
3	4

Mat

3.5 Parcours des matrices

Les parcours des matrices se font avec des boucles imbriquées : une boucle externe pour parcourir les lignes, pour des indices compris entre 0 et LIGMAX-1, et une boucle interne, qui pour chaque ligne, fait le parcours des éléments dans toutes les colonnes de cette ligne. Les indices des colonnes seront alors compris entre 0 et COLMAX-1.

Exemple 1 : *Remplissage d'une matrice par l'utilisateur.*

```
#define LIGMAX 10
#define COLMAX 10

void main()
{
      int Mat[LIGMAX][COLMAX],i,j,l,c;

      do
      {
            printf("Nombre de lignes : ");
            scanf("%d",&l);
      }while(l<1 || l>LIGMAX);



      do
      {
            printf("Nombre de colonnes : ");
            scanf("%d",&c);
      }while(c<1 || c>COLMAX);
/********************** Lecture ****************/
      for (i=0;i<l;i++)
            for (j=0;j<c;j++)
            {
                  printf("Mat[%d][%d] = ",i,j);
                  scanf("%d",&Mat[i][j]);
            }
/********************** Affichage ***************/
      for (i=0;i<l;i++)
      {
            for (j=0;j<c;j++)
                  printf("%d\t",Mat[i][j]);
            printf("\n");
      }
}
```

Exemple 2 : Gestion de notes.

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

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

```
#define LIGMAX 10
#define COLMAX 10

void main()
{
        float notes[LIGMAX][COLMAX],som;
        int i,j,l,c;

        do
        {
                printf("Nombre d'élèves : ");
                scanf("%d",&l);
        }while(l<1|| l>LIGMAX);
        do
        {
                printf("Nombre de matières : ");
                scanf("%d",&c);
        }while(c<1 || c>COLMAX);
        /********************** Lecture *****************/
        for (i=0;i<l;i++)
        {
                printf("\nNotes pour l'élève %d = \n",i+1);
                for (j=0;j<c;j++)
                {
                        printf("\tNote de la matière %d = ",j+1);
                        scanf("%f",&notes[i][j]);
                }
        }
        /******* Calcul de la moyenne de chaque élève ***********/
        for (i=0;i<l;i++)
        {
                for (som=0,j=0;j<c;j++)
                        som += notes[i][j];
                printf("\nLa moyenne de l'élève %d : %.2f", i+1, som/c);
        }
        /******* Calcul de la moyenne par matière ***********/
        for (j=0;j<c;j++)
        {
                for (som=0,i=0;i<l;i++)
                        som += notes[i][j];
                printf("\nLa moyenne de la matière %d : %.2f", j+1, som/l);
        }

}
```

3.6 Exercice d'application

Ecrire un programme qui permet d'initialiser la matrice B par la transposée de la matrice A.

Exemple 1 :

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

Exemple 2 :

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Chapitre 6. Les chaînes de caractères.

1. Généralités.

Il n'y a pas de type chaîne de caractères en C. Une chaîne est représentée par un tableau de caractères contenant un caractère marqueur de fin de chaîne, '\0'. Toute fonction standard manipulant une chaîne devra connaître l'adresse de début de la chaîne, c'est-à-dire l'adresse de son premier caractère, et terminera son travail à la rencontre du caractère de fin '\0'.

2. Déclarations et initialisations de chaînes.

2.1. Tableau de caractères (dimensionné ou non).

```
char texte1[10]={'e','x','e','m','p','l','e','\0'} ;
```

On réserve 10 caractères et on initialise comme un tableau, en n'omettant pas le caractère '\0'.

```
char texte2[10]= "exemple";
```

Nous reviendrons plus tard sur le travail de la notation `""`. C alloue de la mémoire pour représenter la chaîne avec son `'\0'`, puis recopie cette zone mémoire dans la zone réservée pour le tableau `texte2` . 10 octets ont été réservés, alors que 8 suffisaient.

```
char texte3[]= "exemple";
```

Dans ce cas, 8 octets ont été réservés, c'est-à-dire exactement la taille utile.

2.2. Pointeur de caractères.

```
char * texte4= "exemple";
```

Précisons maintenant le travail effectué par la notation `""`:

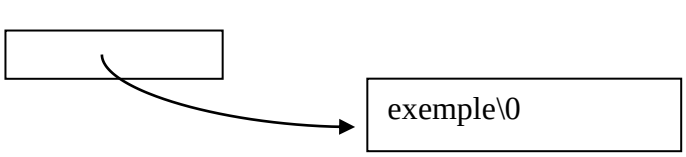
allocation de N+1 octets si la chaîne a une taille de N caractères.

Copie des N caractères dans la zone mémoire réservée.

Ajout de '\0' à la fin

L'adresse de la zone mémoire allouée est renvoyée

`texte4` est un pointeur de caractères pointant vers cette zone mémoire allouée par `""`.



Attention, il y a une différence avec la déclaration de `texte3[]`, on pourra faire, par la suite, `texte4= "salut"` mais on ne pourra pas faire `texte3= "salut"` , car un tableau est analogue à un pointeur constant (la dernière affectation reviendrait à modifier la valeur de l'adresse associée à `texte3`.

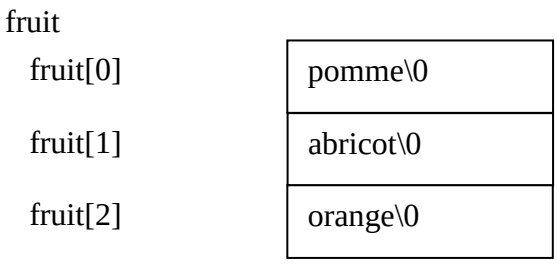
On portera également attention aux différences d'impression dans les exemples suivants :

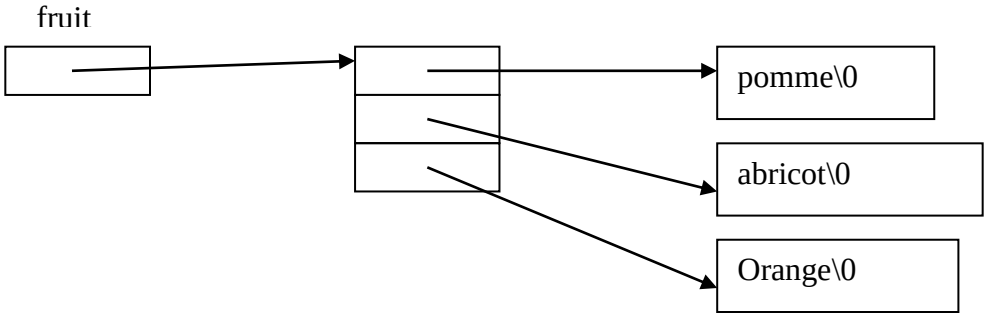
```
printf("%c",* texte4) ;/*affiche :  e */
printf("%s", texte4) ;/*affiche :  exemple */
```

3. Tableau de chaînes de caractères.

On peut déclarer soit un tableau de caractères à 2 dimensions, soit un tableau de pointeurs de caractères à 1 dimension .

```
char fruit[3][8]= {"pomme","abricot","orange"};
char  *fruit[3]= {"pomme","abricot","orange"};
```





4. Saisie de chaîne.

```
scanf({"%s",<adresse début de chaine>)
```

Effectue la saisie d'une chaîne de caractères tapés au clavier, jusqu'à la rencontre du premier espace ou '\n'.

Attention

```
char * nom ;
scanf("%s",nom) ;
```

ne produira pas d'erreur de compilation, mais les caractères lus vont être installés n'importe où en mémoire, puisqu'aucune place mémoire n'a été réservée.

```
gets(<adresse début de chaine>)
```

Effectue la saisie d'une chaîne de caractères tapés au clavier, jusqu'à la rencontre du premier '\n'. L''\0' est rajouté automatiquement.

`gets` renvoie l'adresse de début de chaîne ou `NULL` si on est en fin de fichier.

5. Impression de chaîne.

```
printf({"%s",<adresse début de chaine>)
puts(<adresse début de chaine>)
```

6. Fonctions de manipulation de chaîne.

6.1. Copie de chaînes.

`strcpy( <adresse chaine1>, <adresse chaine2>)`

Recopie le contenu de la mémoire depuis `<adresse chaine2>` jusqu'à la rencontre d'un '\0', à partir de `<adresse chaine1>`. Attention, le compilateur n'effectue aucune vérification quant à la réservation d'un espace mémoire suffisant pour stocker chaîne1.

6.2. Concaténation de chaînes.

`strcat(<adresse chaine1>, <adresse chaine2>)`

concatène le contenu de la mémoire depuis `<adresse chaine2>` jusqu'à la rencontre d'un '\0', au contenu de la mémoire à partir de `<adresse chaine1>`. Attention, le compilateur n'effectue aucune vérification quant à la réservation d'un espace mémoire suffisant pour stocker le résultat de la concaténation de chaîne1 et chaîne2.

6.3. Comparaison de chaînes.

`strcmp(<adresse chaine1>, <adresse chaine2>)`

compare le contenu de la mémoire depuis `<adresse chaine1>` jusqu'à la rencontre d'un '\0', au contenu de la mémoire à partir de `<adresse chaine2>` jusqu'à la rencontre d'un '\0'. La valeur retournée est 0 si les chaînes sont égales, une valeur négative si chaîne1 précède chaîne2 au sens lexicographique, ou une valeur positive si chaîne1 suit chaîne2 au sens lexicographique.

6.4. Longueur de chaîne.

`strlen(<adresse chaine1>)`

La valeur retournée est la longueur de la chaine sans compter le caractère '\0'.

Chapitre 7. Les pointeurs

1. Rappels

1.1 Notion d'adresse

Chaque donnée stockée en mémoire a un emplacement unique, représenté par son *adresse*. Les adresses sont ordonnées linéairement : on peut représenter la mémoire comme un tableau à une dimension (une suite de cases) dont les cases contiennent les données et où les adresses sont les indices. Une donnée peut elle-même occuper plusieurs cases. Son adresse est alors celle de la première case (et le nombre de cases occupées se déduit de son type).

Pour les déclarations de variables suivantes :

```
int  val1 = 2000 ;
long int val2 = 2000000;
```

Ceci est stocké en mémoire :

Nom	Adresse en hexa	Valeur codée en hexa
val1	(9027 0FFE)	07 | D0
val2	(9027 0FFA)	00 | 1E | 84 | 80

- val1 est codé sur 2 octets et son adresse est 9027 0FFE
- val2 est codé sur 4 octets et son adresse est 9027 0FFA

1.2. Opérateur d'adresse : &

En C, l'opérateur & permet de connaître l'adresse d'une variable en mémoire. On place le symbole & devant le nom de la variable.

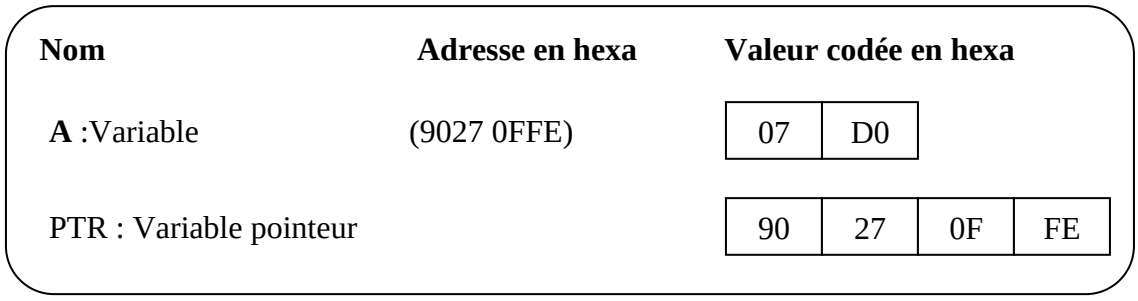
Syntaxe : &<NomVariable>

Exemple :

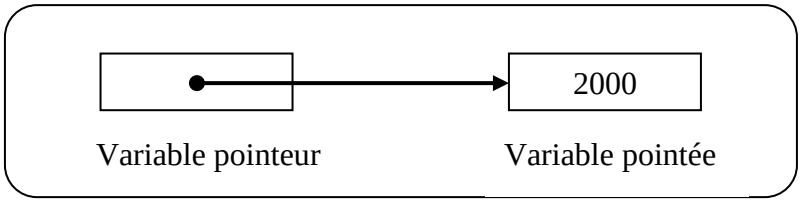
```
printf ("Adresse de val1 : %lx", &val1);  /* 90270FFE  */
printf ("Adresse de val2 : %p", &val2);   /* 9027:0FFA */
```

2. Notion de pointeur

Un *pointeur* est une variable contenant l'adresse d'une autre variable d'un type donné. Il permet donc d'accéder **indirectement** à une variable.



Le schéma ci-dessus montre que le pointeur PTR contient l'adresse de la variable A, on dit que « **PTR pointe sur A** ». Ici, *pointer* signifie « faire référence à ». La valeur d'un pointeur peut changer : cela ne signifie pas que la variable pointée est déplacée en mémoire, mais plutôt que le pointeur pointe sur autre chose.



3. Déclaration d'une variable pointeur

On déclare un pointeur comme on le ferait pour une variable, mais on rajoute * (étoile) devant son identificateur lors de la déclaration :

Syntaxe : type *pointeur [= valeur_initiale];

- Le type utilisé doit être celui qui sera pointé par le pointeur.
- Le type de pointeur est « pointeur de type » et se note « type * ».

Exemple :

```
int *ptr_i;  /* ptr_i est un pointeur sur des entiers , c'est-à-dire il ne
                doit contenir que des adresses des variables du type int */

char *ptr_c; /* ptr_c pointeur sur des caractères*/
```

Note : Pour déclarer plusieurs pointeurs, il faut donc répéter l'étoile devant chaque nom de pointeur.

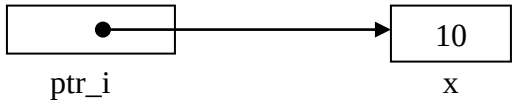
4. Initialisation d'un pointeur

Après avoir déclaré un pointeur il faut l'initialiser. En effet, si on ne prend pas cette précaution, le pointeur est susceptible de pointer vers une zone arbitraire de la mémoire, et accéder à cette zone risque de produire une erreur (détectée par le système d'exploitation).

Pour initialiser un pointeur, la syntaxe est la suivante :
Nom_du_pointeur = **&**nom_de_la_variable_pointee;

Exemple :

```
int x = 10, *ptr_i;

ptr_i = &x;   /* affecte l'adresse de la variable x au pointeur ptr_i */

printf("Adresse de x : %p\n",&x) ;
printf("Valeur de ptr_i : %p\n",ptr_i);
```



Remarque : Une valeur usuelle pour l'initialisation des pointeurs est la valeur **NULL** (équivalente à 0), qui représente un pointeur ne «pointant vers rien».

5. Accéder à une variable pointée

Pour accéder à la valeur pointée par un pointeur, nous utiliserons l'opérateur d'indirection « * » *(= contenu de*) devant le nom du pointeur. En utilisant la syntaxe *pointeur, nous avons

accès au contenu de la variable, aussi bien en lecture qu'en écriture, exactement comme si nous avions travaillé directement sur la variable pointée.

Exemple :

```
int *ptr,
int i;

i = 10;
ptr = &i;

printf("la valeur de i avant : %d\n",i);/*affiche :la valeur de i avant : 10*/

*ptr = 20; /*la variable pointée par ptr reçoit 20, autrement dit *ptr
             désigne le contenu de i*/

printf("la valeur de i après:%d\n",i);/*affiche : la valeur de i après : 20*/
```

Exercice 1 :

Qu'affiche le programme suivant :

```
void main()
{
      int A = 10 , B = 20 , C , D;
      int *ptr1=NULL, *ptr2=NULL ;

      ptr1 = &A;
      ptr2 = &B;
      C = *ptr1 + *ptr2;
      ptr1 = &C;
      ++*ptr1;
      D = *ptr1;
      *ptr1 = *ptr2;
      *ptr2 = D;

      printf("A = %d \nB = %d \nC = %d \nD = %d", A, B, C, D);
}
```

6. Arithmétique de pointeurs

On appelle arithmétique des pointeurs la possibilité de faire des opérations arithmétiques (incrémentation, décrémentation, addition et soustraction, etc.) sur les pointeurs. Cela revient à effectuer un déplacement en mémoire.

6.1 Incrémentation / Décrémentation d'un pointeur

- Soit **ptr** un pointeur de type **T.**

- Soit **a** la valeur de **ptr** (adresse contenue dans ptr).
- Si on exécute **ptr++**, alors la nouvelle valeur de **ptr** sera égale à : **a + sizeof(T).**

Exemple 1:

```
int i = 10 ;
int *ptr = &i;

printf("Valeur de ptr avant incrémentation : %p\n",ptr); /* 8E18 : FFF4 */
ptr++ ;
printf("Valeur de ptr après incrémentation : %p\n",ptr); /* 8E18 : FFF6 */
```

- Le même principe s'applique que **ptr--.**

6.2 Addition / Soustraction d'un entier

- Soit **ptr** un pointeur de type **T.**
- Soit **a** la valeur de **ptr** (adresse contenue dans ptr).
- Soit **i** un entier.
- Si on exécute **ptr+i** alors la nouvelle valeur sera égale à : **a + i*sizeof(T).**

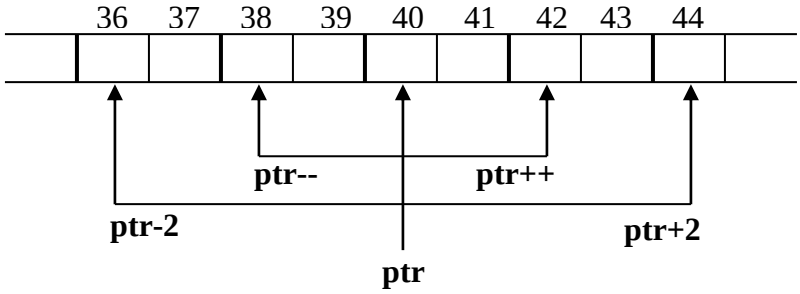
Exemple 2:

```
int i = 10 ;
int *ptr = &i;

printf("Valeur de ptr avant l'addition : %p\n",ptr); /* 8E18 : FFF4 */
ptr = ptr + 2 ;
printf("Valeur de ptr après l'addition : %p\n",ptr); /* 8E18 : FFF8 */
```

- Le même principe s'applique pour la soustraction d'un entier**.**

Exemple récapitulatif

Soit ptr un pointeur d'entier.



6.3 Soustraction de deux pointeurs

Si ptr1 et ptr2 sont deux pointeurs sur des objets de type T, l'expression *ptr1 – ptr2* désigne un entier dont la valeur est égale à *(ptr1 – ptr2)/sizeof(T).*

6.4 Comparaison de deux pointeurs

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

6.5 Accès aux éléments d'un tableau à une dimension par pointeur

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux. Ainsi, le programme suivant initialise un tableau avec la valeur 1 en employant le 'formalisme pointeur'.

```
#define Nmax 10
void main ()
{
        int tab[Nmax], i ;
        int *ptr;

        /* méthode 1 */
        for(ptr = tab, i =  0; i < Nmax; i++)
                *ptr++ = 1; /* ou bien {*ptr = 1; ptr++ ;}*/

        /* méthode 2 */
        for(ptr = tab, i = 0; i < Nmax; i++)
                *(ptr + i) = 1;

        /* méthode 3 */
        for(ptr = tab; ptr < tab + Nmax; ptr++)
                *ptr = 1;
}
```

Commentaires :

- Comme nous l'avons déjà constaté au chapitre 5, le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes **&tab[0]** et **tab** sont une seule et même adresse.
- ptr = tab ⇔ ptr = &tab[0].
- ptr + i : désigne l'adresse de l'élément d'indice i de tab.
- *(ptr + i) : désigne l'élément d'indice i de tab.

7. Relations entre tableaux à une dimension et pointeurs

7.1. Conversion des noms de tableaux à une dimension

Un nom de tableau apparaissant dans une expression est converti en (non pas équivalent à) un pointeur vers le premier élément du tableau.

Cas particulier : si tab est un nom de tableau, alors il n'est pas converti en pointeur

- si tab est utilisé comme opérande de sizeof (renvoie la taille du tableau en octets)
- si tab est utilisé comme opérande de l'opérateur &

7.2 L'opérateur d'indexation []

En C, l'opérateur d'indexation est en fait une abréviation : tab[i] est un raccourci d'écriture mis pour ***(tab + i),** donc tab[i] et *(tab + i) sont deux écritures strictement équivalentes sur le plan syntaxique.

En effet, dès que le compilateur rencontre une expression de la forme :

Exp1[Exp2]

où Exp1 et Exp2 sont deux expression tout à fait quelconques, il la remplace par :

*((Exp1) + (Exp2))

sans même chercher à savoir si Exp1 (ou Exp2) a un quelconque rapport avec un tableau.

Exemple

```
#define Nmax 10

void main ()
{
        int tab[Nmax],i ;

        for(i = 0; i < Nmax; i++)
                scanf("%d", tab + i) ;
        for(i = 0; i < Nmax; i++)
                printf("%d\t", *(tab + i));
}
```

Remarques :

- L'addition étant commutative, il est vrai que *(tab + i) == *(i + tab) ; on en déduit qu'il n'y a absolument aucune différence entre tab[i] et **i[tab].**
- Après les instructions :

```
int  tab[10], *ptr , i;
ptr = &tab[0] ;
```

- *(ptr + i) peut s'écrire **ptr[i]** d'après ce qu'on a vu précédemment.
- tab++ est invalide car tab n'est pas un objet.

Exercice 2 :

Soit ptr un pointeur qui « pointe » sur un tableau tab :

```
int tab[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
int *ptr = tab;
```

Supposons que ptr contient l'adresse **FF00**, quelles valeurs et/ou adresses fournissent ces expressions, sachant que chaque expression dépend de la précédente :

1. *ptr + 2
2. *(ptr + 2)
3. *ptr++
4. ptr - tab + 1
5. ++ptr + 3

Exercice 3 :

Ecrire un programme qui parcoure un tableau d'entiers et qui affiche les indices des éléments nuls du tableau, sans utiliser aucune variable de type entier.

Exercice 4 :

Ecrire un programme qui réalise les opérations suivantes :

1) La lecture d'un tableau.
2) Le tri par extraction.
3) L'affichage du tableau.

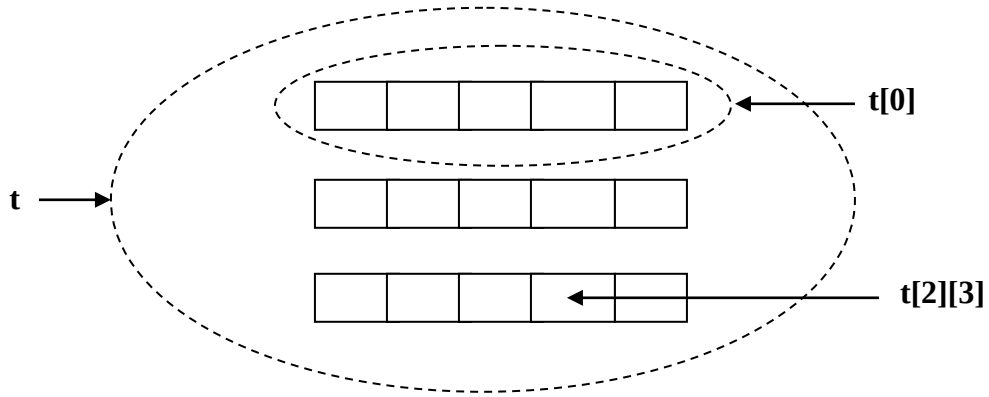
En utilisant la déclaration suivante : int tab[Nmax], *ptr1, *ptr2, n , temp;

8. Relations entre tableaux à deux dimensions et pointeurs

Nous avons parlé des relations existant entre les pointeurs et les tableaux à une dimension. Un nom de tableau sans les crochets est converti en un pointeur vers le premier élément du tableau, cela d'une part. D'autre part si tab est un tableau, alors tab[i] et *(tab + i) sont deux expressions équivalentes. *Que se passe-t-il lorsque les tableaux ont deux dimensions* ?

8.1 La taille des éléments d'un tableau à deux dimensions

Lorsque le compilateur rencontre une déclaration telle que: **int t[3][5];** il considère en fait que **t** désigne un tableau de 3 éléments, chacun de ces éléments étant lui même un tableau de 5 entiers. Comme il s'agit d'un tableau de tableaux, t[i] représente le nom d'un tableau de 5 entiers.



Le programme suivant va nous permettre de déterminer la taille des éléments d'un tableau à deux dimensions :

```
void main()
{
      int t[3][5];

      printf("La taille de t[0][0]  : %u\n", sizeof(t[0][0]) );
      printf("La taille de t[0]     : %u\n", sizeof(t[0]) );
      printf("La taille de t        : %u\n\n", sizeof(t) );
}
```

Affiche :

```
La taille de t[0][0]    : 2
La taille de t[0]       : 10
```

```
La taille de t          : 30
```

8.2 Conversion des noms de tableaux à deux dimensions :

Lorsque le compilateur rencontre une expression de la forme t[i][j] , elle la remplace par l'expression ***(*(t + i) + j)**.

Elle s'interprète de la façon suivante :

- **t** : est converti en un pointeur vers un tableau à 5 entiers ayant l'adresse de la première ligne de la matrice t ;
  Note : Comment on déclare un pointeur vers un tableau à 5 entiers ?
  Par l'instruction suivante : **int (*ptr)[5] ;**
  En effet, si ptr pointe vers le premier tableau de la matrice t, c'est à dire ptr = t, alors ptr + 1 pointe vers le deuxième tableau de cette matrice.

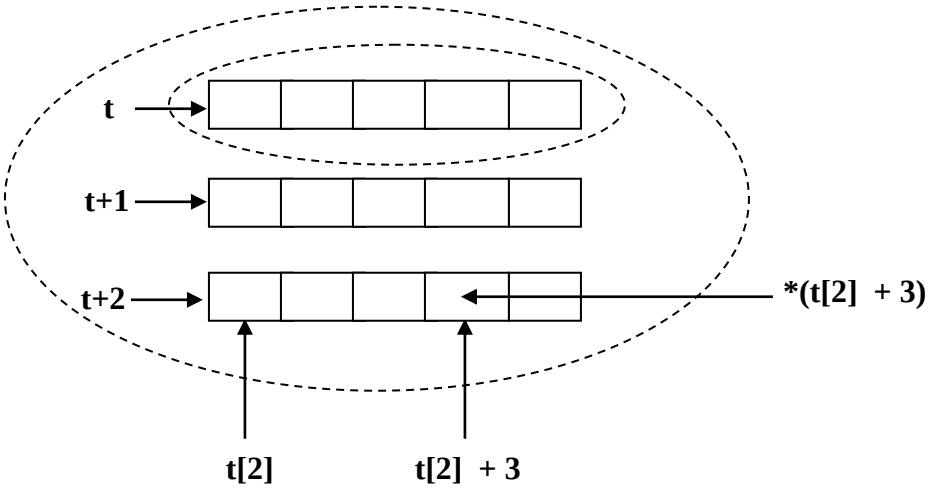
- **t + i** : désigne l'adresse de la ligne d'indice i de la matrice t.

- ***(t + i) ou t[i]** : étant le nom d'un tableau de 5 entiers, il est converti en un pointeur vers le premier élément de cette ligne, c'est à dire un pointeur de type int*.
  Par conséquent, les notations suivantes sont totalement équivalentes (elles correspond à la même adresse et elles sont de même type) :

  t[0] ⟺ &t[0][0]
  t[1] ⟺ &t[1][0]
  t[2] ⟺ &t[2][0]

- ***(t + i) + j ou t[i] + j** : désigne l'adresse de l'élément d'indice (i, j) de la matrice t.

- ***(*(t + i) + j) :** désigne l'élément d'indice (i, j) de la matrice t.



Le programme suivant affiche les adresses des éléments d'un tableau à deux dimensions :

```
void main()
{
      int t[3][5] ;

      printf("&t[0][0]  : %p\n", &t[0][0] );
      printf("t[0]      : %p\n", t[0] );
      printf("t         : %p\n\n", t );

      printf("&t[0][0] + 1    : %p\n", &t[0][0]+1 );
      printf("t[0] + 1        : %p\n", t[0]+1 );
      printf("t + 1           : %p\n\n", t+1 );
}
```

Affiche :

```
&t[0][0]    : 8E19 : FFD8
t[0]        : 8E19 : FFD8
t           : 8E19 : FFD8

&t[0][0] + 1: 8E19 : FFDA     /* FFD8 + 2 = FFDA */
t[0] + 1    : 8E19 : FFDA
t + 1       : 8E19 : FFE2     /* FFD8 + A = FFE2 */
```

Remarque : Les expressions t[i]++ et t++ sont invalide car elles ne sont pas des objets.

Exemple d'utilisation :

Le programme suivant permet de remplir et d'afficher un tableau à deux dimensions en employant le « formalisme pointeur ».

```
#define LIGMAX 10
#define COLMAX 10

void main ()
{
      int t[LIGMAX][COLMAX], i, j, l, c;

      do
      {
            printf("Nombre de lignes : ");
            scanf("%d",&l);
      }while(l<1|| l>LIGMAX);

      do
      {
            printf("Nombre de colonnes : ");
            scanf("%d",&c);
      }while(c<1 || c>COLMAX);



      for(i=0;i<l;i++)
            for(j=0;j<c;j++)

            {
                  printf("t[%d][%d]= ",i,j);
                  scanf("%d",*(t+i)+j);      /*scanf("%d", t[i] + j );*/
            }

      for(i=0;i<l;i++)
      {
            for(j=0;j<c;j++)
                  printf("%d\t",*(*(t+i)+j) );/*printf("%d\t",*(t[i]+j) );*/
            printf("\n");
      }
}
```

Chapitre 8. Les fonctions

1. Introduction

Une fonction est un sous-programme qui accomplit une tâche spécifique. Elle possède une structure analogue à celle de la fonction main.

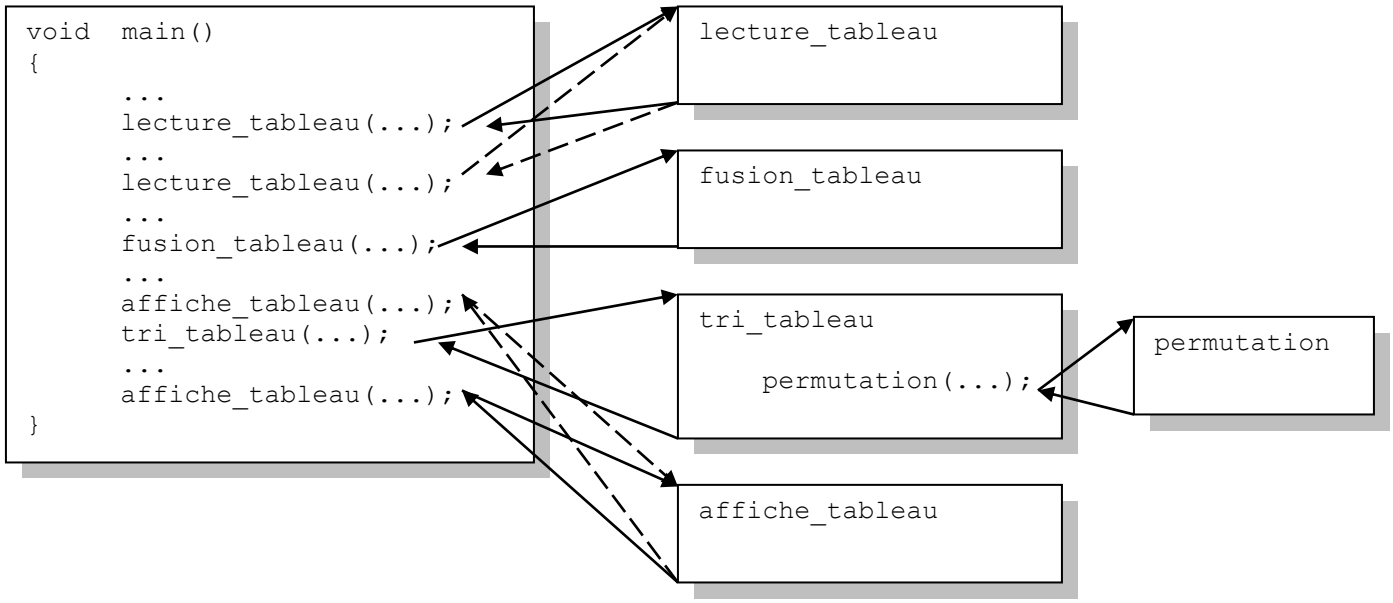
L'utilisation de fonctions présente plusieurs avantages :

- Rendre le programme plus facile à écrire et beaucoup plus facile à lire.
- Eviter d'écrire plusieurs fois la même séquence de code.
- La mise au point est plus simple car les différentes fonctions peuvent être testées séparément.
- Elles peuvent être réutilisées par d'autres programmes.

Exemple : Supposons qu'on veut lire deux tableaux, les fusionner dans un nouveau tableau puis le trier.

On utilisant les fonctions, le programme principal prend la structure suivante :

```
void main()
{
      /* déclaration des variables */
      /* ... */
      printf("Lecture du premier tableau : \n");
      lecture_tableau(...);
      printf("Lecture du deuxième tableau : \n");
      lecture_tableau(...);
      fusion_tableau(...);
      printf("Affichage du tableau avant son tri : ");
      affiche_tableau(...);
      tri_tableau(...);
      printf("Affichage du tableau après son tri : ");
      affiche_tableau(...);
}
```



2. Définition de fonctions

La définition d'une fonction peut être décrite de la façon suivante :

```
type nom_fonction ([type_1 param_1,type_2 param_2, ... ,type_n param_n])
{
      [<déclaration des variables internes à la fonction>];
      instructions;
      [return  expression];
}
```

La première ligne de cette définition est appelée l'en-tête de la fonction. Dans cet en-tête, les différents éléments sont les suivants :

- <**type**> : est le type de valeur qui sera renvoyée à la fin de l'exécution de la fonction ("char", "int", "float", "double", "int *", ...).
    - Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé void.
    - Si aucun type de donnée n'est précisé, le type int est pris par défaut.

- <**nom-fonction**> : est le nom que le programmeur donne à la fonction : les règles à respecter pour les noms de fonctions sont les mêmes que pour les noms de variables.

- <**param_1, param_2, …,param_n**> : se sont des variables qui permettent d'échanger des données avec la fonction appelante, appelées paramètres formels ou paramètres. Dans le cas où une fonction n'a pas de paramètres formels, on place le mot clé "void" entre les parenthèses.

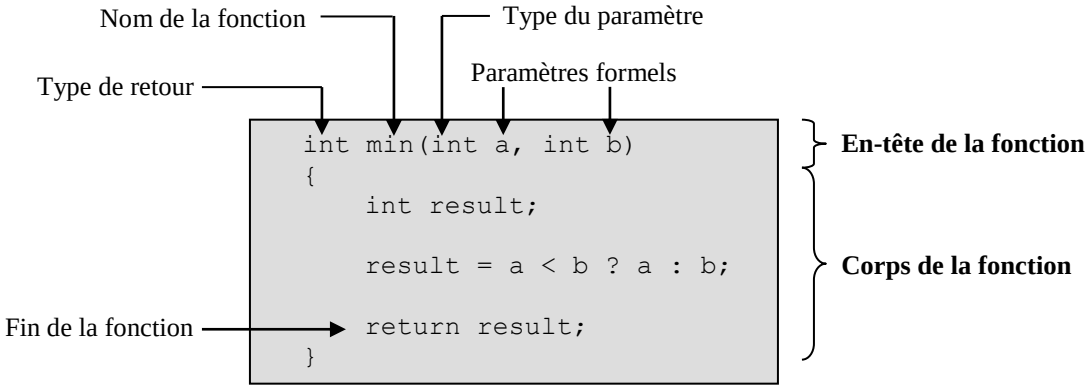
Le corps de la fonction se trouve entre accolades à la suite de l'en-tête. Il contient :

- La déclaration des variables utilisables uniquement dans le corps de la fonction.
- Les instructions de la fonction. Ces instructions peuvent utiliser les variables internes à la fonction et les paramètres formels.
- return expression : Cette instruction permet à une fonction de renvoyer une valeur à la fonction appelante.

Notes :

- expression doit être d'un type compatible avec le type de retour spécifié dans l'en-tête de la fonction.
- Si la fonction ne fournit aucune valeur alors le retour à lieu après la dernière instruction de la fonction (fonction de type void).
- Plusieurs instructions return peuvent apparaître dans une fonction. Le retour à la fonction appelante sera alors provoqué par le premier return rencontré lors de l'exécution.

Voici, par exemple, la définition d'une fonction qui calcule le minimum de deux entiers.



Remarque : Il est interdit de définir des fonctions à l'intérieur d'une autre fonction.

3. Appel d'une fonction

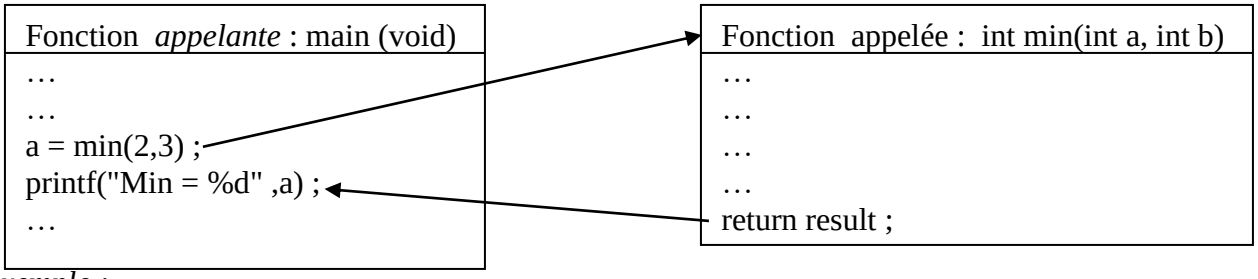
Une fonction est *invoquée* ou *appelée* en faisant suivre son nom d'une liste d'arguments séparés par des virgules et entourés entre parenthèses.

Syntaxe d'un appel :

nomFonction (arg_1, arg_2,…, arg_n);

arg_1, arg_2,…, arg_n : sont appelés arguments ou paramètres effectifs. Ils peuvent être des expressions, constantes ou variables.

A l'appel d'une fonction, la valeur du paramètre effectif est transmise au paramètre formel correspondant ensuite l'exécution du programme se poursuit par l'exécution de la première instruction du corps de la fonction. Au moment où l'exécution de la fonction est terminée, le contrôle est rendu à la fonction appelante c'est à dire à l'instruction qui suit l'appel à la fonction.



Exemple :

```
#include<stdio.h>

int min(int a, int b) /* définition de la fonction min */
{

      int result;

      result = a < b ? a : b;

      return result;
}

void main()
{
      int a,b,c,d ;
      scanf("%d%d%d",&a,&b,&c);
      d = min(a,b);                 /* 1er appel de la fonction min */
      printf("%d", min(d,c) ) ;     /* 2ème appel de la fonction min */
}
```

Notes :

Lors de l'appel à une fonction avec paramètres :

- il doit y avoir autant de paramètres effectifs que des paramètres formels.
- le type du paramètre effectif doit être compatible avec celui du paramètre formel associé.

4. Prototype de fonctions

Il est nécessaire que la fonction ait été définie ou déclarée avant tout appel à cette fonction.

La syntaxe pour la déclaration est :

type nom_fonction (type_1 param_1, type_2 param_2, ...,type_n param_n)**; /* en_tête_fct ;*/**

On parle de prototype de la fonction.

Exemples :

```
int min(int a, int b);              /* prototype de min */
double pow(doube x, double y);      /* prototype de pow */
```

Notez que les noms des paramètres sont facultatifs mais aident à la lisibilité.

Exemples :

```
int som(int , int);
double pow(doube , double);
```

A quoi sert-il ?

La déclaration d'une fonction permet au compilateur de vérifier que le nombre et le type des paramètres formels sont en accord avec le prototype. Si ce contrôle échoue, le compilateur signale une erreur.

De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

Notes :

- La définition de fonction peut aussi servir de prototype, si elle est définie avant son utilisation.
- Si la fonction n'est pas déclarée et elle est définie après son utilisation, alors le compilateur ne contrôle pas le nombre de paramètres, et considère que cette fonction renvoie un « int », cela d'une part, d'autre part, il utilise des règles de conversions systématiques sur les paramètres effectifs : char et short vers int et float vers double. Et par conséquent les paramètres formels ne pourront pas être de types float, char ou short.

⇨ **A éviter !**

5. Variables locales

Les variables déclarées dans les fonctions sont dites **locales**. Elles ne sont modifiables et accessibles que dans la fonction où elles sont déclarées.

D'autre part, par défaut, les variables locales sont temporaires. En effet, les emplacements mémoire correspondants sont alloués à chaque entrée dans la fonction où sont définies ces variables et ils sont libérés à chaque sortie.

Note : les paramètres formels d'une fonction sont traités de la même manière que les variables locales.

6. Variables globales

On appelle **variable globale** une variable déclarée en dehors de tout fonction**.** Elle est modifiable et accessible par toutes les fonctions qui suivent sa déclaration. Une variable globale est systématiquement permanente. Elle occupe un emplacement en mémoire qui reste durant toute l'exécution du programme.

Cet emplacement est alloué une fois pour toutes au moment de l'édition de liens. Elle est initialisée à zéro par le compilateur.

Exemple :

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

```
void affiche(void);           /*prototype de la fonction affiche*/
int x ;                       /* variable globale initialisé par zéro */

void affiche(void)            /*définition de la fonction affiche*/
{
      printf("x = %d\n",x);
      x++;
}

void main(void)
{
      int i ;
      for(i=0 ;i<3 ;i++)
            affiche();        /*appel de la fonction affiche*/
      printf("x = %d\n",x);
}
```

Affiche

x = ……
x = ……
x = ……
x = ……

L'utilisation des variables globales s'oppose au principe d'un programme clair, sûr et structuré. En effet une variable globale peut être modifiée dans n'importe quelle partie d'un programme et, de ce fait, la difficulté de la maintenance d'un programme croît exponentiellement en fonction de sa taille.

7. Conflit de noms entre variables globales et locales

Si un programme possède des variables globales, il peut parfaitement arriver qu'une variable locale s'avère porter le même nom. C'est le cas pour le programme suivant qui utilise trois variables globales (a, b et c) et dont la fonction *conflit_a* utilise une variable locale **a** ainsi que les variables globales b et c :

```
int a = 1, b = 2, c = 3 ; /* variables globales */

void conflit_a(void)
{
      int a = 100 ; /* variable locale */
      printf(" a = %d, b = %d, c = %d\n", a, b, c);
}

void main()
{
      conflit_a();
      printf(" a = %d, b = %d, c = %d\.n", a, b, c);
}
```

L'exécution du programme entraîne l'affichage suivant :

```
a = ……, b = ……, c = ……
a = ……, b = ……, c = ……
```

Lorsqu'un conflit de noms se produit entre variables globales et locales, le C utilise le nom de la variable locale.

8. Transmission des paramètres à une fonction

Lors de l'appel d'une fonction, la fonction reçoit une copie de la valeur de la variable passé comme paramètre effectif. Cette copie est affectée au paramètre formel correspondant. Cela implique en particulier que, les valeurs des paramètres formels peuvent être modifiées dans le corps de la fonction, mais sans influencer les valeurs des paramètres effectifs. On dit que les paramètres d'une fonction sont **transmis par valeurs**.

Par exemple, le programme suivant :

```
# include <stdio.h>

void echange (int , int);

void main (void)
{
      int a = 10 , b = 20;
      printf("Début du programme principal : \na=%d et b=%d\n" , a , b);
      echange(a , b);
      printf("Fin du programme principal : \na=%d et b=%d\n" , a , b);
}

void echange (int A , int B)
{
      int temp;
      printf("\tdébut fonction : \n\tA = %d et B = %d\n" , A , B);
      temp = A;
      A = B;
      B = temp;
      printf("\tfin fonction : \n\tA = %d et B = %d\n" , A, B);
}
```

Affiche

```
Début du programme principal :
a = 10 et b = 20
      début fonction :
```

```
      A = 10 et B = 20
      fin fonction :
      A = 20 et B = 10
Fin du programme principal :
a = 10 et b = 20
```

Cet exemple montre que la transmission de paramètres par valeur représente un mode d'échange d'information dans un seul sens.

Pour changer la valeur d'une variable de la fonction appelante, on procède comme suit:

- la fonction appelante doit fournir l'adresse de la variable;
- la fonction appelée doit déclarer le paramètre comme pointeur.

On peut alors atteindre la variable à l'aide du pointeur. On parle alors de **passage de paramètres par adresse.**

Par conséquent, pour échanger les valeurs de deux variables, il suffit de transmettre en paramètre leurs adresses :

```
#include<stdio.h>

void echange (int *, int *);

void main (void)
{
      int n = 10, p = 20;

      printf("avant l'appel: n=%d et p=%d\n" , n , p);
      echange(&n , &p);
      printf("après l'appel: n=%d et p=%d\n" , n , p);
}

void echange (int *a, int *b)
{
      int temp;

      printf("\tdébut echange: *a = %d et *b = %d\n" , *a , *b);

      temp = *a;
      *a = *b;
      *b = temp;

      printf("\tfin echange: *a = %d et *b = %d\n" , *a , *b);
}
```

9. Passage de tableau à une dimension en paramètre

Du fait de la conversion d'un identificateur de type tableau en adresse du premier élément, lorsqu'un tableau est passé en paramètre effectif, c'est cette adresse qui est passée en paramètre.

Le paramètre formel correspondant peut être déclaré de deux manières différentes mais équivalentes :

- **type *nom_tableau**
- **type nom_tableau[]**

Exemple :

```
#include<stdio.h>
#define Nmax 100

void affiche_tab(int t[], int n);   /*  void affiche_tab(int *t, int n); */
void lire_tab(int t[], int n);      /*  void lire_tab(int *t, int n); */

/*****************************************************************/
void affiche_tab(int t[], int n)   /*  void affiche_tab(int *t, int n) */
{
      int i;
      for (i = 0; i < n; i++)
                printf("%d\t", *(t + i) );
}
/*****************************************************************/
void lire_tab(int t[], int n)   /*  void lire_tab(int *t, int n) */
{
      int i;
      for (i = 0; i < n; i++)
      {
            printf("Elément %d = ", i+1);
            scanf("%d", t+i );      /* scanf("%d ",&t[i]); */
      }
}
/*****************************************************************/
void main(void)
{
      int tab[Nmax], n;
      printf("Donnez la nouvelle dimension du tableau : ") ;
      scanf("%d",&n);
      lire_tab(tab, n);
      affiche_tab(tab, n);
}
```

10. Passage de tableau à deux dimensions en paramètre

Lorsqu'un tableau à deux dimensions est passé en paramètre effectif, le paramètre formel correspondant peut être déclaré de deux manières différentes mais équivalentes :

- **`type nom_tableau[][Mmax]`**
- **`type (*nom_tableau)[Mmax]`**

Exemple:

```
#include<stdio.h>

#define Nmax 100
#define Mmax 100

void affiche_tab(int t[][Mmax],int n,int m);
void lire_tab(int t[][Mmax],int n,int m);

/*****************************************************************/

void affiche_tab(int t[][Mmax],int n,int m)
{
      int i,j;


      for(i=0; i<n; i++)
      {
            for (j=0; j<m; j++)
                  printf("%d\t", t[i][j]);
            printf("\n");
      }
}
/*****************************************************************/
void lire_tab(int t[][Mmax],int n,int m)
{
      int i,j;
      for (i=0; i<n; i++)
            for (j=0; j<m; j++)
            {
                  printf("Element[%d][%d] = ", i+1,j+1);
                  scanf("%d", &t[i][j]);
            }
}
/*****************************************************************/
void main(void)
{
      int tab[Nmax][Mmax], n, m;

      printf("Donnez le nombre de lignes du tableau : ");
      scanf("%d",&n);
      printf("Donnez le nombre de colonnes du tableau : ");
      scanf("%d",&m);

      lire_tab(tab, n, m);
      affiche_tab(tab, n, m);
}
```

Chapitre 9 Les structures et tableaux de structures

A) Le type structure :

1) Domaines d'utilisation :

Le type structure se trouve dans les structures de données suivantes :

- tableau de structures
- liste linéaire chaînée, arbre binaire
- fichier non texte
- classe (programmation orientée objet)
- etc ...

2) Introduction :

Le type structure (struct) en C est semblable au type enregistrement (record) dans le langage PASCAL
C'est une collection des champs qui peuvent être de types différents.

Exemple :

Une personne est une structure avec les champs possibles suivants :

- nom et prénom : chaîne de caractères
- taille et poids : réels
- âge : entier
- sexe : caractère
- etc ...

Grâce à ce type, on peut réduire le nombre de tableaux à traiter dans un programme :

1. Avec les tableaux à un seul indice, on a besoin des tableaux suivants :

   nomPre : tableau des chaînes de caractères
   numero : tableau des entiers (numéros d'inscription)

   intra, final, tp1, tp2, tp3, tps, globale : 7 tableaux des réels

   code : tableau des chaînes de caractères ("A+", "C-",...)

   Au total, on a 10 tableaux à déclarer, à écrire sur les en-têtes des fonctions et des appels. De plus, quand on fait l'échange dans le tri, il faut utiliser 3 affectations par tableau. Avec ces 10 tableaux, ces échanges nécessitent 10 x 3 = 30 affectations. C'est long à écrire.

2. Avec les tableaux à deux indices, on peut réduire sensiblement le nombre de tableaux à traiter :

```
nomPre   : tableau des chaînes de caractères
numero   : tableau des entiers (numéros d'inscription)
note     : un seul tableau à 2 indices
code     : tableau des chaînes de caractères ("A+", "C-",...)
```

Au total, on a 4 tableaux à déclarer, à manipuler sur les en-têtes des fonctions et des appels. Pour le tri, on a besoin de 12 (4 x 3) affectations afin d'échanger les informations.

3. Avec le type structure, il est possible d'utiliser un seul tableau : tableau des étudiants, chaque élément de ce tableau est une variable de type structure. On a ainsi un seul tableau à déclarer et à manipuler.

Plus tard dans le cours, on utilisera le type "struct" dans les listes linéaires chaînées, les arbres binaires, les fichiers non textes, ...

3) Déclaration et terminologie :

Supposons qu'on désire déclarer deux variables pers1 et pers2 de type structure et que chaque personne dispose de quatre informations :

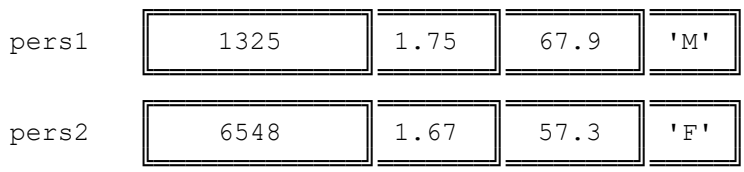
- numéro : un entier
- taille et poids : deux réels
- sexe : un caractère.

a) Façon 1 : déclaration directe sans le nom du type structure.

```
struct
    {
       int     numero ;
       float   taille, poids ;
       char    sexe ;
    }  pers1, pers2 ;
```

Le désavantage de cette manière de faire est qu'il n'est pas flexible pour la déclaration de nouvelles variables de même type structure que nous aimerions utiliser ailleurs dans un même programme.

Schéma d'une structure :



b) Façon 2: déclaration avec le nom du type structure.

b.1) on déclare d'abord le type :

```
struct personne
 {
        int     numero ;
        float   taille, poids ;
        char    sexe ;
  }  ;
```

b.2) on déclare après les variables :

```
struct personne pers1, pers2 ;
```

Si on veut déclarer d'autres informations de ce type, il suffit de se référer au type "struct personne" :

Exemples :

```
1)  void Afficher ( struct personne  unePers )
    /* afficher les informations d' "unePers" qui est
       de type struct personne */


2)  struct personne tempo ; /* une autre variable de
                               même type */
```

etc ...

c) Façon 3: déclarer en même temps le type et les variables:

```
struct personne
{
   int     numero ;
   float   taille, poids ;
   char    sexe ;

}  pers1, pers2 ;
```

d) Façon 4: déclarer en utilisant "typedef" :

```
typedef  struct {

        int     numero ;
        float   taille, poids ;
        char    sexe ;

}  Personne ; /* personne est le nom du type
*/

Personne pers1, pers2 ;
```

Remarques :

1. Pour ce cours, on vous suggère fortement d'utiliser la 4ème façon.

2. Avec la déclaration selon la 4ème façon :

   a) Personne est le nom du type structure qui dispose de 4 champs d'information.

   b) numero est le nom d'un champ de type entier.
      taille et poids sont deux champs de type réels.
      sexe est un champ de type caractère.

   c) pers1 et pers2 sont deux variables de type structure dont le nom est Personne.

3. Le C++ permet de simplifier les déclarations :

```
struct Personne
        {       int     numero ;
                float   taille, poids ;
                char    sexe ;
        };

Personne pers1, pers2 ;
```

Exemple :

Écrire les déclarations d'une variable "unEtud" de type "Etudiant" pour la gestion de ses notes.

Solution :

```
        #define LONG_NP   30   /* 30 caractères pour le nom et le prénom  */
        #define NB_NOTES   7   /*  7 notes */
        #define LONG_CODE  2   /*  2 caractères pour le code littéral "B+"
...*/

        typedef struct
                 {      char  nomPre [LONG_NP +1] ;
                        int   numIns              ;

                        float note[NB_NOTES]       ;

                        char  code[LONG_CODE+1]    ;
                 }
                Etudiant ;

        Etudiant unEtud ;
```

Exercice :

Écrire les déclarations d'une variable "unEmp" de type "Employe" qui comporte les champs d'informations suivantes :

- nom et prénom
- numéro d'employé
- numéro d'assurance sociale
- âge
- salaire hebdomadaire
- poste de travail (parmi les postes : analyste, programmeur, opérateur, secrétaire)

4) Manipulation du type structure :

4.1) Accès à un champ d'une structure :

le C utilise aussi le point "." pour accéder à un champ d'une variable de type structure :

variable_de_type_structure . champ

Exemples :

1. Écrire les instructions pour donner à la "pers1" les informations suivantes :

   C'est un homme qui mesure 1.67 mètre et pèse 67.8 kgs.
   Son numéro d'identification est le 7234.

```
pers1.numero = 7234 ;
pers1.sexe   = 'M'  ;
pers1.taille = 1.67 ;
pers1.poids  = 67.8 ;
```

2. Écrire les instructions pour afficher les informations de "pers1" à l'écran :

```
printf("Son numéro : %6d\n", pers1.numero);
printf("Son sexe   : %6c\n", pers1.sexe);
printf("Son poids  : %6.2f kgs\n", pers1.poids);
printf("Sa taille  : %6.2f m\n", pers1.taille);
```

Cas spécial : Pointeur vers le type structure :

Avec la déclaration : Personne * P ;

P est un pointeur vers le type Personne.

*P est une variable de type Personne.

On peut accéder à n'importe quel champ de *P :

(*P).taille, (*P).poids, etc ....

Le C permet de simplifier l'écriture en utilisant l'opérateur ->

(*P).champ <======> P->champ

4.2) Affectation entre deux variables de type structure :

Avec Personne pers1, pers2 ; on peut affecter l'une à l'autre :

```
pers1 = pers2 ; /* pers1 contiendra toutes les informations
                   de pers2.  */
```

Exemple :

Écrire une fonction permettant d'échanger les informations de deux personnes (de type Personne) :

Solution :

```
void echanger ( Personne * P1, Personne * P2)
{
   Personne Temporaire ;

   temporaire = *P1 ;
   *P1        = *P2 ;
   *P2        = temporaire ;
}
```

4.3) Comparaison entre deux variables de type structure :

La comparaison directe de deux informations de type structure

```
if ( pers1 == pers2 ) printf("C'est pareil") ;
```

n'est pas permise.

Cependant, le C permet de comparer octet par octet (comparaison en mémoire) :

```
#include <mem.h>  /* pour memcmp : memory comparison */

.......

if ( memcmp(&pers1, &pers2, sizeof(Personne)) == 0)
      printf("Elles sont identiques\n");
else
      printf("Elles sont différentes\n");
```

Exemple :

Écrire une fonction permettant d'afficher l'heure courante sous la forme, exemple : 18:25:17:78

Solution :

```
#include <dos.h>   /* pour la fonction gettime(...) dans
                      la fonction afficherHeure */
 ....

 void afficherHeure ()
 {
   /* struct time est un type "structure" prédéfinie dans
      le fichier d'en-tête <dos.h>

      struct time {
         unsigned char ti_min ;
         unsigned char ti_hour;
         unsigned char ti_hund;
         unsigned char ti_sec ;
      }

      où unsigned char : entier de 0 à 255
      ti_min : minute, ti_hour : heure, ti_hund : pourcent seconde,
      ti_sec : seconde.
   */

    struct time Temps ;

    gettime(&Temps);

    printf("%2d:%2d:%2d:%2d\n", Temps.ti_hour, Temps.ti_min,
                               Temps.ti_sec, Temps.ti_hund);
 }
```

Exercices :

Exercice 1 :

Écrire un programme permettant de :

- saisir les informations d'une personne (type Personne) en utilisant une fonction dont l'argument est transmis par pointeur : void saisir (Personne * P)
- appeler la fonction "saisir" 2 fois pour obtenir les informations de deux personnes différentes ;
- afficher les informations de ces deux personnes avec une fonction du genre :

  void afficher (Personne unePers)

- échanger (permuter) les informations de ces deux personnes en utilisant une fonction ;
- réafficher de nouveau les informations de ces deux

personnes après l'échange.

Notes :

La lecture suivante est parfaitement valide :

```
void saisir ( Personne * P )
{  ......
   printf("Entrez la taille ");
   scanf("%f", &(P->taille));
   ......
}
```

Le compilateur TURBO C réagit souvent mal à la lecture d'un réel avec l'opérateur ->. Si c'est le cas, il suffit de contourner le problème comme suit :

```
float t ;
......
printf("Entrez la taille ");
scanf("%f", &t) ;
P->taille = t ; /* c'est la même chose */
......
```

Exercice 2 :

Écrire un programme en mode interactif permettant de gérer les notes d'un seul étudiant (type structure) du cours IFT 1160. Les champs prévus sont :

```
nomPre  : une chaîne de 30 caractères
numIns  : un entier
note    : un tableau de 7 notes
code    : le code littéral "A-", "B+", ...
```

B) Tableau de structures :

1) Domaines d'utilisation :

Supposons qu'on a besoin de manipuler les informations "structurées" (des personnes, des étudiants, des employés, etc ...) dont les traitements prévus sont :

- le tri, la recherche d'un élément
- l'affichage
- les statistiques
- la création de nouveaux fichiers

Si le nombre de données est raisonnable (exemple : 500 à 600 personnes à traiter, 400 étudiants à calculer les notes , etc ...), on peut utiliser un tableau des structures.

2) Déclaration :

Écrire les déclarations d'un tableau de personnes. Chaque élément du tableau est de type structure nommé "personne" qui comporte les champs suivants :

```
                sexe            : 1 seul caractère
                taille et poids : 2 réels
```

On a 100 personnes ou moins à traiter.

Solution :

```
                #define  MAX_PERS  100

                typedef struct
                          {      char    sexe ;
                                 float   taille, poids ;
                          }
                        Personne ;

                 Personne pers[MAX_PERS] ;

                 int      nbPers        ; /* le nombre effectif de
personnes*/
```

3) Schéma d'un tableau des structures :

pers[0]	'M'	1.56	65.8
pers[1]	'F'	1.70	56.4
pers[2]	'M'	1.56	65.8

.....

4) Compréhension de base :

Avec ces déclarations :

1) pers est un tableau des personnes ;

2) pers[0], pers[1], ..., pers[99] sont 100 éléments du tableau pers. Chacun est une variable de type structure nommé "Personne" ;

3) pers[15].sexe est le sexe de la 16 ième personne ;

4) La déclaration : Personne * P ;

   rend valide l'affectation suivante :

```
            P = pers ;
```

   Cette affectation est équivalente à : P = &pers[0] ;

   De plus, *(P + i) est équivalent à pers[i].
   Ainsi :

```
             *(P + i). taille  est  pers[i].taille
```

   Donc, (P + i) -> taille est pers[i].taille

5) Exemples sur les tableaux des structures :

Exemple 1 :

On dispose du fichier texte "Metrique.Dta" :

```
ROY CHANTAL                  F      1.63     54.88
MOLAISON CLAUDE              M      1.57     56.25
BEDARD MARC-ANDRE            M      1.43     42.50
etc ....
```

Écrire un programme en C permettant de lire le fichier et de :

1. créer un tableau de personnes
2. afficher la liste des 10 premières personnes
3. compter et afficher le nombre de personnes dont :
   - la taille dépasse 1.80 mètre
   - le poids dépasse 56.78 kgs

Solution :

```
/* Fichier Struct1.A95: exemple sur le type "struct" en C

*/

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

#define MAX_PERS 25
#define LONG_NP  30

   typedef struct
          { char   nomPre[LONG_NP+1] ;
            char   sexe             ;
            float  taille, poids    ;
          }
           Personne ;

   Personne pers[MAX_PERS] ;

int      nbPers        ;

   void continuer()
   {
      printf("Appuyez sur Entrée ");
      fflush(stdin);
      getchar();
   }
```

```
 void lireCreer (Personne pers[], int * P)

{ FILE * donnees ;
  int    n = 0   ;
  float t, p ;

  donnees = fopen("Metrique.Dta", "r");

  while (!feof(donnees)) {

     /* On lit 30 caractères pour un nom et prénom et on fait
        déposer le caractère '\0' à la fin                    */
     fgets(pers[N].nomPre, LONG_NP+1, donnees);

     fscanf(donnees,"%c%f%f\n", &pers[N].sexe, &t, &p);
     pers[n].taille = t ;
     pers[n].poids  = p ;
     n++;
  }
  fclose(donnees);

  *P = n ;
}

int nombre (Personne pers[], int nbPers, int code, float borne)
 {
    int k = 0 , i ;

    for (i = 0 ; i < nbPers ; i++)

     if ( (code == 1 && pers[i].taille > borne) ||
          (code == 2 && pers[i].poids  > borne)     ) k++;

    return k ;
 }


 /* démonstration du pointeur vers le type structure */

 void Afficher ( Personne * P, int nombre )
 { int i ;

   printf("Liste des %d premières personnes :\n\n", nombre);

   for (i = 0 ; i < nombre ; i++)
    printf("%3d) %s %6.2f m %8.1f kgs   %s\n", i+1,
             (P+i)->nomPre, (P+i)->taille, (P+i)->poids,
               (P+i)->sexe == 'F' ? "Féminin" : "Masculin");

   Continuer();
 }


void main()
```

```
{
   lireCreer (Pers, &nbPers);
   afficher(Pers, nbPers);

   printf("\n\n");
   printf("Le nombre de personnes dont la taille dépasse"
            " 1.80 mètre :%3d\n",
             nombre(Pers, nbPers, 1, 1.80));
   printf("Le nombre de personnes dont le poids  dépasse"
            " 56.78 kgs  :%3d\n",
             nombre(Pers, nbPers, 2, 56.78));

   continuer();
}
```

Exécution :

```
Liste des 10 premières personnes :

  1) ROY CHANTAL                    1.63 m     54.9 kgs   Féminin
  2) MOLAISON CLAUDE                1.57 m     56.2 kgs   Masculin
  3) BEDARD MARC-ANDRE              1.43 m     42.5 kgs   Masculin
  4) MONAST STEPHANE                1.65 m     61.7 kgs   Masculin
  5) JALBERT LYNE                   1.63 m     47.6 kgs   Féminin
  6) DUBE FRANCOISE                 1.63 m     53.5 kgs   Féminin
  7) LABELLE LISE                   1.73 m     63.0 kgs   Féminin
  8) RIVERIN HELENE                 1.70 m     60.8 kgs   Féminin
  9) MICHAUD NORMAND                1.73 m     71.7 kgs   Masculin
 10) RICHER AGATHE                  1.65 m     53.1 kgs   Féminin

Appuyez sur Entrée

Le nombre de personnes dont la taille dépasse 1.80 mètre :  1
Le nombre de personnes dont le poids  dépasse 56.78 kgs  : 10

Appuyez sur Entrée
```

Exemple 2 :

Adapter le programme de l'exemple 1 afin qu'il permette aussi de

1. trier le tableau selon le nom et prénom

2. afficher la liste des 10 premières personnes avant et après le tri

3. chercher et afficher les informations d'une personne

4. créer un fichier texte nommé "Hommes.Gra" qui contient les informations des hommes dont la taille dépasse 1.75 mètre.

Solution :

```
/* Fichier Struct2.A95: Suite de Struct1.A95     */

#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAX_PERS 25
#define LONG_NP  30

typedef struct
          { char   nomPre[LONG_NP+1] ;
            char   sexe             ;
            float  taille, poids    ;
          }
        Personne ;

Personne pers[MAX_PERS] ;
int      nbPers        ;


   void continuer()
   {  printf("\n\nAppuyez sur Entrée ");
      fflush(stdin);
      getchar();
   }


   void lireCreer (Personne pers[], int * P)
   { FILE * donnees ;
     int    n = 0   ;
     float  taille, poids;

     donnees = fopen("Metrique.Dta", "r");

     while (!feof(donnees)) {

        fgets(Pers[n].nomPre, LONG_NP+1, donnees);

        fscanf(donnees,"%c%f%f\n", &Pers[n].sexe, &taille, &poids);

        pers[n].taille = taille ;
        pers[n].poids  = poids  ;

        n++;
     }

     fclose(donnees);

     *P = n ;
   }


   void afficher ( Personne * P, int nombre, char * quand )
   /* note :   (*P).champ <===> P->champ */
   { int i ;

     printf("Liste des %d premières personnes %s le tri:\n\n",
```

```
                nombre, quand);

   for (i = 0 ; i < nombre ; i++)
    printf("%3d) %s %6.2f m %8.1f kgs   %s\n", i+1,
             (P+i)->nomPre, (P+i)->taille, (P+i)->poids,
               (P+i)->sexe == 'F' ? "Féminin" : "Masculin");

   continuer();
 }


 /* Tri par sélection */
 void Trier (Personne pers[], int nbPers)
 {
    int i, j, indMin ; /* pour le tri par sélection */
    Personne  tempo  ; /* pour échanger 2 personnes */

    for (i = 0 ; i < nbPers-1 ; i++) {

      indMin = i ; /* par défaut */

      for ( j = i+1 ; j < nbPers ; j++ )

         if ( strcmp(pers[j].nomPre , pers[indMin].nomPre) < 0 )
                  indMin = j ;

      if (indMin != i)  {  /* faire des échanges*/

           tempo         = pers[i] ;
           pers[i]       = pers[indMin];
           pers[indMin] = tempo ;
      } /* fin de if (commentaire pédagogique) */
    } /* fin de for (commentaire pédagogique)  */
 }


void  creer(Personne pers[], int nbPers, char sexeVoulu,
               float borne, char * nomFichier)

{ FILE * aCreer = fopen(nomFichier, "w" ) ; /* w => for writing */

  for (int i = 0 ; /* cas spécial de C++, déclaration "flexible" */
         i < nbPers ; i++)

    if (pers[i].sexe == sexeVoulu && pers[i]. taille > borne)

       fprintf(aCreer,"%s %c %6.2f %7.1f\n", pers[i].nomPre,
                pers[i].sexe, pers[i].taille, pers[i].poids);

  fclose(aCreer);
}

 /* Recherche dichotomique dans un tableau trié  */

 void Chercher(Personne pers[], int nbPers)
```

```
{
   char aChercher[LONG_NP+1] ;
   int  mini, maxi, milieu, trouve ;
   int  compare, longueur ;

   #define VRAI 1
   #define FAUX 0

   do {
     printf("RECHERCHE DICHOTOMIQUE\n");

     printf("Quelques noms existants :\n");
     printf("Dube Francoise, Labelle Lise,  ... ");

     printf("\n\nEntrez le nom et prénom de la "
                  "personne recherchée ");
     gets(aChercher); /* get string = lire une chaîne */

     longueur = strlen(aChercher) ;

     mini = 0           ;
     maxi = nbPers - 1 ;
     trouve = FAUX ; /* on ne trouve pas encore */
```

```
                while (!trouve && mini <= maxi) {
                    milieu = (mini + maxi) / 2 ;
                    compare = strnicmp(aChercher, pers[milieu].nomPre,
longueur);
                    if ( compare < 0 )
                         maxi = milieu - 1 ;
                    else  if (compare > 0)
                                 mini = milieu + 1 ;
                          else trouve = VRAI ;
                 }
                if (!trouve)
                  printf("Désolé, on ne trouve pas %s\n", aChercher);

                else {
                   printf("Yahou!, voici ses informations :\n\n");
                   printf("Nom et prénom    : %s\n", pers[milieu].nomPre);
                   printf("taille           : %6.2f mètre\n",
                                              pers[milieu].taille);
                   printf("poids            : %6.2f kgs  \n",
                                              pers[milieu].poids);
                   printf("sexe             : %s\n",
                           pers[milieu].sexe == 'F' ? "Féminin": "Masculin");
                }
                printf("\n\n");
                printf("Avez-vous une autre personne à traiter ? (O/N) ");

                fflush(stdin);
               } while (toupper(getchar()) == 'O');
           }

       void main()
       {
          lireCreer (pers, &nbPers);

          afficher(pers, 10, "avant");

          trier (pers, nbPers) ;

          afficher(pers, 10, "après");

          chercher(pers, nbPers);

          /* Créer un fichier texte nommé "Hommes.Gra" qui ne
             contient que les informations des hommes dont
             la taille est supérieure à 1.75 mètre   */

          creer(Pers, nbPers, 'M', 1.75, "Hommes.Gra");

          printf("\n\nFin de la création du nouveau fichier Hommes.Gra\n");

          continuer();

       }
```

Exécution :

```
Liste des 10 premières personnes avant le tri:

  1) ROY CHANTAL                    1.63 m     54.9 kgs   Féminin
  2) MOLAISON CLAUDE                1.57 m     56.2 kgs   Masculin
  3) BEDARD MARC-ANDRE              1.43 m     42.5 kgs   Masculin
  4) MONAST STEPHANE                1.65 m     61.7 kgs   Masculin
  5) JALBERT LYNE                   1.63 m     47.6 kgs   Féminin
  6) DUBE FRANCOISE                 1.63 m     53.5 kgs   Féminin
  7) LABELLE LISE                   1.73 m     63.0 kgs   Féminin
  8) RIVERIN HELENE                 1.70 m     60.8 kgs   Féminin
  9) MICHAUD NORMAND                1.73 m     71.7 kgs   Masculin
 10) RICHER AGATHE                  1.65 m     53.1 kgs   Féminin

Appuyez sur Entrée

Liste des 10 premières personnes après le tri:

  1) BEDARD MARC-ANDRE              1.43 m     42.5 kgs   Masculin
  2) BEGIN MARIE-LUCE               1.60 m     49.0 kgs   Féminin
  3) DESMARAIS DENISE               1.75 m     64.0 kgs   Féminin
  4) DUBE FRANCOISE                 1.63 m     53.5 kgs   Féminin
  5) DUMITRU PIERRE                 1.80 m     79.4 kgs   Masculin
  6) FILLION ERIC                   1.78 m     75.8 kgs   Masculin
  7) JALBERT LYNE                   1.63 m     47.6 kgs   Féminin
  8) LABELLE LISE                   1.73 m     63.0 kgs   Féminin
  9) MICHAUD NORMAND                1.73 m     71.7 kgs   Masculin
 10) MOLAISON CLAUDE                1.57 m     56.2 kgs   Masculin


Appuyez sur Entrée

RECHERCHE DICHOTOMIQUE
Quelques noms existants :
Dube Francoise, Labelle Lise, Michaud Normand ...



Entrez le nom et prénom de la personne recherchée dube francoise

Yahou!, voici ses informations :

nom et prénom    : DUBE FRANCOISE
taille           :   1.63 mètre
poids            :  53.52 kgs
sexe             : Féminin

Avez-vous une autre personne à traiter ? (O/N) o

RECHERCHE DICHOTOMIQUE
Quelques noms existants :
Dube Francoise, Labelle Lise, Michaud Normand ...

Entrez le nom et prénom de la personne recherchée ali baba
Désolé, on ne trouve pas ali baba
```

```
Avez-vous une autre personne à traiter ? (O/N) N

Fin de la création du nouveau fichier Hommes.Gra


Appuyez sur Entrée
```

Contenu du fichier "Hommes.Gra" :

```
DUMITRU PIERRE                M   1.80    79.4
FILLION ERIC                  M   1.78    75.8
TREMBLAY SYLVAIN              M   1.83    86.2
```

Chapitre 10 : Les Fichiers C.

1. Généralités.

Les fichiers en C sont vus comme des **flots d'octets**, il n'existe aucune structuration logique ou physique en enregistrements comme c'est le cas en COBOL par exemple. Si l'on a besoin d'une telle structuration, cette dernière reste à la charge du programmeur.

Il existe 2 types de fichiers, les fichiers **texte** contenant des codes ASCII et des passages à la ligne, et les fichiers **binaires**.

Il existe 2 méthodes de manipulation des fichiers :

Une méthode assistée avec buffer

Une méthode non assistée de bas niveau.

Le fichier header stdio.h contient :

- Les fonctions d'Entrée/Sortie
- La constante EOF
- La constante NULL
- Le type FILE

Avant tout travail sur un fichier, il faut l'ouvrir dans un certain mode de travail. Quand les travaux sont terminés, il faut fermer le fichier.

2. Ouverture/fermeture de fichier.

a) Ouverture.

Syntaxe :

```
FILE * fopen(char * nom,char * mode)
```

- **`FILE`** est un type descripteur de fichier
- **`nom`** est une chaîne de caractères contenant le nom externe du fichier (nom système).
- **`mode`** indique le mode d'ouverture, sous forme d'une chaîne de caractères également :

" r " lecture seule (erreur, si le fichier n'existe pas)

" w " écriture seule(écrasement, si le fichier existe)

" a" écriture en fin de fichier

un **+** après le **r** ou **w** autorise la lecture.

Un **t** à la fin du mode indique un fichier texte, un **b** un fichier binaire.

Si l'ouverture échoue, fopen renvoie NULL, sinon un pointeur vers un descripteur de fichier. Toutes les autres instructions de manipulation de fichiers utilisent ce pointeur.

Exemple d'ouverture d'un fichier texte de nom « f_donnees », en lecture :

```
#include <stdio.h>

-----------
-----------
-----------
FILE *fd ;
if((fd=fopen("f_donnees", "rt"))==NULL)
     printf("\nerreur d'ouverture");
else /*  travailler sur le fichier */
     {-----------
      ------------
      ------------
     }
```

N.B.

Il existe 3 fichiers standards prédéfinis :

```
stdin
stdout
stderr
```

b) Fermeture.

```
int fclose(FILE * f)
```

Ferme le fichier de nom interne f, et renvoie 0 si OK.

3. Lecture/écriture de caractères.

a) Lecture de caractères.

```
int getc(FILE * f)
```

Retourne le caractère lu ou `EOF` (c'est-à-dire la valeur -1).

b) Ecriture de caractères.

```
void putc(int c,FILE * f)
```

`c` est le caractère à écrire.

Exemple :

Copie d'un fichier texte de nom « entree » dans un fichier texte de nom « sortie », caractère par caractère.

```
int c ;
FILE *fentree, *fsortie ;
fentree=fopen(" entree ", "rt") ;
fsortie=fopen("sortie" , "wt");
while ((c=getc(fentree))!=EOF)
          putc(c,fsortie);
```

4. Lecture/écriture de lignes.

a) Lecture de lignes.

```
char * fgets(char * tampon, int taille,FILE * f)
```

Lit les caractères sur le fichier, les charge dans le tableau de caractères **`tampon`**, ou bien jusqu'à ce qu'il ne reste qu'un seul caractère libre dans le tampon, et complète par '\0'.

Renvoie `NULL` en fin de fichier ou l'adresse du tableau `tampon` sinon.

b) Ecriture de lignes.

```
int fputs(char * tampon,FILE * f)
```

5. Lecture/écriture formatées.

a) Lecture formatée.

```
int fprintf(FILE * f, char * format,<liste de parametres>)
```

`format` est une chaine de caractères contenant des caractères à afficher tels quels et des spécifications de format, comme dans le `printf`. Le fonctionnement est analogue, sauf que l'écriture se fait dans le fichier f.

b) Ecriture formatée.

```
int fscanf(FILE * f, char * format,<liste de parametres>)
```

fonctionnement est analogue à celui de `scanf`.

6. Lecture/écriture de blocs.

a) Lecture de blocs.

```
int fread(char * tampon, int taille,int nombre,FILE * f)
```

Le `fread` tente de lire **`nombre`** objets de taille **`taille`**, les installe, octet par octet dans le tableau de caractères **`tampon`**, et renvoie le nombre d'objets effectivement transférés.

b) Ecriture de blocs.

```
int fwrite(char * tampon, int taille,int nombre,FILE * f)
```

Le `fwrite` tente d'écrire **`nombre`** objets de taille **`taille`**, octet par octet depuis le tableau de caractères **`tampon`**, vers le fichier **`f`** et renvoie le nombre d'objets effectivement transférés.

7. Instructions de contrôle.

a) Test de fin fichier.

```
int feof(FILE * f)
```

Renvoie 1 si fin de fichier, 0 sinon.

b) Erreur d'E/S.

```
int ferror(FILE * f)
```

Renvoie 0 si pas d'erreur, ≠0 sinon.

L'erreur concerne la dernière opération d'entrée/sortie effectuée.

c) Positionnement direct sur un octet.

A chaque fichier est associé un « point de déplacement » qui indique sur quel octet on est positionné.

A l'ouverture en lecture ou écriture, on est positionné sur le premier octet. En append on est positionné en fin de fichier.

En accès séquentiel, la gestion du « point de déplacement » est automatique

En accès direct, ou lorsqu'on mélange les lectures et les écritures, c'est au programmeur de se repositionner.

```
int fseek(FILE * f, long dep,int mode)
```

dep est un entier long indiquant un déplacement en octets à partir d'une certaine origine définie par l'entier mode, dont les valeurs possibles sont :

0 début du fichier

2 fin du fichier

1 position actuelle

Pour les valeurs 0 et 2, les déplacements sont positifs, alors que pour 1 le déplacement peut être signé.

N.B.

`fseek` renvoie 0 en cas de réussite, ≠ 0 en cas d'échec

d) Repositionnement en début de fichier.

```
void rewind(FILE * f)
```

équivaut à `fseek(f,0L,0)`

e) Position courante.

```
long ftell(FILE * f)
```

renvoie la valeur actuelle du « point de déplacement ».