

CHAPITRE 1 : LES ÉLÉMENTS DE BASE DU LANGAGE C

Introduction :

Un langage de programmation a pour finalité de communiquer avec la machine. Il y a diverses manières de communiquer avec la machine. Le langage « maternel » de la machine n'utilise que deux symboles (0 et 1) : c'est le langage machine. Par exemple le nombre 5 est reconnu par une machine par la succession des symboles 1,0,1 (c'est la représentation du nombre en base 2). De même, les opérations qu'une machine est capable d'exécuter sont codées par des nombres, c'est-à-dire une succession de 0 et 1. Par exemple, l'instruction machine

```
00011010 0001 00000010
```

demande à la machine d'effectuer l'opération $1 + 2$. A chaque type de machine correspond, un jeu d'instructions spécifiques; de même le codage des instructions est également dépendant de la machine utilisée.

Même si ce langage est le seul qui soit compris par l'ordinateur, il n'est pas le seul moyen de communiquer avec celui-ci. En effet, le besoin d'humaniser cette communication et la première tentative en ce sens est l'invention du langage assembleur. Par exemple, l'instruction assembleur

```
add $1 $2
```

demande à la machine d'effectuer l'opération $1 + 2$. Ce langage est très proche du langage machine et se contente de donner des noms mnémotechniques pour les instructions ainsi qu'une manière plus naturelle de désigner des entiers. Le langage assembleur fut suivi par des langages plus sophistiqués. En particulier, on distingue les

```
main()
{
    printf("Bonjour");
}
```

langages qui permettent la programmation structurée (fortran, pascal, algol, C, perl, tcl),

- la programmation structurée et modulaire (ada, modula, C, pascal),
- la programmation fonctionnelle (lisp)
- la programmation logique (prolog)
- la programmation objet (smalltalk, eifel, C++, java).

Le compilateur :

Ces langages, que nous avons cités, ont tous pour ambition de faciliter la programmation en la rendant plus proche du « langage humain ». Tous ces langages de programmation ont besoin d'un « traducteur » pour être compris par la machine. De tels traducteurs sont généralement appelés interpréteurs ou compilateurs.

ils traduisent le texte écrit par un programmeur en un programme exécutable contenant que des suites de 0 et 1 (compréhensible par la machine).

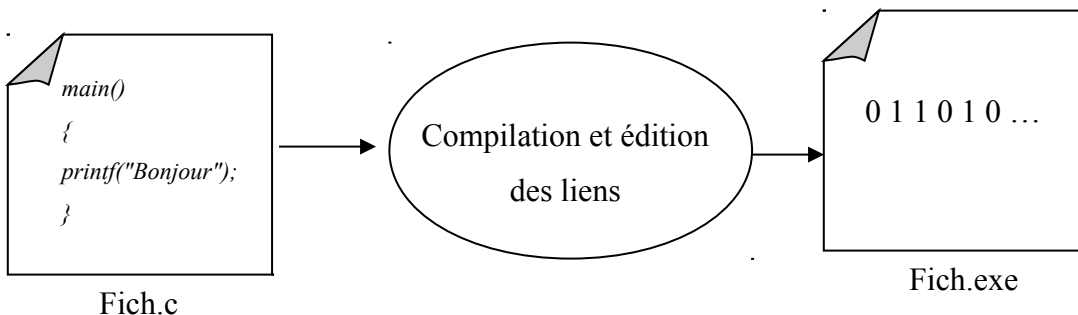
Un programme C est un texte écrit avec un éditeur de texte, respectant une certaine syntaxe et stocké sous forme d'un ou plusieurs fichiers (généralement avec l'extension .c). A l'opposé du langage assembleur, les instructions du langage C sont obligatoirement encapsulées dans des fonctions et il existe une fonction privilégiée appelée main qui est le point de départ de tout programme. Voici, un exemple de programme C

Ce programme affiche la chaîne de caractères Bonjour à l'écran. Pour afficher cette chaîne, le programme fait appel à la fonction printf

qui fait partie de l'une des fonctions prédéfinies fournies avec tout compilateur C (la fonction *write()* est son équivalente en langage Pascal). L'ensemble de ces fonctions prédéfinies (appelé bibliothèque C) est stocké dans un ou plusieurs fichiers(s). La traduction du fichier texte ci-dessus en un programme exécutable se décompose en deux phases:

- la compilation qui est la traduction d'un programme C en une suite d'instructions machine; le résultat produit est un fichier objet (généralement avec l'extension .o).
- l'édition des liens produit à partir d'un ou de plusieurs fichiers objets et des bibliothèques, un fichier exécutable. Outre l'assemblage des divers fichiers objets, l'édition des liens inclut les définitions des fonctions prédéfinies utilisées par le programme.

Exemple : compilation de fichier (fich.c) contenant un programme :



Les fichiers include :

Pour compiler correctement un fichier, le compilateur a besoin d'informations concernant les déclarations de structures de données et de variables externes ainsi que de l'aspect (on dira prototype) des fonctions prédéfinies se trouvent dans des fichiers avec l'extension .h.

Ces fichiers doivent être inclus dans le fichier que l'on veut compiler. Pour ce faire, le langage C offre la directive du préprocesseur.

include nom de fichier

```
#include <stdio.h>
main()
{
printf("Bonjour");
}
```

Par exemple, pour utiliser la fonction *printf* (fonction qui affiche une chaîne de caractère sur l'écran), il faut inclure le fichier « `stdio.h` », qui contient les déclarations de variables externes et les prototypes de fonctions de la bibliothèque d'entrée-sortie standard (standard input/output), dans le fichier que l'on veut compiler de la manière suivante:

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

Voici la version correcte du programme présenté précédemment

Les commentaires :

Dès lors que l'on écrit un programme important, il est indispensable d'y inclure des

commentaires qui ont pour but d'expliquer ce qu'est sensé faire le programme, les conventions adoptées et tout autre information rendant le programme lisible à soi même et à autrui.

Un commentaire commence par les caractères `/*` et se terminent par `*/`. A l'intérieur de ces délimiteurs toute suite de caractères est valide (sauf évidemment `*/`).

Exemple :

```
/* Ce programme imprime la chaîne de caractères "bonjour" à  
l'écran  
*/  
#include <stdio.h> /* Fichier include pour utiliser la fonction  
printf */  
main()  
{  
    printf(" Bonjour ");  
}
```

Les types de données :

1) Les entiers :

En C, on dispose de divers types d'entiers qui se distinguent par la place qu'ils occupent en mémoire :

- sur 1 octet, les entiers signés et non signés (char) et (unsigned char).
- sur 2 octets, les entiers signés et non signés (short) et (unsigned short).
- sur 4 octets, les entiers signés et non signés (long) et (unsigned long) .
- le type int (unsigned int) est selon les machines synonymes de short (unsigned short) ou de long (unsigned long) 4

1.1) Le type char :

Le type char désigne un entier signé codé sur 1 octet.

Décimal	Caractère
0	NULL
...	...
48	0
...	...
57	9
...	...
65	A
...	...
90	Z
...	...
97	a
...	...
122	z
...	...
127	

Comme nous pouvons constater le type char n'est qu'un entier codé sur un octet. Il en découle que toutes les opérations autorisées sur les entiers peuvent être utilisées sur les caractères, on peut par exemple ajouter ou soustraire deux caractères, ajouter ou soustraire un entier à un caractère.

1.2) Les types short, long ou int

Le type short représente un entier signé codé sur 2 octets (de -32768 à 32767) et le type unsigned short représente un entier non signé codé sur 2 octets (de 0 à 65535). Le type long (ou int pour nos machines) représente un entier signé codé sur 4 octets (de -2147843648 à 2147843647) et le type unsigned long (ou unsigned int pour nos machines) représente un entier non signé codé sur 4 octets (de 0 à 4294967295).

2) Le type réel :

Les nombres à virgule flottante (abusivement appelés réels) servent à coder de manière approchée les nombres réels. Un nombre à virgule flottante est composé d'un signe, d'une mantisse et d'un exposant. On dispose de trois types de nombres à virgule flottante, les types **float**, **double** et **long double**.

3) Les constantes de type caractère :

Les constantes de type caractère se note entre apostrophes:

'a' , '2'

Le caractère ' se note \' et le caractère \ se note '\\'. On peut également représenter des caractères non imprimables à l'aide de séquences d'échappement. Voici une liste non exhaustive de caractères non imprimable:

Séquence	
<code>\n</code>	nouvelle ligne
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\b</code>	retour d'un caractère en arrière
<code>\r</code>	retour chariot
<code>\f</code>	saut de page
<code>\a</code>	beep
<code>\'</code>	apostrophe
<code>\"</code>	guillemet
<code>\\</code>	anti-slash
<code>\ddd</code>	code ASCII en notation octale
<code>\xdd</code>	code ASCII en notation hexadécimale

4) Les chaînes de caractères :

Les chaînes de caractères se note entre guillemets:

"langage C" "C'est bientôt fini !!!"

```

La fonction suivante : /*variable de type entier*/
printf("Bonjour\nComment ça va\n"); /*variable de type caractère */
float q = 1.3; /* variable réel initialisée par 1,3 */

```

produit la sortie suivante :

```

Bonjour
Comment ça va

```

Une chaîne de caractères est une suite de caractères (éventuellement vide) entre guillemets. Il en découle que l'on est autorisé à utiliser les séquences d'échappement dans les chaînes.

Exemple :

En mémoire, une chaîne de caractères est une suite de caractères consécutifs et dont le dernier élément est le caractère nul '\0'.

Définir et déclarer des variables

En C, toute variable utilisée dans un programme doit auparavant être définie. La définition d'une variable consiste à la nommer et lui donner un type et éventuellement lui donner une valeur initiale (on dira initialiser). C'est cette définition qui réserve (on dira alloue) la place mémoire nécessaire en fonction du type.

Initialiser une variable consiste à remplir, avec une constante, l'emplacement réservé à cette variable. Cette opération s'effectue avec l'opérateur =.

Exemple :

Un certain nombre d'identificateurs sont réservés et ne peuvent être utilisés comme noms de variables. Voici la liste des noms réservés :

```

auto   break   case   char   const   continue   default
                                     do
double  else   enum   extern  float   for   goto   if
                                     int
long    register  return  short   signed  sizeof  static

```

struct
switch *typedef* *union* *unsigned* *void* *volatile*
while

Les entrées-sorties :

Un programme n'a d'intérêt que dans la mesure où il communique avec l'utilisateur. Il faut donc que l'utilisateur lui entre des données et que le programme, après calcul sorte des résultats. Les fonctions d'entrée-sortie vont permettre de réaliser cette communication.

1) Les sorties.

Les résultats calculés par un programme sont affichés à l'utilisateur soit sur l'écran (sortie standard), soit dans un fichier de résultats.

sortie formatée avec printf : La fonction printf affiche à l'écran les arguments fournis.

La syntaxe de cette fonction est de la forme :

```
printf("format", arg1, arg2, ..., argN); .
```

L'argument format est une chaîne de caractères contenant éventuellement des spécifications.

Une spécification de format est donnée par le caractère % suivi d'une ou plusieurs lettres clé.

Voici à titre d'exemple, quelques spécifications possibles:

```

#include<stdio.h>

main()
{
    int i,j;           /*déclaration de deux variables i et j de
type entier*/
    float x;          /*déclaration de variable de type réel*/
    char c;           /*déclaration de variable de type
caractère*/
    i=5;              /*initialisation de i par 5*/
    j=i+1;            /*initialisation de j par i+1*/
    x=3.2;            /* initialisation de x par 3.2*/
    c='A';            /*initialisation de c par le caractère A*/

    printf(" la valeur du nombre réel est : %f " ,x );
    printf(" la variable c contient le caractère : %c", c );
    printf(" i= %d et j=%d" ,i , j );
}

```

1)

%c	caractère
%s	chaîne
%d	entier
%u	entier non signé
%f,%g	flottant
%e,%E	flottant avec exposant
	...

Exemple :**2) Les entrées.**

Il s'agit de donner au programme les données nécessaires à son exécution. Par exemple, si l'on veut calculer la somme de deux entiers,

le programme attend de l'utilisateur qu'il donne la valeur des entiers. Selon la nature du programme, l'utilisateur aura le choix de fournir ces valeurs de manière interactive (entrée standard), soit en remplissant un fichier contenant ces valeurs.

Entrée formatée avec *scanf*. La fonction *scanf* lit à la console des données. La syntaxe de cette fonction est de la forme :

```
scanf("format", &arg1, &arg2, ..., &argN); .
```

L'argument format est celui décrit plus haut. On notera la présence du symbole & devant chaque argument à imprimer. Lorsqu'une lecture est faite avec *scanf*, il est impératif de rentrer quelque chose qui est rigoureusement identique au format que l'on a défini.

Exemple 1:

scanf("%d", &x); : Le contenu de la variable x est rempli avec ce que l'utilisateur fournira au clavier. Il faut donc entrer un entier suivi d'un retour à la ligne.

Autre entrée. : On dispose également des fonctions *int getchar(void)* et *char *gets(char *s)* qui retourne un caractère (resp. une chaîne de caractères) lu par le clavier.

Exemple 2 :

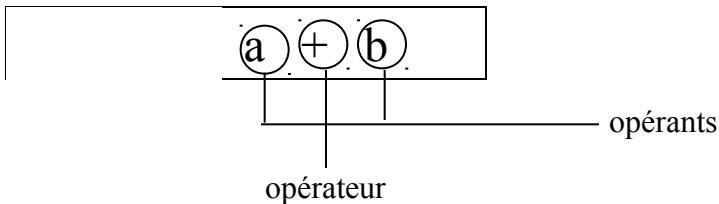
```
#include<stdio.h>
```

```
main()
{
    int i,j;          /*déclaration de deux variables i et j de
type entier*/
    scanf("%d",&i);  /* lecture à partir du clavier de la valeur de
i */
    scanf("%d",&j);  /* lecture à partir du clavier de la valeur de
j */
    printf(" la somme de i et j : %d", i+j);
}
```

CHAPITRE 2 : OPÉRATEURS ET EXPRESSIONS

3) Généralités sur les opérateurs :

Une expression est un objet syntaxique obtenu en assemblant correctement des constantes, des variables et des opérateurs. Par exemple, l'expression $x+3$ est une expression. Dans le langage C, il y a bien d'autres opérateurs que les opérateurs arithmétiques (+, -, /, ...) qu'on a l'habitude de manipuler; il y en a, en fait, plus de quarante opérateurs



4) Opérateurs et priorités :

Nous avons l'habitude de manipuler des expressions (par exemple arithmétiques) et il nous est relativement aisé de préciser exactement le sens des expressions comme : $2+3*4*5-2$, $2-3-4$ etc...

On sait que ces expressions sont équivalentes à $(2+(3*(4*5)))-2$, $(2-3)-4$.

Introduire les parenthèses permet de définir sans ambiguïté l'expression que l'on manipule. A priori, c'est notre culture mathématique qui nous permet de parenthéser ces expressions. Pour éviter l'usage des parenthèses qui alourdissent la lecture, lorsque cela est possible, les mathématiciens ont fixé des règles de priorité pour entre les opérateurs. Par exemple, dans l'expression $2+3*4$, la sous

expression $3*4$ est évaluée en premier et le résultat obtenu est ajouté à la valeur 2 (forme parenthésée : $2 + (3 * 4)$). On dit que l'opérateur * possède une priorité supérieure à la priorité de l'opérateur +.

De même, dans l'expression $2-3-4$, la sous expression $2-3$ est évaluée en premier et, au résultat obtenu, on soustrait la valeur 4 (forme parenthésée : $(2 - 3) - 4$). On dit que l'ordre (d'évaluation) de l'opérateur - est de gauche à droite.

En effet, on utilise les parenthèses lorsqu'on veut évaluer une expression d'une manière autre que celle définie à l'aide de la priorité et de l'ordre d'évaluation. Par exemple, si l'on veut faire $2+3$ et multiplier le résultat par 4, on notera $(2+3)*4$ et il n'est pas possible de l'écrire sans les parenthèses.

5) Les opérateurs du langage C :

5.1) L'opérateur d'affectation :

L'opération la plus importante dans un langage de programmation est celle qui consiste à donner une valeur à une variable. Cette opération est désignée par le symbole =

L'affectation range une valeur dans une variable (une zone mémoire).

Exemple :

- l'affectation $x=2$ range la valeur 2 dans la variable x,
- l'affectation $x=y$ range dans la variable x le contenu de la variable y,
- l'affectation $x=y+1$ range dans la variable x le contenu de la variable y incrémenté de 1.

Remarque :

Une affectation peut figurer en membre droit d'une autre affectation.

L'affectation $x=y=1$ est parfaitement valide car elle représente l'affectation $x=(y=1)$. Puisque $y=1$ est une expression, elle peut figurer donc en membre droit d'une affectation. Puisque elle est syntaxiquement juste.

5.2) Les opérateurs arithmétiques :

Opérateur	Nom	Notation	Priorité	Ordre
+	Addition	$x+y$	12	gauche-droite
-	soustraction	$x-y$	12	gauche-droite
*	multiplication	$x * y$	13	gauche-droite
/	division	x/y	13	gauche-droite
%	modulo	$x\%y$	13	gauche-droite

Les opérateurs $+$, $-$, $*$ fonctionnent comme on s'y attend. Par contre, l'opérateur $/$ se comporte de manière différente selon que les opérandes sont des entiers ou des nombres flottants. Lorsqu'il s'agit de nombres flottants, le résultat est un nombre flottant obtenu en divisant les deux nombres. Lorsqu'il s'agit de nombres entiers, le résultat est un nombre entier obtenu en calculant la division entière.

L'opérateur $\%$ n'est défini que pour les entiers et le résultat est le reste de la division entière des opérandes.

Rappel sur la division entière : On appelle **quotient (q)** et **reste (r)** de la division entière de a et de b, les nombres entiers q et r vérifiant :

$$a = q * b + r; \quad 0 \leq r < b$$

$$\begin{array}{r|l}
 35 & 2 \\
 \hline
 & 17 \\
 \hline
 & 1
 \end{array}$$

Le reste ← $r_0 = 1$ 17 → quotient

Exemple : Si $a=20$ et $b=3$ alors $q=6$ et $r=2$ ($20 = 6 * 3 + 2$).

5.3) Les opérateurs de comparaison :

Opérateur	Nom	Notation	Priorité	Ordre
==	test d'égalité	$x == y$	9	gauche-droite
!=	test de non-égalité	$x != y$	9	gauche-droite
<=	test d'inférieur ou égal	$x <= y$	10	gauche-droite
>=	test de supérieur ou égal	$x >= y$	10	gauche-droite
<	test d'inférieur strict	$x < y$	10	gauche-droite
>	test de supérieur strict	$x > y$	10	gauche-droite

Théoriquement, le résultat d'une comparaison est une valeur booléenne (vrai ou faux). Dans le langage C, le résultat d'une comparaison est 1 ou 0 selon que cette comparaison est vraie ou fausse.

Il n'existe pas de type booléen en C; la valeur entière 0 sera considérée comme équivalente à la valeur faux et toute valeur différente de 0 équivalente à la valeur vrai.

Attention !!!

Il ne faut pas confondre == (test d'égalité) et = (affectation).

Cette confusion (souvent involontaire) est source de nombreuses erreurs. Cette confusion est d'autant plus facile à faire que les deux écritures sont syntaxiquement correctes :

```
if (x == 2) { ... }
```

```
if (x = 2) { ... }
```

La première teste l'égalité de x et de 2, alors que la deuxième teste la valeur de l'affectation $x=2$ qui vaut 2. Ainsi, selon que la valeur de x est 2 ou pas, la première forme fera des choses différentes. Par contre, quelque soit la valeur de x , la valeur de l'expression $x=2$ est toujours 2.

Est donc :

La condition *if* ($x == 2$) { ... } est vrai si x égal à 2

La condition *if* ($x = 2$) { ... } est toujours vrai

5.4) Les opérateurs logiques :

Rappel de logique : Une variable booléenne est une variable pouvant prendre la valeur vrai ou faux.

Dans le langage C, on dispose des opérateurs logiques (sous une syntaxe particulière) avec lesquels on peut construire des expressions. La valeur d'une expression booléenne est, comme le résultat des comparaisons, une valeur entière.

Opérateur	Nom	Notation	Priorité	Ordre
&&	ET	x && y	5	gauche-droite
	OU	x y	4	gauche-droite
! (unaire)	NON	! x	14	droite-gauche

exemple :

(1): ($x < 2$ && $y > 3$)

L'expression (1) est vrai si x est inférieur à 2 et y supérieur à 3.

5.5) Les autres opérateurs binaires d'affectation :

Les opérateurs suivants ne sont que des raccourcis de notation :

Opérateur	équivalent	Notation	Priorité	Ordre
+=	$x = x + y$	$x += y$	2	droite-gauche
-=	$x = x - y$	$x -= y$	2	droite-gauche
. *=	$x = x * y$	$x *= y$	2	droite-gauche
/=	$x = x / y$	$x /= y$	2	droite-gauche
%=	$x = x \% y$	$x \% = y$	2	droite-gauche
>>=	$x = x >> y$	$x >> = y$	2	droite-gauche
<<=	$x = x << y$	$x << = y$	2	droite-gauche
&=	$x = x \& y$	$x \& = y$	2	droite-gauche
=	$x = x y$	$x = y$	2	droite-gauche
^=	$x = x \^ y$	$x \^ = y$	2	droite-gauche

5.6) Les autres opérateurs unaires d'affectation :

++	$x = x + 1$	$x++$ ou $++x$	14	droite-gauche
--	$x = x - 1$	$x--$ ou $--x$	14	droite-gauche

Les opérateurs d'incrémention et de décrémention sont en position de préfixe ou de suffixe :

L'expression $x++$ est le raccourci pour :

$$x = x + 1;$$

L'expression $y = x++$ est le raccourci pour :

$$y = x;$$

$$x = x + 1;$$

L'expression $y = ++x$ est le raccourci pour :

$$x = x + 1;$$

$$y = x;$$

CHAPITRE 3 : LES STRUCTURES DE CONTRÔLE

6) Introduction :

Un programme C est constitué de définitions et déclarations de variables et de fonctions. Les fonctions sont construites à l'aide d'instructions combinées entre elles avec des structures de contrôles.

Qu'est qu'une instruction?

Une expression telle que $x=0$, $i++$ ou `printf("langage C \n")` sont des instructions lorsqu'elles sont suivies du caractère point-virgule (;).

```
x=0;
```

```
i++;
```

```
printf("langage C \n");
```

Le point-virgule est appelé terminateur d'instruction.

Les instructions composées ou blocs sont construites avec des instructions simples regroupées à l'aide des accolades { et }. On peut déclarer des variables au début d'un bloc. L'accolade fermante n'est pas suivie d'un point-virgule.

```
{  
  int i = 0;  
  i++;  
  printf("langage C\n");  
}
```

L'instruction if :

Cette instruction conditionnelle permet d'exécuter des instructions de manière sélective en fonction du résultat d'un test. La syntaxe de l'instruction est :

```

if (i<0)
    printf(" la valeur de i est négative")
Else
    printf(" la valeur de i est positive") ;

```

```

if ( expression) instruction1
ou
if ( expression)
    instruction1
else
    instruction2

```

Si l'expression est vraie, l'instruction1 s'exécute; sinon, dans le deuxième cas, c'est l'instruction2 qui s'exécute.

Exemple :

si la valeur de i est négative le programme affichera sur l'écran " la valeur de i est négative"

si la valeur de i est positive le programme affichera sur l'écran " la valeur de i est positive"

Les instructions while, do et for :

Les instructions itératives ou boucles sont réalisées à l'aide d'une des trois structures de contrôle suivantes:

<pre> while (expression) instruction </pre>
<pre> do instruction while (expression); </pre>
<pre> for (expression1; expression2; expression3) instruction </pre>

```

#include<stdio.h>
main()
{
    x = 10;
    while (x >= 0)                /* tant que x est positif */
    {
        printf("%d ", x);        /* afficher x */
        x = x -1;                /* décrémenter x */
    }
}

```

1) L'instruction *while* :

La structure de contrôle *while* évalue l'expression et exécute l'instruction tant que cette expression est vraie.

Exemple :

Le programme suivant affiche à l'écran (dans l'ordre) les nombres de 10 à 0.

Le résultat affiché par ce programme :

10 9 8 7 6 5 4 3 2 1 0

2) L'instruction *do ... while* :

Une variante de l'instruction *while* est l'instruction *do ... while*. Contrairement à l'instruction *while*, l'instruction :

do instruction while (expression1);

est exécutée au moins une fois. L'itération s'arrête lorsque l'*expression1* est fausse.

Exemple :

Le programme suivant fournit le même résultat que l'exemple précédent

```

main()
{
    x = 10;
    do
    {

```

```
printf("%d", x);           /* afficher x */
x = x -1;                 /* décrémenter x */
} while (x >= 0)          /* tant que x est positif */
}
```

La différence entre ce programme et le programme précédent (programme avec l'instruction *while*) c'est que le programme précédent fait le teste sur x avant l'exécution de bloc d'instruction (*printf("%d", x)* et *x=x-1*), alors que ce programme effectue le test sur x après l'exécution du bloc d'instruction.

3) L'instruction for :

L'instruction for, comme l'instruction while, n'exécute l'instruction que si l'expression est vraie.

```
for ( expression1; /* Initialisation */
expression2; /* Conditions d'arrêt */
expression3) /* Fin du corps de boucle */
instruction;
```

Cette instruction qui est composée de trois parties séparées par des point- virgules:

- expression1 : constitue les initialisations nécessaires avant l'entrée dans la boucle.
- expression2 : constitue les conditions de bouclage.
- expression3 : constitue les instructions de fin de boucle.

Exemple :

Le programme suivant calcule le tableau de multiplication de 2.

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

```
main()
```

```
int c,i ;
```

```
For (i=1 ; i<=10 ; i++)
```

```
{
```

```
    c = 2 * i ;
```

```
    printf("2 x %d =",c);
```

```
}
```

L'exécution de ce programme donnera le

résultat suivant :

$$2 \times 1 = 2$$

$$2 \times 2 = 4$$

$$2 \times 3 = 6$$

$$2 \times 4 = 8$$

$$2 \times 5 = 10$$

$$2 \times 6 = 12$$

$$2 \times 7 = 14$$

$$2 \times 8 = 16$$

$$2 \times 9 = 18$$

$$2 \times 10 = 20$$

L'instructions break :

L'instruction break est utilisée dans les structures de contrôle :

switch, while, do ... while et for.

Dans une boucle, cette instruction provoque la sortie immédiate de la boucle sans tenir compte des conditions d'arrêt de la boucle.

L'utilisation de l'instruction break dans les boucles n'est que très rarement utilisée. Dans la quasi totalité des cas, une bonne condition d'arrêt sera utilisée pour sortir de la boucle.

L'instruction switch :

On dispose d'une instruction pour faire une étude de cas: c'est l'instruction switch.

```
switch ( expression)  
{  
    case constante_1 : instruction_1  
    case constante_2 : instruction_2  
    ...  
    case constante_N : instruction_N  
    default : instruction
```


}

Si la valeur de l'expression vaut *constante_I*, on exécute la suite des instructions commençant à *instruction_I*. Attention, cette instruction ne se contente pas d'exécuter les instructions comprises entre *instruction_I* et *instruction_I+1*; elle exécute toutes les instructions *instruction_I* ainsi que toutes celles qui suivent *instruction_I* jusqu'à la rencontre de l'instruction *break* ou de la fin de l'instruction *switch*.

Lorsque la valeur de l'expression est égale à aucune des constantes mentionnées, ce sont les instructions étiquetées par *default* qui seront exécutées.

Exemple :

```

#include<stdio.h>
main()
{
int c;
scanf("%d",&c);
switch(c)
{
case '1' :
case '2' : /* notez l'absence
d'instruction ici */
case '3' :
case '5' :
case '7' : printf("%d est un nombre premier\n", c);
break; /* notez l'instruction
break */
case '6' : printf("%d est un multiple de 3\n", c);
/* notez l'absence
d'instruction break */
case '4' :
case '8' : printf("%d est un multiple de 2\n", c);
break;
case '9' : printf("%d est un multiple de 3\n", c);
break;
default : /* instruction par
défaut */
printf("%d n'est pas un chiffre\n", c);
}
}

```

Dans le programme ci-dessus, on a regroupé les cas 1, 2, 3, 5 et 7 pour n'écrire qu'une seule fois l'instruction d'affichage. L'instruction break arrête l'exécution de ce switch. Le chiffre 6 est à la fois un multiple de 2 et de 3. On factorise une partie de l'affichage avec les cas 4 et 8; d'où l'absence de l'instruction break.

```
int MOIS[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

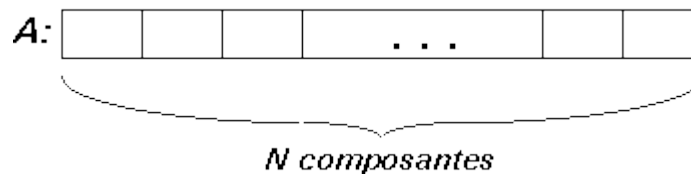
LES TABLEAUX

Les tableaux sont les variables structurées les plus populaires. Ils sont disponibles dans tous les langages de programmation et servent à résoudre une multitude de problèmes. Le traitement des tableaux en C ne diffère pas de celui des autres langages de programmation.

1) Les tableaux à une dimension

2) Définitions

Un tableau (unidimensionnel) A est une variable structurée formée d'un nombre entier N de variables simples du même type, qui sont appelées les **composantes** du tableau. Le nombre de composantes N est alors la **dimension** du tableau.



Exemple :

La déclaration d'un tableau MOIS unidimensionnel de type *int*

La déclaration avec initialisation d'un tableau MOIS unidimensionnel

Définit un tableau de 12 composants de type *int*. Les 12 composants sont initialisées par les valeurs respectives 31, 28, 31, ... , 31.

On peut accéder à la première composante du tableau par *MOIS[0]*, à la deuxième composante par *MOIS[1]*, . . . , à la dernière composante par *MOIS[11]*.

```

int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float B[10] = {1.0, 2.6, 7.8, 9.9, 10.0, 11.1, 12.2, 13.3, 14.4, 15.5};
tableau char C[10] = {"10, 20, 30, 40, 50"};
int D[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float E[10] = {-1.05, 3.33, 8e2, -3};
int F[10] = {1, 3, 2};

```

2)

3) Déclaration de tableaux en C

<TypeSimple> <NomTableau>[<Dimension>];

Exemples

4) Initialisation et réservation automatique

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades.

Exemples

Il faut évidemment veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro (cas de C[10])

Si la longueur du tableau n'est pas indiquée explicitement lors de l'initialisation, alors l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

Exemples

Réservation de 5 emplacement de type int

Réservation de 5 emplacement de type float .

Les déclarations suivantes sont incorrectes :

5) Accès aux composantes

En déclarant un tableau par:

```
int A[5];
```

```

#include<stdio.h>
main()
{
    float T[5];
    int i;
    for(i=0;i<5;i++) /*i est un indice nous permettant de passer
de premier à dernier element du tableau */
    {
        printf("donner l'element numero : %d \n",i);
        scanf("%f",&T[i]);
        S+=T[i];
    }
    printf("la somme de ces elements : %f",S);
} printf("vous avez saisi la valeur :%f pour l'element numero :
%d\n",T[i],i);
}

```

Nous avons défini un tableau A avec cinq composantes, auxquelles on peut accéder par:

$A[0], A[1], \dots, A[4]$

Le premier élément du tableau A est A[0] et le dernier est A[4]

Si nous avons la déclaration suivante :

`int A[5]={3,5,1,2,8} ;`

L'instruction :

`printf("%d",A[2])`

Affichera 1 sur l'écran (troisième élément du tableau).

Application 1:

On veut introduire successivement 5 éléments de type réel à partir du clavier et les afficher ensuite.

Application 2 :

Calculer la somme de 5 nombres de type réel.

Le programme suivant donne le même résultat :

```
#include<stdio.h>
main()
{
    float T[5],S;
    int i;
    S=0;
    for (i=0;i<5;i++)
    {
        printf("donner l'element numero : %d \n",i);
        scanf("%f",&T[i]);
    }
    for (i=0;i<5;i++)
        S=S+T[i];
    printf("\nla somme de ces elements : %f",S);
}
```

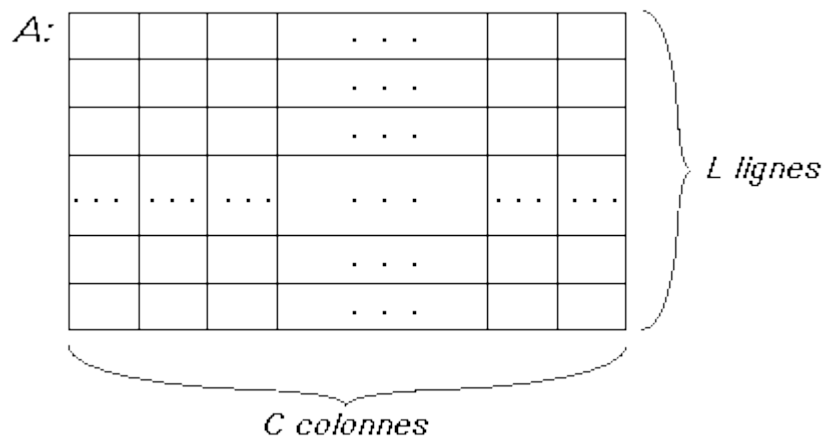
Les

tableaux à deux dimensions

1) Définitions

En C, un tableau à deux dimensions A est à interpréter comme un tableau (uni-dimensionnel) de dimension L dont chaque composante est un tableau (uni-dimensionnel) de dimension C.

On appelle L le **nombre de lignes** du tableau et C le **nombre de colonnes** du tableau. L et C sont alors les deux **dimensions** du tableau. Un tableau à deux dimensions contient donc **L*C composantes**.



```
float NOTE[20][10] = {{12, 11, ... , 15, 10},
                      {9, 7, ... , 12, 14},
                      ...
                      {10, 11, ... , 16, 10}};
```

```
float NOTE[20] = {10, 13, ... , 16, 8};
```

On dit qu'un tableau à deux dimensions est **carré**, si L est égal à C.

En faisant le rapprochement avec les mathématiques, on peut dire que "A est un vecteur de L vecteurs de dimension C", ou mieux:

"A est une **matrice** de dimensions L et C".

Exemple

Considérons un tableau *NOTES* à une dimension pour mémoriser les notes de 20 élèves d'une classe dans un devoir:

NOTE:	45	34	...	50	48
	A[0]	A[1]		A[18]	A[19]

Supposons maintenant qu'on veut mémoriser les notes des élèves dans les 10 devoirs d'un trimestre, nous pouvons rassembler plusieurs de ces tableaux uni-dimensionnels dans un tableau *NOTES* à deux dimensions :

12	11	...	15	10
9	17	...	12	14
...
10	11	...	16	1

Dans une colonne nous retrouvons les notes de tous les élèves dans un devoir. Dans une ligne, nous retrouvons toutes les notes d'un élève.

L'élève 2 a 17 dans le deuxième devoir (*NOTE[1][1]*) et 15 dans le neuvième devoir (*NOTE[1][8]*).

L'élève 10 a 10 dans le premier devoir (*NOTE[9][0]*) et 16 dans le neuvième devoir (*NOTE[9][8]*).

```
int A[3][10][20]; /* {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19},
float B[2][20]; /* {10,11,12,13,14,15,16,17,18,19},
char C[3][3]; /* {1,12,23,34,45,56,67,78,89,90} */;
```

1)

2) Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades. À l'intérieur de la liste, les composantes de chaque ligne du tableau sont encore une fois comprises entre accolades. Pour améliorer la lisibilité des programmes, on peut indiquer les composantes dans plusieurs lignes.

Exemple

Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite. Nous ne devons pas nécessairement indiquer toutes les valeurs: Les valeurs manquantes seront initialisées par zéro. Il est cependant défendu d'indiquer trop de valeurs pour un tableau.

3) Déclaration :

```
<TypeSimple> <NomTabl>[<DimLigne>][<DimCol>];
```

4) Accès aux composants

Pour accéder à un élément d'un tableau de deux dimensions, on doit indiquer l'indice de ligne et de colonne de cet élément.

Exemple :

Considérons la déclaration suivante :

L'indice de ligne du tableau T varie de 0 à 9 (10 lignes), et pour les colonnes varie 0 à 19.

```

#include<stdio.h>
main()
{
    float T[3][5];
    int i,j;
    for (i=0;i<3;i++)
        for (j=0;j<5;j++)
            {
                printf("donner l'element de l'indice: %d , %d \n",i,j);
                scanf("%f",&T[i][j]);
            }
    for (i=0;i<3;i++)
        for (j=0;j<5;j++)
            printf("T [%d , %d]=%f \n",i, j, T[i][j]);
}

```

Pour affecter la valeur 30 à la case qui a comme l'indice de ligne 3 et l'indice de colonne 7 on peut utiliser l'instruction suivante:

```
T[3][7]=30 ;
```

Si on veut charger la valeur de la case d'indice (5 , 11) à partir du clavier l'instruction s'écrit :

```
scanf("%f",&T[5][11]);
```

Application 1:

Le programme suivant lit à partir du clavier des valeurs pour les charger dans un tableau de deux dimensions ensuite il les affiche sur l'écran :

Les fonctions

Les fonctions en C correspondent aux fonctions et procédures en Pascal et en langage algorithmique. Nous avons déjà utilisé des fonctions prédéfinies dans des bibliothèques standard (printf et scanf de `<stdio>...`). Dans ce chapitre, nous allons découvrir comment nous pouvons définir et utiliser nos propres fonctions.

5) Modularisation de programmes

Pour des problèmes plus complexes, nous obtenons de longues listes d'instructions, peu structurées et par conséquent peu compréhensibles. En plus, il faut souvent répéter les mêmes suites de commandes dans le texte du programme, ce qui entraîne un gaspillage de mémoire interne et externe.

La plupart des langages de programmation nous permettent de subdiviser nos programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts. A l'aide de ces structures nous pouvons **modulariser** nos programmes pour obtenir des solutions plus élégantes et plus compréhensibles.

La modularité se fait en C à l'aide des fonctions, voici les avantages d'un programme utilisant les fonctions :

- Meilleure lisibilité
- Diminution du risque d'erreurs
- Possibilité de tests sélectifs
- Réutilisation de fonctions déjà existantes : il est facile d'utiliser des fonctions que nous avons conçus ou qui ont été développées par d'autres personnes.
- Simplicité de l'entretien : une fonction peut être changée ou remplacée sans devoir toucher aux autres fonctions du programme.

- Favorisation du travail en équipe : un programme peut être développé en équipe par délégation de la programmation des fonctions à différentes personnes ou groupes de personnes. Une fois développées, les fonctions peuvent constituer une base de travail commune.

6) Exemples de fonctions en C

Les programmes présentés ci-dessous donnent un petit aperçu sur les propriétés principales des fonctions en C.

Afficher un rectangle d'étoiles

Le programme suivant permet d'afficher à l'écran un rectangle de longueur L et de hauteur H, formé d'astérisques '*' :



A diagram showing a rectangle formed by asterisks. The width is labeled 'L' and the height is labeled 'H'. The rectangle consists of 6 rows of 15 asterisks each.

```

/*****/
#include <stdio.h>
int main()
{
    int L,H;
    printf("donner le nombre de lignes");
    scanf("%d",&L);
    printf("donner le nombre de colonnes");
    scanf("%d",&H);
    ligne(L,H);
    return 0;
}

void ligne(int L,int C)
{
    for(i=1;i<=L;i++)
    {
        printf("\n");
        for(j=1;j<=C;j++)
        printf(" ");
    }
}

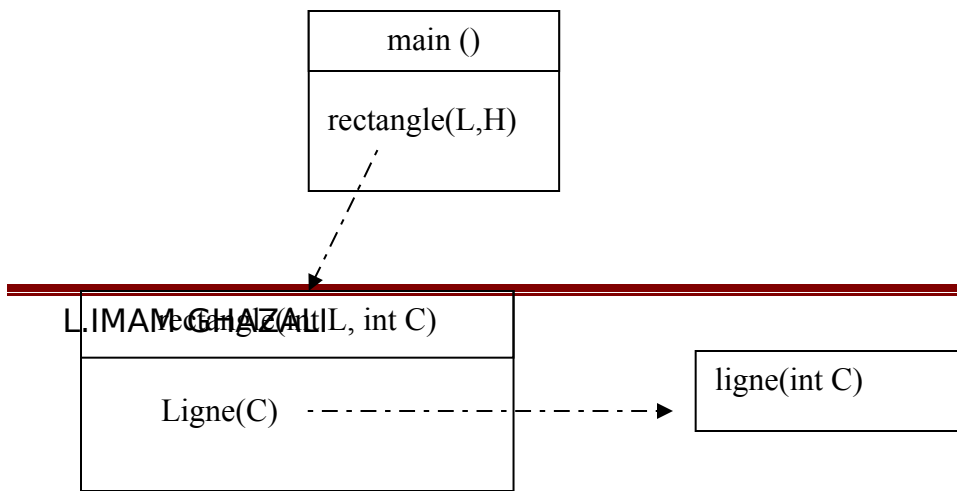
void rectangle(int L,int C)
{
    printf("donner le nombre de lignes");
    scanf("%d",&L);
    printf("donner le nombre de colonnes");
    scanf("%d",&H);
    ligne(L,H);
}

```

Nous pouvons rendre notre programme plus modulaire et plus lisible en introduisant une fonction rectangle qui permet d'afficher le rectangle :

On peut encore introduire une fonction ligne :

Schématiquement, nous pouvons représenter la hiérarchie des fonctions du programme comme suit:



Déclaration et définition de fonctions

En général, le nom d'une fonction apparaît à trois endroits dans un programme:

- 1) lors de la **déclaration**
- 2) lors de la **définition** (le programme de la fonction)
- 3) lors de l'**appel** de la fonction (dans main () par exemple)

Exemple

```

#include<stdio.h>
main()
{
    int i,j,L,H;
    void rectangle(int,int);
    printf("donner le nombre de lignes");
    scanf("%d",&L);
    printf("donner le nombre de colonnes");
    scanf("%d",&H);
    rectangle(L,H);
    fflush(stdin);
    getchar();
}

void rectangle(int L,int C)
{
    int i,j;
    for(i=1;i<=L;i++)
    {
        for(j=1;j<=C;j++)
            printf("**");
        printf("\n");
    }
}

```

The diagram illustrates the three locations where a function name appears in a C program:

- Déclaration de la fonction:** A box points to the line `void rectangle(int,int);` inside the `main()` function.
- Appel de la fonction:** A box points to the line `rectangle(L,H);` inside the `main()` function.
- Définition de la fonction:** A box points to the entire function definition block starting with `void rectangle(int L,int C)`.

1) Définition d'une fonction

Dans la définition d'une fonction, nous indiquons:

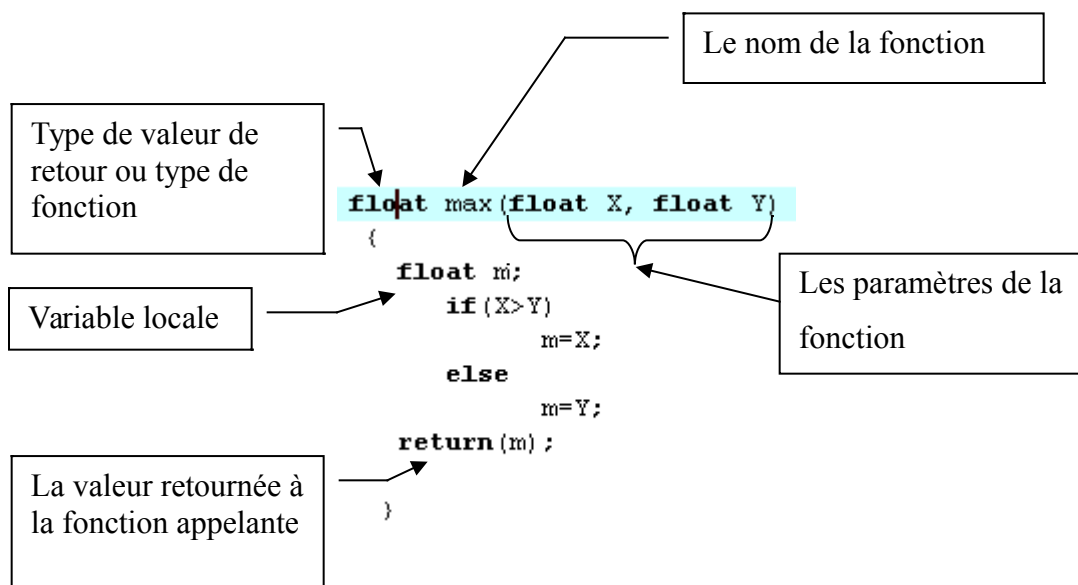
- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction

- les instructions à exécuter

Définition d'une fonction en C

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                        <TypePar2> <NomPar2>, ... )
{
<déclarations locales>
<instructions>
}
```

Exemple :



1.1) Les variables locales :

Les variables locales d'une fonction sont utilisées pour effectuer le traitement à l'intérieur de la fonction.

Dans l'exemple de la fonction rectangle on a déclaré deux variables locales `i` et `j`, pour réaliser les deux boucles `for` de la fonction rectangle.

```
int A, B;
```

```
...  
A = max (B, 2);
```

1.2) Type de fonction

Si une fonction F fournit un résultat du type T , on dit que 'la fonction F est de type T , pour notre exemple la fonction max est de type $float$.

Le type de fonction c'est le type de la valeur qui va être retournée à la fonction appelante.

Pour la fonction rectangle ne renvoie aucune valeur à la fonction appelante elle est donc de type $void$.

1.3) Les paramètres de la fonction :

Les *paramètres* ou *arguments* d'une fonction servent à passer données de l'extérieur (lors de l'appel de la fonction) et déterminent les actions et le résultat de la fonction. Techniquement, nous pouvons résumer le rôle des paramètres de la façon suivante:

Les paramètres d'une fonction sont simplement des **variables locales** qui sont initialisées par les **valeurs** obtenues lors de l'appel.

➤ Conversion automatique

Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction. Les paramètres sont automatiquement convertis dans les types de la déclaration avant d'être passés à la fonction.

Exemple

La déclaration de la fonction max est déclaré comme suit:

float max (float, float);

On suppose que dans la fonction $main$ nous avons les instructions suivantes,

Nous assistons à trois conversions automatiques:

```
float PI(void)
{
    return 3.1415927;
}
```

1.1)

Avant d'être transmis à la fonction, la valeur de B est convertie en float c'est à dire que si la valeur de B est 5, elle sera convertie à 5.0 ; la valeur 2 est convertie en 2.0 . Comme max est du type **float**, le résultat de la fonction doit être converti en **int** avant d'être affecté à A.

➤ Des fonctions sans paramètres :

il existe aussi des fonctions qui fournissent leurs résultats ou exécutent une action sans avoir besoin de données. La liste des paramètres contient alors la déclaration **void** ou elle reste vide.

Exemple :

La fonction PI fournit un résultat rationnel du type **float**. La liste des paramètres de PI est déclarée comme **void** (vide), c.-à-d. PI n'a pas besoin de paramètres et il faut l'appeler par: **PI()**

1.4) Renvoyer un résultat :

Par définition, toutes les fonctions fournissent un résultat d'un type que nous devons déclarer. Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau.

Pour fournir un résultat en quittant une fonction, on peut utiliser l'instruction **return**:

Syntaxe :

```
return <expression>;
```

return a les effets suivants:

- *Évaluation de l'<expression>*
- *Conversion automatique du résultat de l'expression dans le type de la fonction*

```
double CARRE(double X)    printf("Le carre de %f est %f \n", X, CARRE(X));
{
    return X*X;           /***** OU *****/
}                          Y=CARRE(X) + 2
```

- Renvoi du résultat
- Terminaison de la fonction

Exemples

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

Les appels de la fonction CARRE peut être intégré dans des calculs ou des expressions:

➤ **Fonction sans valeur à renvoyer (void) :**

Nous pouvons employer une fonction du type **void**.

La fonction de type *void* ne renvoie aucune valeur au programme appelant.

Exemple :

```
void rectangle(int L,int C)
{
    int i,j;
    for(i=1;i<=L;i++)
    {
        for(j=1;j<=C;j++)
            printf("*");
        printf("\n");
    }
}
```

L'appel de la fonction *rectangle* se fait de la manière suivante.

```

double CARRE(int L,int H)
{
  int i,j,L,H;
  printf("donner le nombre de lignes");
  scanf("%d",&L);
  printf("donner le nombre de colonnes");
  scanf("%d",&H);
  rectangle(L,H);      /***** ← appel de la fonction
rectangle*****/
  fflush(stdin);
  getchar();
}

```

L'appel
suivant, de
la fonction
rectangle,
n'est pas
correcte :

$Y = \text{rectangle}(L, H)$

Car la fonction *rectangle* ne renvoie aucune valeur est par
conséquence Y ne recevra aucune valeur.

2) Déclaration de la fonction :

Il faut déclarer chaque fonction avant de pouvoir l'utiliser. La
déclaration informe le compilateur du type des paramètres et du
résultat de la fonction.

➤ Prototype d'une fonction

La déclaration d'une fonction se fait par un prototype de la fonction
qui indique uniquement le type des données transmises et reçues par
la fonction.

Syntaxe :

<TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...);

Exemple :

Le prototype de la fonction rectangle :

Le prototype de la fonction CARRE :

NB : Lors de la *déclaration*, le nombre et le type des paramètres
doivent nécessairement correspondre à ceux de la définition de la
fonction.

2.1) Règles pour la déclaration des fonctions

De façon analogue aux déclarations de variables, nous pouvons déclarer une fonction localement ou globalement. La définition des fonctions joue un rôle spécial pour la déclaration. En résumé, nous allons considérer les règles suivantes:

➤ **Déclaration locale:**

Une fonction peut être déclarée localement dans la fonction qui l'appelle (avant la déclaration des variables). Elle est alors disponible à cette fonction.

➤ **Déclaration globale:**

Une fonction peut être déclarée globalement au début du programme (derrière les instructions `#include`). Elle est alors disponible à toutes les fonctions du programme.

➤ **Déclaration implicite par la définition:**

La fonction est automatiquement disponible à toutes les fonctions qui suivent sa définition.

3) Les variables locales et variables globales :

Les variables déclarées dans une fonction sont uniquement visibles à l'intérieur de cette fonction. On dit que ce sont des variables locales de la fonction.

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions sont disponibles à toutes les fonctions du programme. Ce sont alors des variables globales. En général, les variables globales sont déclarées immédiatement derrière les instructions `#include` au début du programme.

Exemple :

```

#include<stdio.h>
float X,Y                               /*variables globales*/
void rectangle(int L,int C)
{
    int i,j;
    for(i=1;i<=L;i++)
    {
        for(j=1;j<=C;j++)
            printf("*");
        printf("\n");
    }

}
main()
{
    int i,j,L,H;
    printf("donner le nombre de lignes");
    scanf("%d",&L);
    printf("donner le nombre de colonnes");
    scanf("%d",&H);
    rectangle(L,H);
    fflush(stdin);
    getchar();
}

```

```

void ligne(int C)
{
    int i;
    for(i=1;i<=C;i++)
        printf("*");
    printf("\n");
}

```

La variable *i* est une variable locale de la fonction *ligne*. Elle est visible uniquement dans la fonction *ligne*.

X et Y sont deux variables globales, ils sont visibles a toutes les fonctions du fichier.

LES POINTEURS

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

4) Adressage de variable :

Lorsque l'on exécute un programme, celui-ci est stocké en mémoire, cela signifie que d'une part le code à exécuter est stocké, mais aussi que chaque variable que l'on a défini a une zone de mémoire qui lui est réservée, et la taille de cette zone correspond au type de variable que l'on a déclaré.

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

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

On peut donc accéder à une variable de 2 façons :

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

L'adresse d'une variable est désignée par l'opérateur : &

Exemple :

```
short X;
```

```
short Y;
```

```
X=3;
```

```
Y=X*2;
```

```
Mémoire
```

```
0
```

L'adresse X (&X) est la case N° 2345, elle contient la valeur 3.

La variable Y se trouve dans la case mémoire N° 2347, elle contient la valeur 6.

```
&X= 2345
      3
```

Utilisation des pointeurs :

```
&Y= 2347
```

Les pointeurs ont un grand nombre d'intérêts :

- Ils permettent de manipuler de façon simple des données pouvant être importantes (au lieu de passer à une fonction un élément très grand (en taille) on pourra par exemple lui fournir un pointeur vers cet élément...)
- Les tableaux ne permettent de stocker qu'un nombre fixé d'éléments de même type. En stockant des pointeurs dans les cases d'un tableau, il sera possible de stocker des éléments de taille diverse, et même de rajouter des éléments au tableau en cours d'utilisation (la notion de tableau dynamique) ce qui n'est pas possible pour les tableaux statiques.
- Il est possible de créer des structures chaînées.

```

int* ptr;          /*pointeur ptr sur les variables de type
int*int A=10 ;
P=&A;             /*pointeur p sur les variables de type P
pointeur sur A*/

```

1) Déclaration des pointeurs :

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

```
type * Nom_du_pointeur
```

Grâce au symbole '*' le compilateur sait qu'il s'agit d'une variable de type *pointeur* et non d'une variable ordinaire, de plus, en précisant le type de variable, le compilateur saura combien de blocs suivent le bloc situé à l'adresse pointée.

Exemple :

2) Initialisation des pointeurs :

Après avoir déclaré un pointeur il faut l'initialiser. L'initialisation se fait de la manière suivante :

```
Nom_du_pointeur = &nom_de_la_variable_pointee;
```

On doit toujours initialiser un pointeur avant de l'utiliser.

Exemple :

Pour cet exemple le pointeur P reçoit l'adresse de la variable A, ce qui nous donne deux façons pour manipuler la variable A, soit en utilisant le nom de la variable (A) ou en utilisant le pointeur P, on dit que le pointeur P est un accès indirecte à la variable A.

3) Accéder à une variable pointée :

Après (et seulement après) avoir déclaré et initialisé un pointeur, il est possible d'accéder au contenu de l'adresse mémoire pointée par le pointeur grâce à l'opérateur '*'

Si P un pointeur, on doit distinguer *P et &P : &P contient l'adresse de la variable dont le pointeur P pointe, et *P permet d'accéder à la valeur de la variable sur laquelle pointe P.

Exemple 1 :

```
int *P ;
int A=10 ;
P=&A ;           /*le pointeur p pointe
sur A*/
*p=*p+1 ;       /*equivalent à
A=A+1*/
*p=5 ;          /*équivalent à : A=5*/
```

Exemple 2 :

Main()	
int X,Y ; int *P ;	
X=10 ; Y=5 ;	
P=&X ; Printf(" *P=%d et X= %d",*P,X) ; /*affichera : *P=10 et X= 10*/	
P=&Y ; Printf(" *P=%d et Y= %d",*P,Y) ; /*affichera : *P=5 et Y= 5*/	
*P=6 ; Printf(" *P=%d et Y= %d",*P,Y) ; /*affichera : *P=6 et Y= 6*/	

4) Les opérations élémentaires sur les pointeurs :

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

➤ **Priorité de * et &**

Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incréméntation ++, la décrémentation --). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

Exemple :

Après l'instruction

P = &X;

Les expressions suivantes, sont équivalentes:

Y =	Y =
*P+1	X+1
*P =	X =
*P+10	X+10
*P	X
+= 2	+= 2
++*P	+
(*P)+	+X
+	X+
+	+

Dans le dernier cas, les parenthèses sont nécessaires:

Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas* l'objet sur lequel P pointe.

➤ **On peut uniquement affecter des adresses à un pointeur.**

Le pointeur NUL :

Seule exception: La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.

```
int *P;
P = 0;
```

Les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur int, alors l'affectation

```
P1 = P2;
```

Copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

Les pointeurs et tableaux

Les pointeurs et les tableaux sont conceptuellement très similaires en C. A l'exception des tableaux multidimensionnels, toute opération que l'on effectue par indexation dans un tableau peut être réalisée avec des pointeurs.

En simplifiant, nous pouvons retenir que le nom d'un tableau est un pointeur constant sur le premier élément du tableau.

Exemple :

En déclarant un tableau A de type int et un pointeur P sur int,

```
int A[10];
int *P;
```

L'instruction:

$P = A;$ est équivalente à $P = \&A[0];$



Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante. Plus généralement,

P+i pointe sur la i-ième composant devant P:

Ainsi, après l'instruction,

```

/*version 2 : sans pointeur*/
#include<stdio.h>
main()
{
    int T[10];
    int *P;
    for(P=T;P<T+10;P++)
        *P=1;
}

```

```

/*OU *(T+i)=1*/
for(i=0;i<10;i++)
    *(P+i)=1;
}

```

P = A;

Le pointeur P pointe sur A[0], et

***(P+1)** désigne le contenu de A[1]

***(P+2)** désigne le contenu de A[2]

...

***(P+i)** désigne le contenu de A[i]

Puisque le nom tableau est un pointeur constant sur le premier

élément on peut écrire :

***(A+1)** désigne le contenu de A[1]

***(A+2)** désigne le contenu de A[2]

...

***(A+i)** désigne le contenu de A[i]

Exemples :

Le programme suivant initialise les éléments du tableau T à

1

.

On peut utiliser un pointeur P qui pointe sur le tableau T :

Le programme suivant donnera le même résultat :

Remarque importante :

Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:

- Un *pointeur* est une variable, donc des opérations comme $\mathbf{P} = \mathbf{A}$ ou $\mathbf{P}++$ sont permises.
- Le *nom d'un tableau* est une constante, donc des opérations comme $\mathbf{A} = \mathbf{P}$ ou $\mathbf{A}++$ sont impossibles.

```
int t[3][4] = {{ 0, 1, 2, 3},
              {10,11,12,13},
              {20,21,22,23}};
```

Pointeur sur les tableaux de deux dimensions :

Comme pour les tableaux à un indice, l'identificateur d'un tableau, employé seul, représente toujours son "adresse" de début. Lorsque le compilateur rencontre une déclaration telle que: `int t[3][4]`; il considère en fait que `t` désigne un tableau de 3 éléments, chacun de ces éléments étant lui même un tableau de 4 entiers. Autrement dit, si `t` représente bien l'adresse de début de notre tableau `t`, il n'est plus de type `int *` (comme c'était le cas pour un tableau à un indice) mais d'un type "pointeur sur des blocs de 4 entiers", type qui devrait se noter théoriquement 4.

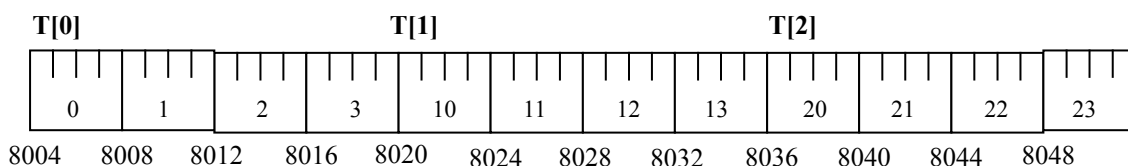
Exemple:

On suppose `T` à deux dimensions défini comme suit:

Le nom du tableau `T` représente l'adresse du premier élément du tableau et pointe sur le **tableau** `T[0]` qui a les valeurs: `{0,1,2,3}`

L'expression `(T+1)` est l'adresse du deuxième élément du tableau et pointe sur `T[1]` qui a les valeurs: `{10,11,12,13}`

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice `T` est le **vecteur** `{0,1,2,3}`, le deuxième élément est `{10,11,12,13}` et ainsi de suite



```

#include<stdio.h>
main()
{
    int T[6][5];
    int *P[6],ligne,col;
    printf("introduire le nombre de ligne ");
    scanf("%d",&ligne);
    printf("introduire le nombre de colonne ");
    scanf("%d",&col);
    P=(int*)0;
    for(i=0;i<ligne;i++)
        for(j=0;j<ligne*col;j++)
            P[i]=T[i][j];
    for(i=0;i<ligne;i++)
        for(j=0;j<col;j++)
            printf("%d ",*(P+i*col+j));
}

```

```

for(i=0;i<ligne;i++)
    for(j=0;j<col;j++)
        printf("%d ",*(P[i]+j));

```

1) Utilisation de pointeur sur un tableau de deux dimensions:

➤ Méthode 1 : utilisation d'un tableau des pointeurs :

Exemple :

Initialisation d'un tableau à deux dimensions.

Pour cet exemple nous avons utiliser un tableau des pointeurs *P[6] chaque pointeur P[i] pointe sur une ligne T[i] .

➤ Méthode 2 :

Dans la mémoire les éléments d'un tableau à deux dimensions sont adjacents, on peut utiliser un pointeur qui pointe sur le premier élément du tableau et ensuite déplacer ce pointeur sur les autres éléments du tableau.

Exemple :

```

void permuter(int X,int Y)
{
    apres l'appel de la fonction permuter A=4, B=5
    temp=X;
    X=Y;
    Y=X;
}
main()
{
    int A,B;
    A=4;
    B=5;
    printf("avant l'appel de la fonction permuter A=%d, B=%d",A,B);
    permuter(A,B);
    printf("\n apres l'appel de la fonction permuter A=%d, B=
%d",A,B);
}

```

Transmission des paramètres d'une fonction

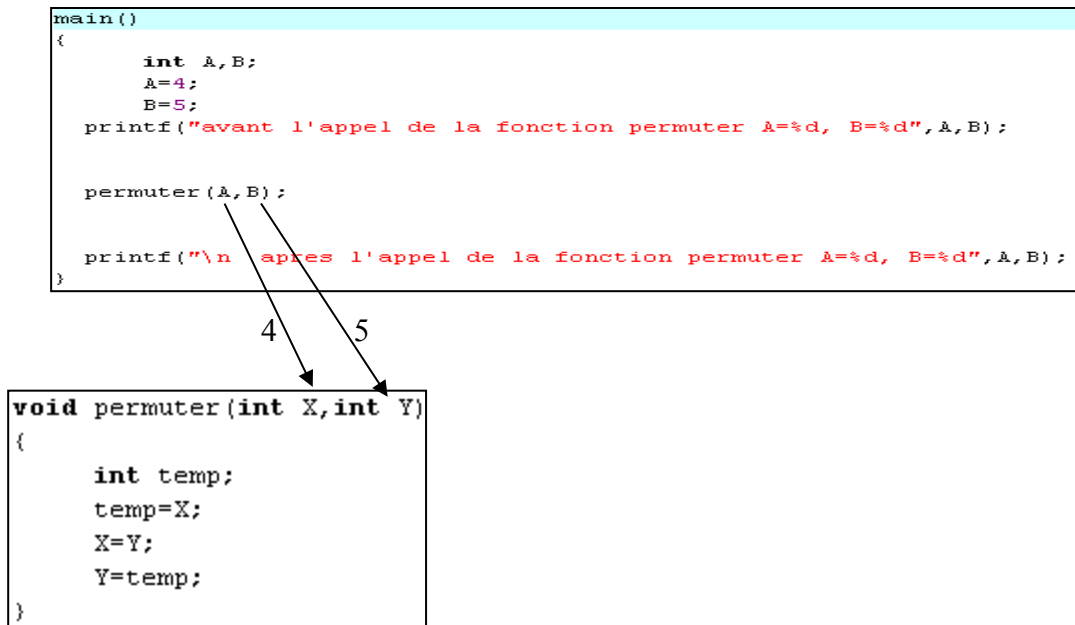
1) Transmission par valeur :

Nous voulons écrire une fonction PERMUTER qui échange le contenu de deux variables du type **int**. Nous écrivons la fonction suivante:

Malgré que la fonction permuter est conçue pour permuter la valeur de deux variables du type int le résultat du programme ne donne pas les valeurs prévues (A=5 et B=4):

Explications :

Lors de l'appel, les *valeurs* de A et de B sont copiées dans les paramètres X et Y. PERMUTER échange bien le contenu des variables *locales* X et Y, mais les valeurs de A et B restent les mêmes on dit que A et B sont transmises par valeur.



2) Transmission par adresse :

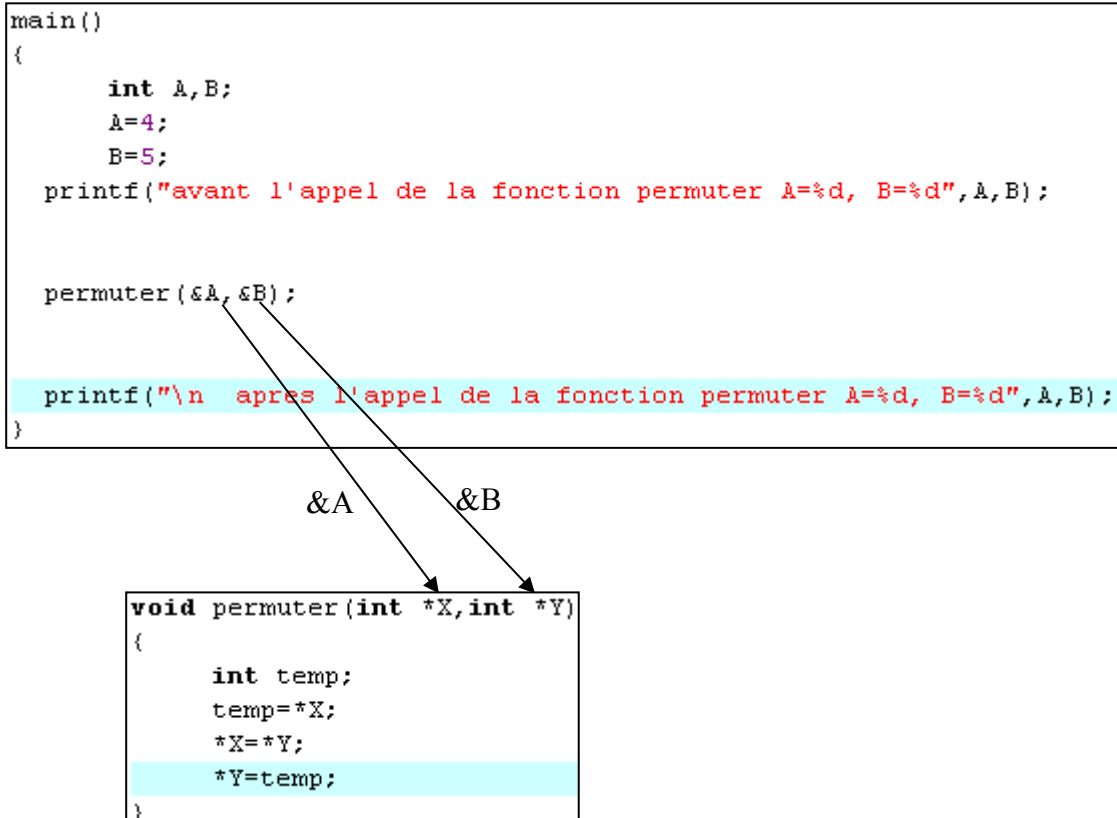
Reprenons le même exemple, pour que la fonction `permuter` puisse échanger les valeurs de `A` et `B` on doit lui renvoyer les adresses de `A` et `B` on dit que `A` et `B` sont transmises par adresse le programme deviendra :

```
void permuter(int *X,int *Y)
{
    int temp;
    temp=*X;
    *X=*Y;
    *Y=temp;
}
main()
{
    int A,B;
    A=4;
    B=5;
    printf("avant l'appel de la fonction permuter A=%d, B=%d",A,B);
    permuter(&A,&B);
    printf("\n apres l'appel de la fonction permuter A=%d, B=
%d",A,B);
}
```

Le résultat du programme est :

avant l'appel de la fonction permuter A=4, B=5

après l'appel de la fonction permuter A=5, B=4



Allocation dynamique de mémoire :

1) Déclaration statique des données :

Chaque variable dans un programme a besoin d'un certain nombre d'octets (= de case) en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre de case à réserver était déjà connu pendant la compilation. Nous parlons alors de la **déclaration statique** des variables.

Exemple

```

main() /*1 case = 1 octet*/
int A ; /*réservation de 4 octets*/
float B[100],somme /*réservation de 8 octets */
short D[400]; /* réservation de 400 octets */
int N; /*réservation de 20 octets*/
printf("le nombre des valeurs à saisir : ");
scanf("%d",&N_element);
for (i=0;i<N_element;i++)
{
printf("\nintroduire la valeur N %d ",i);
scanf("%f",&T[i]);

}
somme=0;
for (i=0;i<N_element;i++)
somme+=T[i];
printf("la somme des valeurs introduites : %f",somme);
}

```

Souvent nous utilisons des données dont ne nous pouvons pas prévoir le nombre, ce qui nous amène à réserver un espace maximale, cela entraîne un gaspillage d'espace mémoire.

Par exemple on veut faire un programme qui calcule la somme des valeurs introduites par l'utilisateur, le nombre des valeurs est déterminé par l'utilisateur :

On charge les valeurs dans un tableau de 100 éléments on suppose que l'utilisateur a introduit 5 comme nombre des valeurs, 95 des espaces de float sont réservés inutilement dans la mémoire.

2) Allocation dynamique :

Un moyen d'éviter ce gaspillage de mémoire est de réserver l'espace mémoire lors de l'exécution du programme cette procédure s'appelle allocation dynamique mémoire.

```

int A ;      sizeof(A) ; /*retourne 4 : variable de type int occupe 4 case
int T[4] ; dans la mémoire */
char B ;     sizeof(B),som; /*retourne 1: variable de type char occupe 1 case
dans la mémoire */;
printf("le nombre de valeurs à saisir ?");
scanf("%d",&N_element);
T=(float*) malloc(sizeof(float)* N_element);
for (i=0;i<N_element;i++)
{
printf("\nintroduire la valeur N %d ",i);
scanf("%f",&T[i]);

}
som=0;
for (i=0;i<N_element;i++)
som+=T[i];
printf("la somme des valeurs introduites : %f",som);
}

```

2.1) La fonction malloc et l'opérateur sizeof

La fonction **malloc** de la bibliothèque `<stdlib>` nous aide à localiser et à réserver de la mémoire au cours d'un programme par exemple `malloc (40)` réserve 40 octets dans la memoire.

La fonction malloc retourne un pointeur générique (void *).

➤ L'opérateur sizeof :

sizeof permet de récupérer la taille d'une variable ou un type de données.

Exemple

Après les déclaration :

Nous obtenons les résultats suivants :

En utilisation de la fonction malloc on peut économiser l'espace mémoire utiliser par le programme précédent de la somme. Le programme deviendra :

Au lieu de déclarer un tableau de nombre des éléments fixes nous avons déclaré un pointeur de type float. En suite la fonction malloc

génère dynamiquement 'N_element' éléments de type float, et retourne l'adresse du premier élément, que le pointeur T recevra. La fonction malloc retourne un pointeur générique (void *), on doit donc le convertir en pointeur de type float (float*).

➤ **La fonction free :**

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque `<stdlib>`.

Par exemple à la fin du programme précédent on peut ajouter l'instruction suivante :

free(T) pour libérer l'espace du tableau.

Remarque :

Si nous ne libérons pas explicitement la mémoire à l'aide de **free**, alors elle est libérée automatiquement à la fin du programme.

```
char NOM [20];  
char PRENOM [20];  
char PHRASE [300];
```

Les chaînes de caractères

En C Une chaîne de caractères est un *tableau à une dimension de caractères* terminée par le caractère '\0' pour indiquer la fin de la chaîne de caractères.

3) Déclaration et initialisation

4) Déclaration

Comme une chaîne de caractères est un tableau de caractères la déclaration est de la forme :

Type nom [longueur] ;

Exemples

Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne.

La représentation interne d'une chaîne de caractères est terminée par le symbole '\0' (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** cases (octets).

Le compilateur C ne contrôle pas si nous avons réservé un octet pour le symbole de fin de chaîne; l'erreur se fera seulement remarquer lors de l'exécution du programme.

5) Initialisation

Le nom d'une chaîne est le représentant de l'adresse du premier caractère de la chaîne. Pour mémoriser une variable qui doit être capable de contenir un texte de N caractères, nous avons besoin de N+1 octets en mémoire:

Exemple: initialisation d'un tableau

```

Char TXT[10] = "FSTT MIPC"; // déclaration, 5 cases dans la mémoire
Char TXT[2] = "FSTT"; // dérivation de 5 cases dans la mémoire
mémoire de la chaîne de caractères "FSTT" */

```

TXT

F	S	T	T		M	I	P	C	\0
---	---	---	---	--	---	---	---	---	----

Cette déclaration est équivalente à :

C'est à dire un tableau de caractères chaque caractère se trouve dans une case du tableau TXT, le dernier caractère est le caractère nul '\0', donc le tableau TXT doit contenir au moins 10 case pour contenir la chaîne de caractères "FSTT MIPC", la dernière case contient le caractère '\0'.

Les déclarations suivantes sont correctes :

Les déclarations suivantes sont incorrectes :

Accès aux éléments d'une chaîne

L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau. En déclarant une chaîne par:

```
char A[6];
```

Nous avons défini un tableau A avec six éléments, auxquels on peut accéder par:

```
A[0], A[1], ... , A[5]
```

Exemple

```
Char TXT[] = "FSTT MIPC" ;
```

TXT

F	S	T	T		M	I	P	C	\0
TXT[0]	TXT[1]	TXT[2]	TXT[3]	TXT[4]	TXT[5]	TXT[6]	TXT[7]	TXT[8]	TXT[9]

```

char NOM[15] = "MEXCE"; // une première ligne."; char LIEU[25];
printf("E%SE"); // à afficher printf("%s", MEXCE); *printf("Entrez le lieu: \n");
puts("Voici une deuxième ligne."); scanf("%s", LIEU);

```

Travailler avec des chaînes de caractères

Les bibliothèques de fonctions de C contiennent une série de fonctions spéciales pour le traitement de chaînes de caractères.

1) Les fonctions de <stdio.h>

La bibliothèque <stdio> nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions **printf** et **scanf** que nous connaissons déjà, nous y trouvons les deux fonctions **puts** et **gets**, spécialement conçues pour l'écriture et la lecture de chaînes de caractères.

➤ Affichage de chaînes de caractères

printf avec le spécificateur de format **%s** permet d'intégrer une chaîne de caractères dans une phrase.

Exemples

La fonction **puts** est idéale pour écrire une chaîne constante ou le contenu d'une variable dans une ligne isolée.

Exemples

➤ Lecture de chaînes de caractères

scanf avec le spécificateur **%s** permet de lire un mot isolé à l'intérieur d'une suite de données du même ou d'un autre type.

Exemple

gets est idéal pour lire une ou plusieurs lignes de texte (p.ex. des phrases) terminées par un retour à la ligne.

Exemple

```
int MAXI = 1000;      char A[] = "Bonjour !"; /* un tableau */
char LIGNE[MAXI];    C = 'e'; B est "une chaîne de caractères constante";
gets(LIGNE);
```

Pointeurs et chaînes de caractères

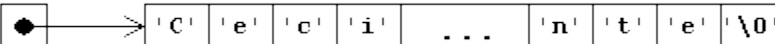
De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être initialisé avec une telle adresse.

1) Pointeurs sur char et chaînes de caractères constantes

➤ Affectation

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur char:

Exemple

C: 

➤ Initialisation

Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *B = "Bonjour !";
```

Attention !

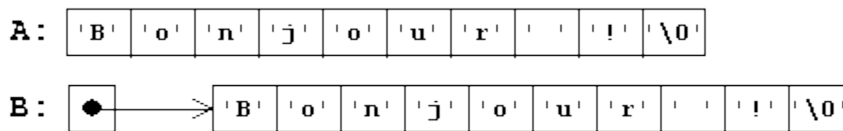
Il existe une différence importante entre les deux déclarations:

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A = B;
```

peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.

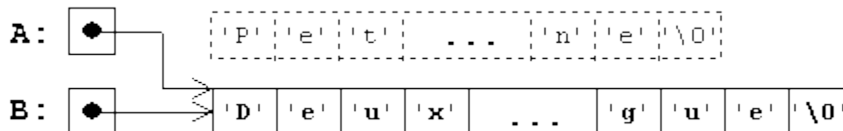


➤ Modification

Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur:

Exemple

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères.

```
#include<stdio.h>
#include<string.h>
main()
{
  char t1[50]="les caract" ;
  char t2[50]= "tères";
  strcat(t1,t2,2);
  puts(t1);
}
/*affiche : les caractè
```

1)

Quelques fonctions prédéfinis en langage C (dans *string.h*)

1) La fonction **strncat**

strncat(but,source,igmax)

Cette fonction recopie un nombre de caractères *igmax* de la seconde chaîne (*source*) à la suite de la première (*but*)

Exemple

2) La fonction **strlen**

strlen(ch)

Fournit une valeur de type *int* correspondant à la longueur de la chaîne *ch*

3) La fonction **strcmp**

strcmp(chaine1,chaine2)

Compare deux chaînes dont on lui fournit l'adresse et elle fournit une valeur entière définie comme étant :

Positive si *chaine1 > chaine2* (c'est à dire si *chaine1* arrive après *chaine2*, au sens de l'ordre défini par le code des caractères).

Nulle si $chaine1=chaine2$ (c'est à dire si ces deux chaînes contiennent exactement les mêmes caractères).

Négative si $chaine1 < chaine2$.

Par exemple :

strcmp("ABC","ABC") est nulle.

strcmp("ABC","ABD") est négative.

4) La fonction ***strcpy***

strcpy(destination,source)

Recopie la chaîne 'source' dans l'emplacement d'adresse destination. Il est nécessaire que la taille du second emplacement soit suffisante pour accueillir la chaîne à recopier. Cette fonction fournit comme résultat l'adresse de la chaîne '*destination*'.

LES FICHIERS

En C, les communications d'un programme avec son environnement se font par l'intermédiaire de fichiers. Pour le programmeur, tous les périphériques, même le clavier et l'écran, sont des fichiers. Jusqu'ici, nos programmes ont lu leurs données dans le fichier d'entrée standard, (c.-à-d.: le clavier) et ils ont écrit leurs résultats dans le fichier de sortie standard (c.-à-d.: l'écran). Nous allons voir dans ce chapitre, comment nous pouvons créer, lire et modifier nous-mêmes des fichiers sur les périphériques disponibles.

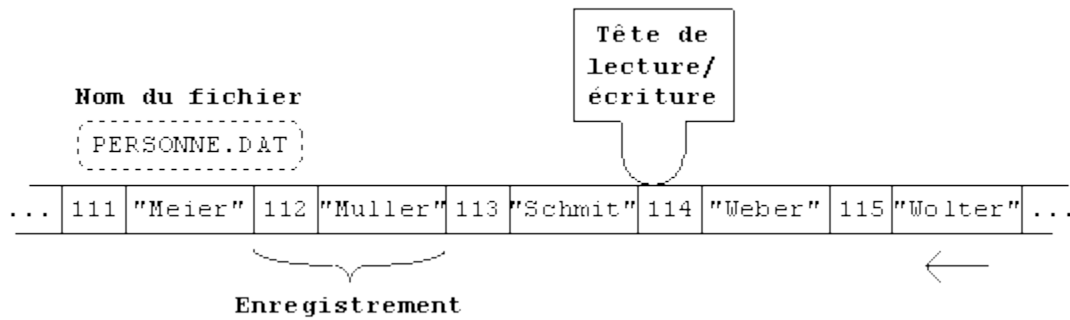
5) Définitions et propriétés

Un **fichier** (angl.: *file*) est un ensemble structuré de données stocké en général sur un support externe (disquette, disque dur, disque optique, bande magnétique, ...). Un **fichier structuré** contient une suite *d'enregistrements* homogènes, qui regroupent le plus souvent plusieurs composantes appartenant ensemble (*champs*).

Fichier séquentiel

Dans des **fichiers séquentiels**, les enregistrements sont mémorisés consécutivement dans l'ordre de leur entrée et peuvent seulement être lus dans cet ordre. Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

En simplifiant, nous pouvons nous imaginer qu'un fichier séquentiel est enregistré sur un support de stockage (disquette, disque dur...):



Les fichiers séquentiels que nous allons considérer dans ce cours auront les propriétés suivantes:

- Les fichiers se trouvent ou bien en état d'écriture ou bien en état de lecture; nous ne pouvons pas simultanément lire et écrire dans le même fichier.
- A un moment donné, on peut uniquement accéder à un seul enregistrement; celui qui se trouve en face de la tête de lecture/écriture.
- Après chaque accès, la tête de lecture/écriture est déplacée derrière la donnée lue en dernier lieu.

1) Accès aux fichiers séquentiels

Il y a 3 étapes pour accéder aux données d'un fichier séquentiel :

1. Ouvrir le fichier en utilisant la fonction **fopen**.
2. Lire ou écrire les données dans le fichier.
3. Fermer le fichier en utilisant la fonction **fclose**.

Exemple1 :

Le programme suivant enregistre des noms des personnes dans un fichier (le nom de fichier est introduit par l'utilisateur)

```

#include <stdio.h>

main()
{
    FILE *P_FICHER;           /* pointeur sur
fichier */
    char NOM_FICHER[30], NOM_PERS[30];
    int C,NB_ENREG;

    /* Créer et remplir le fichier */
    printf("Entrez le nom du fichier à créer : ");
    scanf("%s", NOM_FICHER);
    P_FICHER = fopen(NOM_FICHER, "w") ; /*ouverture de
fichier en
mode
write */
    printf("Nombre d'enregistrements à créer : ");
    scanf("%d", &NB_ENREG);
    C = 0;
    while (C<NB_ENREG)
    {
        printf("Entrez le nom de la personne : ");
        scanf("%s", NOM_PERS);
        fprintf(P_FICHER, "%s\n", NOM_PERS); /*écriture des
donnees sur
le
fichier*/
        C++;
    }
    fclose(P_FICHER);        /*fermeture du
fichier*/

```

2)

2) Le type FILE*

Pour pouvoir travailler avec un fichier, un programme a besoin d'un certain nombre d'informations au sujet du fichier:

- adresse de la mémoire tampon : adresse de la mémoire réservée pour accueillir les données d'un fichier
- position actuelle de la tête de lecture/écriture,
- type d'accès au fichier: écriture, lecture, ...

- état d'erreur,
- ...

Ces informations (dont nous n'aurons pas à nous occuper), sont rassemblées dans une structure du type spécial **FILE**. Lorsque nous ouvrons un fichier avec la commande **fopen**, le système génère automatiquement un bloc du type **FILE** et nous fournit son adresse.

Tout ce que nous avons à faire dans notre programme est:

- déclarer un pointeur du type FILE* pour chaque fichier dont nous avons besoin,
- affecter l'adresse retournée par fopen à ce pointeur,
- employer le pointeur à la place du nom du fichier dans toutes les instructions de lecture ou d'écriture,
- libérer le pointeur à la fin du traitement à l'aide de fclose.

3) Ouvrir et fermer des fichiers séquentiels

Avant de créer ou de lire un fichier, nous devons informer le système de cette intention pour qu'il puisse réserver la mémoire pour la zone d'échange et initialiser les informations nécessaires à l'accès du fichier. Nous parlons alors de l'**ouverture** d'un fichier.

Après avoir terminé la manipulation du fichier, nous devons vider la mémoire tampon et libérer l'espace en mémoire que nous avons occupé pendant le traitement. Nous parlons alors de la **fermeture** du fichier.

L'ouverture et la fermeture de fichiers se font respectivement à l'aide des fonctions **fopen** et **fclose** définies dans la bibliothèque standard `<stdio>`.

3.1) Ouvrir un fichier séquentiel

Lors de l'ouverture d'un fichier avec **fopen**, le système s'occupe de la réservation de la mémoire tampon dans la mémoire centrale et génère les informations pour un nouvel élément du type **FILE**.

L'adresse de ce bloc est retournée comme résultat si l'ouverture s'est déroulée avec succès. La commande **fopen** peut ouvrir des fichiers en écriture ou en lecture en dépendance de son deuxième paramètre ("r" ou "w") :

<FP> = fopen (<Nom> , "w");

ou bien

<FP> = fopen (<Nom> , "r");

- **<Nom>** est une chaîne de caractères constante ou une variable de type chaîne qui représente le nom du fichier sur le support de stockage,
- **le deuxième argument** détermine le mode d'accès au fichier:

"w"	pour 'ouverture en	-
"w"	écriture'	write -
"r"	pour 'ouverture en	-
"r"	lecture'	read -

Si un fichier **existant** est ouvert en mode écriture (w), alors son contenu est perdu. Si un fichier **non existant** est ouvert en mode écriture, alors il est créé automatiquement. Si la création du fichier est impossible alors **fopen** indique une erreur en retournant la valeur zéro.

Dans le cas d'ouverture de fichier en mode lecture (r), le nom du fichier doit être retrouvé dans le répertoire du médium et la tête de lecture/écriture est placée sur le premier enregistrement de ce fichier.

- **<FP>** est un pointeur du type FILE* qui sera relié au fichier.

Dans la suite du programme, il faut utiliser **<FP>** au lieu de

nom de fichier pour le référencer. <FP> doit être déclaré comme:

FILE *FP;

Par exemple l'instruction : ptr= fopen("c:\\donnees.dat","r")

Ouvre le fichier "donnees.dat" qui se trouve dans le chemin "c:\\" en mode lecture et renvoie un pointeur de type FILE sur ce fichier.

il faut indiquer le symbole '\' (back-slash) par '\\', dans le chemin du fichier pour qu'il ne soit pas confondu avec le début d'une séquence d'échappement (p.ex: \n, \t, \a, ...).

Le résultat de fopen

Si le fichier a pu être ouvert avec succès, **fopen** fournit l'adresse d'un nouveau bloc du type **FILE**. En général, la valeur de cette adresse ne nous intéresse pas; elle est simplement affectée à un pointeur <FP> du type **FILE*** que nous utiliserons ensuite pour accéder au fichier.

A l'apparition d'une *erreur* lors de l'ouverture du fichier, **fopen** fournit la valeur numérique zéro qui est souvent utilisée dans une expression conditionnelle pour assurer que le traitement ne continue pas avec un fichier non ouvert (voir exemple 2).

Voici quelques erreurs possibles pouvant empêcher l'ouverture d'un fichier en mode écriture (w) (**fopen** renvoie une valeur nulle):

- chemin d'accès non valide,
- pas de disque/bande dans le lecteur,
- essai d'écrire sur un médium protégé contre l'écriture,
- ...

Les erreurs suivantes peuvent empêcher l'ouverture d'un fichier en mode lecture (r) (**fopen** renvoie une valeur nulle):

- essai d'ouvrir un fichier non existant,
- essai d'ouvrir un fichier sans autorisation d'accès,
- essai d'ouvrir un fichier protégé contre la lecture,
-

4) Fermer un fichier séquentiel

fclose(<FP>);

<FP> est un pointeur du type **FILE*** relié au nom du fichier que l'on désire fermer.

La fonction **fclose** provoque le contraire de **fopen**:

Si le fichier a été ouvert en écriture, alors les données non écrites de la mémoire tampon sont écrites et les données supplémentaires (longueur du fichier, date et heure de sa création) sont ajoutées dans le répertoire du médium de stockage.

Si le fichier a été ouvert en lecture, alors les données non lues de la mémoire tampon sont simplement 'jetées'.

La mémoire tampon est ensuite libérée et la liaison entre le pointeur sur **FILE** et le nom du fichier correspondant est annulée.

Exemple 2 :

En pratique, il faut contrôler si l'ouverture d'un fichier a été accomplie avec succès avant de continuer les traitements. L'exemple suivant traite le cas d'erreur d'ouverture de fichier.

```

#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *P_FICHIER; /* pointeur sur FILE */
    char NOM_FICHIER[30]; /* nom du fichier */
    ...

    printf("Entrez le nom du fichier : ");
    scanf("%s", NOM_FICHIER);
    P_FICHIER = fopen(NOM_FICHIER, "w"); /* ou bien
*/
/* P_FICHIER =
fopen(NOM_FICHIER, "r"); */
    if (!P_FICHIER)
    {
        printf("ERREUR: Impossible d'ouvrir "
            "le fichier: %s.\n", NOM_FICHIER);
        exit(-1); /* Abandonner le
programme en */
    } /* retournant le
code d'erreur -1 */

    ...

    fclose(P_FICHIER);
}

```

3)

5) Lire et écrire dans des fichiers séquentiels

Les fichiers que nous allons employer dans ce cours sont des fichiers texte, c.-à-d. toutes les informations dans les fichiers sont mémorisées sous forme de chaînes de caractères et sont organisées en lignes. Même les valeurs numériques (types **int**, **float**, **double**, ...) sont stockées comme chaînes de caractères.

Pour l'écriture et la lecture des fichiers, nous allons utiliser les fonctions standard **fprintf**, **fscanf**, qui correspondent à **printf**, **scanf**.

➤ Écrire une information dans un fichier séquentiel

Syntaxe :

```
fprintf( <FP>, "<Form>\n", <Expr>);
```

- **<FP>** est un pointeur du type **FILE*** qui est relié au nom du fichier cible.
- **<Expr>** représente un enregistrement dont la valeur est écrite dans le fichier.
- **<Form>** représente le spécificateur de format pour l'écriture dans le fichier.

Attention !

Notez que `fprintf` (et `printf`) écrit toutes les chaînes de caractères sans le symbole de fin de chaîne `'\0'`. Dans les fichiers texte, il faut ajouter le symbole de fin de ligne `'\n'` pour séparer les données.

- **Lire une information dans un fichier séquentiel**

syntaxe :

```
fscanf( <FP>, "<Form>\n", <Adr>);
```

- **<FP>** est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.
- **<Adr>** représente l'adresse de la variable qui va recevoir la valeur de données lu dans le fichier.
- **<Form>** représente le spécificateur de format pour la lecture de données.

Attention !

Pour les fonctions **scanf** et **fscanf** tous les signes d'espacement sont équivalents comme séparateurs. En conséquence, à l'aide de **fscanf**, il nous sera impossible de lire toute une phrase dans laquelle les mots sont séparés par des espaces.

6) Détection de la fin d'un fichier séquentiel

Lors de la fermeture d'un fichier ouvert en écriture, la fin du fichier est marquée automatiquement par le symbole de fin de fichier **EOF**

```
while (!feof(FP))
{
    fscanf(FP, "%s\n", NOM);
    ...
}
```

(End Of File). Lors de la lecture d'un fichier, la fonction `feof(<FP>)` nous permet de détecter la fin du fichier:

```
feof( <FP> );
```

feof retourne une valeur différente de zéro, si la tête de lecture du fichier référencé par <FP> est arrivée à la fin du fichier; sinon la valeur du résultat est zéro.

<FP> est un pointeur du type **FILE*** qui est relié au nom du fichier à lire.

Pour que la fonction **feof** détecte correctement la fin du fichier, il faut qu'après la lecture de la dernière donnée du fichier, la tête de lecture arrive jusqu'à la position de la marque **EOF**. Nous obtenons cet effet seulement si nous terminons aussi la chaîne de format de **fscanf** par un retour à la ligne '\n' (ou par un autre signe d'espacement).

Exemple

Une boucle de lecture typique pour lire les enregistrements d'un fichier séquentiel référencé par un pointeur FP peut avoir la forme suivante:

Le programme suivant lit et affiche sur l'écran le contenu du fichier "C:\AUTOEXEC.BAT" :

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *FP;
    Char ch[100];
    FP = fopen("C:\\\\AUTOEXEC.BAT", "r");
    if (!FP)
    {
        printf("Impossible d'ouvrir le fichier\n");
        exit(-1);
    }
    while (!feof(FP))
    {
        fscanf(FP, "%s", ch);
        puts(ch);
    }
    fclose(FP);
}
```

7) Mise à jour d'un fichier séquentiel en C

Dans ce chapitre, nous allons résoudre les problèmes standards sur les fichiers, à savoir:

- l'ajoute d'un enregistrement à un fichier
- la suppression d'un enregistrement dans un fichier
- la modification d'un enregistrement dans un fichier

Comme il est impossible de lire et d'écrire en même temps dans un fichier séquentiel, les modifications doivent se faire à l'aide d'un fichier supplémentaire. Nous travaillons donc typiquement avec au moins deux fichiers: l'ancien fichier ouvert en lecture et le nouveau fichier ouvert en écriture:

7.1) Ajouter un enregistrement à un fichier

Nous pouvons ajouter le nouvel enregistrement à différentes positions dans le fichier:

- **Ajoute à la fin du fichier**

L'ancien fichier est entièrement copié dans le nouveau fichier, suivi du nouvel enregistrement.

➤ **Ajoute au début du fichier**

L'ancien fichier est copié derrière le nouvel enregistrement qui est écrit en premier lieu.

➤ **Insertion dans un fichier trié relativement à une rubrique commune des enregistrements**

Le nouveau fichier est créé en trois étapes:

1. Copier les enregistrements de l'ancien fichier qui précèdent le nouvel enregistrement,
2. Ecrire le nouvel enregistrement,
3. Copier le reste des enregistrements de l'ancien fichier.


```
struct enreg
{
    int numero ;
    int qte;
    float prix;
}
```

Les structures

Nous avons déjà vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.

La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nommé "champ") se fera, non plus par un indice de position, mais par son nom au sein de la structure.

1. Déclaration d'une structure

Voyons tout d'abord cette déclaration :

Celle-ci définit un modèle de structure mais ne réserve pas de « variables » correspondant à cette structure. Ce modèle s'appelle ici *enreg* et il précise le nom et le type de chacun de « champs » constituant la structure (numero, qte, prix).

Une fois un tel modèle défini, nous pouvons déclarer des variables de type correspondant par exemple :

```
struct enreg art ;
```

Réserve un emplacement nommé "art" de type *enreg* destiné à contenir deux entiers et un flottant.

De manière semblable :

```
struct enreg art1,atr2 ;
```

Réserverait deux emplacements art1, art2 du type *enreg*.

```

struct enreg
{
    int numero ;
    int qte;
    float prix;
} art1, art2;

```

Remarque : il est possible de regrouper la définition de la structure et déclaration de type de variable dans une seule instruction comme pour cet exemple :

2. Utilisation d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable par le nom du champ séparé par point "."

Voici quelque exemple utilisant le modèle *enreg* et les variables *art1*, *art2* déclarés de ce type.

```
art1.numero=15 ;
```

Affecte la valeur 15 au champ *numero* de la structure *art1*.

```
printf("%f", art2.prix) ;
```

Affiche la valeur du champ *prix* de la structure *art2*.

```
art1.qte++ ;
```

Incrémente de 1 la valeur du champ *qte* de la structure *art1*.

Remarque : la priorité de l'opérateur '.' est très élevée, de sorte qu'aucune expression ci-dessus ne nécessite des parenthèses.

Il est possible d'affecter le contenu d'une structure à une autre structure défini à partir de même modèle, nous pourrons écrire :

```
art1=art2 ;
```

3. Initialisation de structures :

Il est possible d'initialiser explicitement une structure, lors de sa déclaration.

<pre>struct enreg { int numero ; int qte; float prix; } ; typedef struct enreg s_enreg ; s_enreg art1,art2[20] ;</pre>	<pre>typedef struct { int numero ; int qte; float prix; int t[3]; }s_enreg; s_enreg art1,art2;</pre>
--	--

//art2[20] est un tableau de structures

Voici un exemple d'initialisation de notre structure art1 au moment de sa déclaration:

```
struct enreg art1={100,285,2000} ;
```

Pour simplifier la déclaration de types :

La déclaration typedef permet de définir ce que l'on nomme en langage C des « types synonymes ». A priori, elle s'applique à tous les types et pas seulement aux structures.

La déclaration :

```
typedef int entier;
```

Signifie que *entier* est synonymes de *int* de sorte que les déclarations suivantes sont équivalents :

```
int n,p ;           entier n,p ;
```

de même :

```
typedef float reel ;
```

Signifie que *reel* est synonyme de *float* les déclarations suivantes sont équivalents :

```
float x,y           reel x,y
```

en faisant usage de *typedef*, les déclarations des structures art1 et art2 peuvent être réalisées comme suit :

Ou encore

Exemple

Ecrire une structure complexe dont les membres sont un réel et un imaginaire, puis écrire un programme qui calcul la norme d'un nombre complexe.

```
#include<stdio.h>
#include<math.h>
struct complexe{
    double reelle;
    double imaginaire;
};
main()
{
    struct complexe z;
    double norme;
    printf("entrer un reel et un imaginaire:\n");
    scanf("%lf%lf",&z.reelle,&z.imaginaire);
    norme=sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f\n",z.reelle,z.imaginaire,norme);
    fflush(stdin);
    getchar();
}
```

4. Pointeurs sur des structures

Comme pour les types de base, on peut utiliser des « pointeurs sur structure » :

Etudiant e, *pe ; pe est un pointeur vers une structure Etudiant, le pointeur doit être initialisé par l'adresse de la structure

pe = &e ;

⇒ pe contiendra l'adresse du premier champ de la structure

Utilisation du symbole -> pour accéder aux champs *dans le cas des pointeurs* :

pe->CNE = 12 ;

Ou

(*pe).CNE = 12 ;

Exercice d'application

Ecrire un programme qui permet de stocker le nom, prénom, Cin et la date de naissance dans un tableau, puis afficher un enregistrement suivant l'index entré par l'utilisateur.

NB:

Utiliser une structure nommée élève.

Un pointeur de structure nommé classe

Un tableau nommé tab

La taille de ce tableau sera déterminé dynamiquement.

Solution avec l'utilisation de _

```
#include<stdlib.h>
#include<stdio.h>
struct eleve{
    char nom[20];
    int date; };
typedef struct eleve *classe;
main()
{ int n, i;
  classe tab;
  printf("nombre d'eleves de la classe = ");
  scanf("%d",&n);
  tab = (classe)malloc(n * sizeof(struct eleve));
  for (i = 0 ; i < n; i++)
```

```

{          printf("\n saisie de l'eleve numero %d\n",i);
           printf("nom de l'eleve = ");
           scanf("%s",&tab[i].nom);
           printf("\n date de naissance JJMMAA = ");
           scanf("%d",&tab[i].date);
}
printf("\n Entrez un numero  ");
scanf("%d",&i);
printf("\n Eleve numero %d : ",i);
printf("\n nom =%s",tab[i].nom);
printf("\n date de naissance = %d\n",tab[i].date);
free(tab);
}

```

Solution avec ->

```

#include<stdlib.h>
#include<stdio.h>
struct eleve{
    char nom[20];
    int date; };
typedef struct eleve *classe;
main()
{ int n, i;
  classe tab;
  printf("nombre d'eleves de la classe = ");
  scanf("%d",&n);
  tab = (classe)malloc(n * sizeof(struct eleve));
  for (i = 0 ; i < n; i++)
  {          printf("\n saisie de l'eleve numero %d\n",i);
             printf("nom de l'eleve = ");
             scanf("%s",&tab[i].nom);

```

```

        printf("\n date de naissance JJMAAA = ");
        scanf("%d",&tab[i].date);
    }
    printf("\n Entrez un numero  ");
    scanf("%d",&i);
    printf("\n Eleve numero %d : ",i);
    printf("\n nom =%s",(tab +i)->nom);
    printf("\n date de naissance = %d\n",(tab + i)->date);
    free(tab);
}

```

5. Structures et fonctions

Une variable de type structure peut être passée en entrée d'une fonction et/ou renvoyée en sortie.

Une structure peut être passée, comme une autre variable, par valeur ou par adresse

Mieux vaut passer par un pointeur sur la structure si cette dernière contient beaucoup de champs.

Le passage de la structure dans des paramètres de la fonction se fait soit par valeurs ou par adresses.

Exemples de déclarations

void Par_valeur(struct Membre m);

void Par_reference(struct Membre *m);

Lors de l'appel

Par_valeur(m);

Par_reference(&m);

Exemple d'une fonction

*void affiche_membre (struct Membre *p)*

```

{
    printf("nom = %s\n", p->nom);

```

```

printf("adresse = %s\n", p->adresse);
printf("numéro de membre = %d\n", p->numero);
printf("\nDate d'affiliation %d/%d/%d\n",
      p->creation.jour, p->creation.mois, p->creation.an);
}

```

6. Structures imbriquées

Imbrication d'une structure dans une autre structure :

```

struct date {
    int jour, mois, annee;
};
typedef struct date Date ;
struct etudiant {
    int CNE ;
    char nom[80], prenom[80];
    Date date_naissance ;
};
typedef struct etudiant Etudiant ;

```

On peut toujours initialiser à la déclaration :

```
Etudiant e = {70081, "Bush", "Georges", {12, 06, 1978} } ;
```

Pour l'accès aux champs, double indirection :

```

printf("L'étudiant %s est né le %d / %d / %d\n", e.nom,
      e.date_naissance.jour, e.date_naissance.mois,
      e.date_naissance.annee) ;

```

7. Structures auto-référentes

Utilisation d'un champ qui soit du même type que la structure :

```

struct etudiant {
    int CNE ;
    char nom [80], prenom [80] ;

```



```
    struct etudiant binome ;  
};
```

⇒ **INCORRECT** (le compilateur ne connaît pas la taille mémoire à réserver pour le champ binôme)

Utilisation de l'adresse :

```
struct etudiant {  
    int CNE ;  
    char nom [80], prenom [80] ;  
    struct etudiant *binome ;  
};
```