Support de cours:
Programmation C

Réalisé par:
Mohamed Elhoucine ELHDHILI
Hela Boucetta

Table des matières

Chapitre 1 : Introduction Générale au langage C

Objectifs

- *Connaître quelques termes informatiques utiles*
- *Avoir une idée sur les langages de programmation*
- *Initiation au langage C*

Eléments de contenu

- *Terminologie informatique*
- *Démarche à suivre pour résoudre un problème*
- *Les langages de programmation*
- *Présentation générale du langage C*

1. Terminologie informatique :

- ***Information*** : *tout* ensemble de données qui a un sens (textes, nombres, sons, images, vidéo, instructions composant un programme…). Toute information est manipulée sous forme *binaire* (ou numérique) par l'ordinateur.

- ***Informatique*** : Science du traitement automatique de l'information.

- ***Ordinateur*** : machine de traitement de l'information (acquérir, conserver, restituer et effectuer des traitements sur les informations).

- ***Programme*** : suite d'instructions élémentaires, qui vont être exécutées dans l'ordre par le processeur. Ces instructions correspondent à des actions très simples, comme additionner deux nombres, lire ou écrire une case mémoire, etc.

- ***Système informatique*** : ensemble de moyen matériel et logiciels (programmes) pour satisfaire les besoins informatiques des utilisateurs.

- ***Système d'exploitation*** : ensemble de programmes pour servir d'interface entre l'utilisateur et la machine. S'occupe de gérer les différentes ressources de la machine et de ses périphériques.

- ***Démarche à suivre pour résoudre un problème :***
  - compréhension du problème
  - spécification : quoi faire ?
  - conception : comment faire ? (algorithmes)
  - codage (langage de programmation).
  - test et validation.
  - maintenance.

- ***Programmation*** : consiste à écrire une suite d'instruction dans un langage compréhensible par un ordinateur.

- ***Algorithme*** : successions d'action destinée à résoudre un problème en un nombre fini d'instructions.

- ***Langage de programmation*** : langage structuré sans ambiguïté utilisé pour décrire des actions (ou algorithmes) exécutables par un ordinateur. Historiquement, le premier langage informatique a été l'assembleur. Or, la programmation en assembleur est souvent fastidieuse et dépend étroitement du type de machine pour lequel il a été écrit. Si l'on désire l'adapter à une autre machine ("porter" le programme), il faut le réécrire entièrement. C'est pour répondre à ces problèmes qu'ont été développés dès les années 50 des langages de plus haut niveau. Dans ces langages, le programmeur écrit selon des règles strictes, mais dispose d'instructions et de structures de données plus expressives qu'en assembleur. Par exemple, dans certains langages comme MATLAB, on pourra écrire en une ligne que l'on désire multiplier deux matrices, alors que le programme correspondant en assembleur prendrait quelques centaines de lignes.

- ***Les principaux langages :***

Les principaux langages compilés sont :
- C/C++ : programmation système et scientifique ;
- ADA : logiciels embarqués ;
- Cobol : gestion ;
- Fortran : calcul scientifique ;
- Pascal : enseignement.

Quelques langages interprétés :
- BASIC : bricolage ;
- LISP : "Intelligence Artificielle" ;
- Prolog : idem ;
- Perl : traitement de fichier textes ;
- Python : programmation système, Internet ;
- Java "applets" : Internet ;
- MATLAB : calcul scientifique ;
- Mathematica : idem.

- ***Compilation, édition de liens et chargement*** : Un programme écrit en langage évolué (sous forme de code source) doit subir ces trois étapes pour être exécuté par un ordinateur. Le compilateur est un programme qui transforme un module en code source en un module en code objet (code machine). L'éditeur de lien (programme) essaye de mettre ensemble les différents modules d'un programme. Le chargeur s'occupe d'amener en mémoire centrale un programme complet et prêt à être exécuté.

2. Introduction au langage C

2.1. Historique

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson chercheurs aux Bell Labs afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978 Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre « The C

Programming language ». Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En l983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI-C. Celle ci fut reprise telle qu'elle par l'ISO (International Standards Organization).

2.2. Présentation générale du langage C

- ***Contenu du langage***

En allant du plus simple au plus élaboré, on trouve en C :

- des instructions de définition, de traitement, de structuration. Comme dans les langages classiques, on utilise l'affectation, le test, la boucle…
- des fonctions. Elles rassemblent aux sous-programmes, fonction procédures et sous routines bien connues, mais elles portent les caractéristiques spécifiques du C.
- un pré processeur qui permet d'apporter des perfectionnements à la présentation des programmes.
- Un compilateur, un éditeur de lien, une bibliothèque et les utilitaires permettant de la gérer. La bibliothèque est une riche collection de fonctions à caractère utilitaire. Beaucoup de ces fonctions sont utilisable comme des macro-instructions

- ***La compilation***

Le C est un langage compilé (par opposition aux langages interprétés). Cela signifie qu'un programme C est décrit par un fichier texte, appelé fichier source. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur. La compilation se décompose en fait en 4 phases successives :

- Le traitement par le pré processeur : le fichier source est analysé par le pré processeur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source . . .).

- La compilation : la compilation proprement dite traduit le fichier généré par le pré processeur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.

- L'assemblage : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet.

- L'édition de liens : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit exécutable.

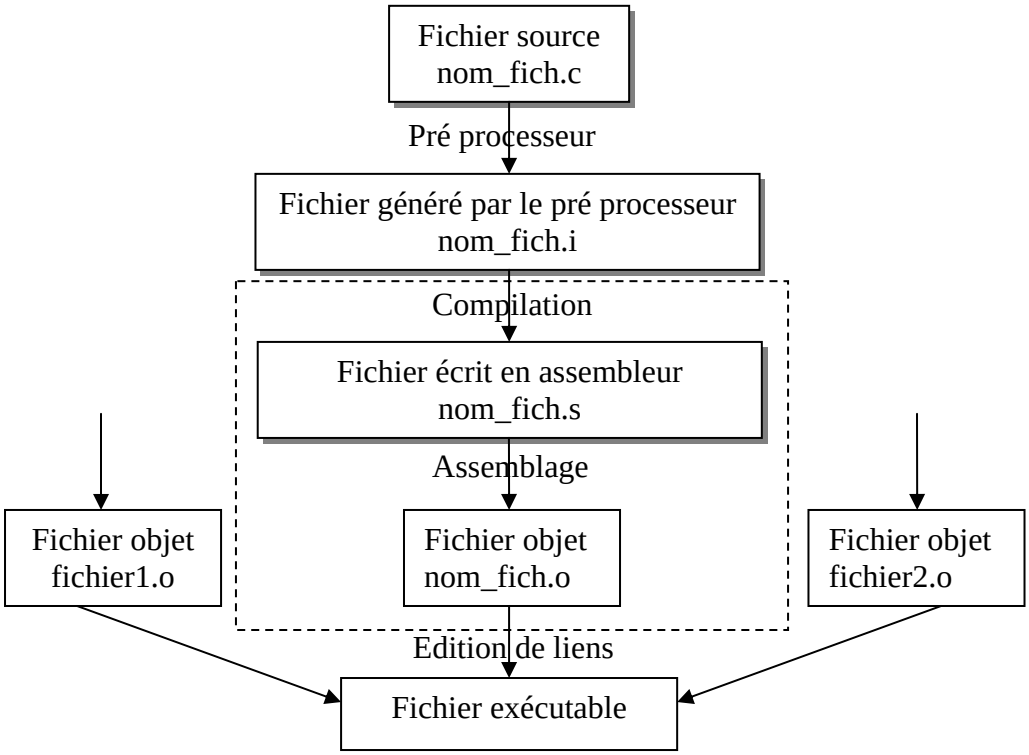


Figure 1.1 : étapes de création d'un exécutable à partir d'un code écrit en C

- ***caractéristique du langage C*** :

Le langage C est un langage évolué et structuré, assez proche du langage machine destiné à des applications de contrôle de processus (gestion d'entrées/sorties, applications temps réel ...). Les compilateurs C possèdent les taux d'expansion les plus faibles de tous les langages évolués (rapport entre la quantité de codes machine générée par le compilateur et la quantité de codes machine générée par l'assembleur et ce pour une même application);

Le langage C possède assez peu d'instructions, il fait par contre appel à des bibliothèques, fournies en plus ou moins grand nombre avec le compilateur.

Exemples: math.h : bibliothèque de fonctions mathématiques
stdio.h : bibliothèque d'entrées/sorties standard

On ne saurait développer un programme en C sans se munir de la documentation concernant ces bibliothèques.

Chapitre 2 : Structure d'un programme C

Objectifs

- *Connaître la structure d'un programme C*
- *Connaître les composantes élémentaires du C*

Eléments de contenu

- *Composants d'un programme C*
- *Les composantes élémentaires du C*
- *Les Types de données*
- *La déclaration de variables*

1. Structure d'un programme C

Un programme en C comporte :

- ***Un entête*** (header) constitué de méta instructions ou **directives** destinées au pré processeur.
- ***Un bloc principal*** appelé **main ().** Il contient des instructions élémentaires ainsi que les appels aux modules (fonctions) considérés par le compilateur comme des entités autonomes. Chaque appel à une fonction peut être considéré comme une instruction.
- ***Le corps*** des fonctions placées avant ou après le main () dans un ordre quelconque, les une après les autres. Partout, les variables et les fonctions font l'objet d'une déclaration précisant leur type. Le schéma général est donc :

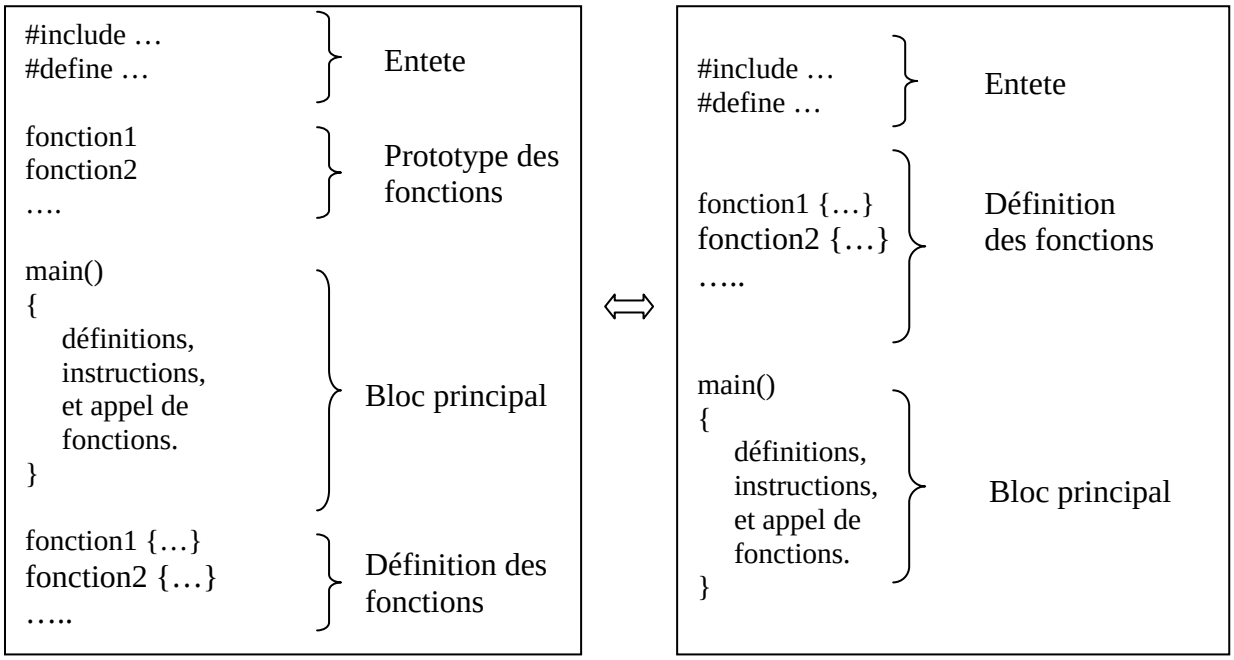


Figure 2.1 : structure d'un programme C

Exemple : calcul de la surface d'un cercle.

Tableau 1 : calcul de la surface d'un cercle

```
#include <stdio.h>
#define PI 3.14

float f_surface(float rayon)
{float s;
 s=rayon*rayon*PI;
 return(s);
}

void main()
{
float surface;
surface=f_surface(2.0);
printf("%f\n",surface);
}
```

```
#include <stdio.h>
#define PI 3.14

float f_surface(float);
void main()
{
float surface;
surface=f_surface(2.0);
printf("%f\n",surface);
}
float f_surface(float rayon)
{float s;
 s=rayon*rayon*PI;
 return(s);
}
```

2. Les composants élémentaires du C :

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- les identificateurs
- les mots-clefs
- les constantes
- les chaînes de caractères,
- les opérateurs
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui ne sont pas considérés par le pré processeur.

2.1. Les identificateurs

- Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut designer :

  - un nom de variable ou de fonction,
  - un type défini par typedef, struct, union ou enum,
  - une étiquette.

- Un identificateur est une suite de caractères parmi les lettres (minuscules ou majuscules, mais non accentuées), les chiffres, le tirait bas.
- Le premier caractère d'un identificateur ne peut pas être un chiffre.
- Les majuscules et minuscules sont différenciées.
- Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le

standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

2.2. Les mots-clefs

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI-C compte 32 mots clefs :

auto const double float int short struct unsigned
break continue else for long signed switch void
case default enum goto register sizeof typedef volatile
char do extern if return static union while

Ces mots peuvent être rangé en catégories :

Tableau 2.2 : mots-clefs du langage C

les spécificateurs de stockage	auto register static extern typedef
les spécificateurs de type	char double enum float int long short signed struct union unsigned void
les qualificateurs de type	const volatile
les instructions de contrôle	break case continue default do else for goto if switch while
divers	return sizeof

2.3. Les commentaries

Un commentaire débute par /* et se termine par */. Par exemple,

/* Ceci est un commentaire */

On ne peut pas imbriquer des commentaires.

2.4. Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes vont être utilisées, par exemple, pour l'initialisation de variables.

2.4.1. Les constantes entières

Une constante entière peut être représentée de 3 manières différentes suivant la base dans laquelle elle est écrite :

- décimale : par exemple, 0 et 2437 sont des constantes entières décimales.
- octale : la représentation octale d'un entier correspond à sa décomposition en base 8. Les constantes octales doivent commencer par un zéro. Par exemple, les représentations octales des entiers 0 et 255 sont respectivement 00 et 0377.
- hexadécimale : la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de a/A à f/F sont utilisées pour représenter les nombres de 10 à 15. Les constantes hexadécimales doivent commencer par 0x ou 0X. Par exemple, les représentations hexadécimales de 14 et 255 sont respectivement 0xe et 0xff.

Remarque :

On peut spécifier explicitement le format d'une constante entière en la suffixant par u ou U pour indiquer qu'elle est non signée, ou en la suffixant par l ou L pour indiquer qu'elle est de type long. Par exemple :

Tableau 2.3 : exemples de constantes entières

constante	type
1234	Int
02154	Int
0x4d2	Int
123456789L	Long
1234U	Unsigned int
124687564UL	Unsigned long int

Exemples :

```
int Nb_lignes ; /* déclaration d'une variable de type entier */
Nb_lignes = 15 ; /* initialisation de cette variable à 15 */
#define TEL  75410452
const  annee = 2002 ; /* affecte une valeur à une variable, cette valeur ne pourra
plus être changé par la suite */
```

2.4.2. Les constantes réelles

Ils sont de l'une des deux formes suivantes :

- En notation décimale : [+-]m.n ou m et n sont des entiers. Exemple : 2.5 -123.47 ….
- En notation scientifique (mantisse et exposant) : [+-]m.n [e/E][+-]p ou m,n et p sont des entiers. Exemple : -14.5 e-2 (-14.5 10^{-2})

Par défaut, une constante réelle est représentée avec le format du type double. On peut cependant influer sur la représentation interne de la constante en lui ajoutant un des suffixes f/F ou l/L. Les suffixes f et F forcent la représentation de la constante sous forme d'un float, Les suffixes l et L forcent la représentation sous forme d'un long double.

Tableau 2.4 : Exemples de constantes réelles

constante	type
12.34	Double
12.3 e-4	Double
12.34L	Long double
12.34F	Float

2.4.3. Les constantes caractères

Elles sont de l'une de l'une des formes suivantes :

- 'x' désigne le caractère imprimable x sauf l'apostrohe et l'antislash et les guillemets
- '\nnn' désigne le nombre de code octal nnn (nnn est un nombre octal)
- '\\' désigne le caractère \ (ou encore '\134' en ASCII)
- '\'' désigne le caractère ' (ou encore '\47' en ASCII)
- '\ " ' désigne le caractere "
- '\b' désigne le caractère <backspace> retour arrière
- '\f' désigne le caractère <formfeed> saut de page
- '\n' désigne le caractère <newline> nouvelle ligne
- '\r' désigne le caractère <return> retour chariot
- '\t' désigne le caractère <tab> tabulation horizontale

- '\v' désigne le caractère de tabulation verticale

2.4.4. Les constantes chaînes de caractères

- Une constante chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple,"Ceci est une chaîne de caractères".
- Elle peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple, "ligne 1 \n ligne 2"
- A l'int'erieur d'une chaîne de caractères, le caractère " doit être désigné par \". Enfin, le caractère \ suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple :

```
"ceci est une longue longue longue longue longue longue longue longue \
chaîne de caractères"
```

Remarques :

- pour connaître la fin de la chaîne, le compilateur fait suivre le dernier caractère par le caractère nul \0.
- Il faut savoir distinguer le constant caractère de la constante chaîne de caractère. Ainsi 'x' et "x" ne sont pas équivalents. Le premier représente la valeur numérique du caractère x, le second est une chaîne ne comprenant qu'un caractère (la lettre x) en plus \0 final.

3. Les types de données

En C, il n'existe que quelques types fondamentaux de données :

- Le type **char :** un seul byte représentant un caractère.
- Le type **int :** un nombre entier dont la taille corresponds à celles des entiers du SE.
- Le type **float :** un nombre en virgule flottante en simple précision.
- Le type **double :** un nombre en virgule flottante en double précision.

Des qualificatifs peuvent préciser le type int : **short, long, unsigned, signed**. Les qualificatifs **signed** et **unsigned** peuvent préciser le type char. Le qualificatif **long** peut préciser le type double. Quand une declaration ne précise pas le type de base, **int** est supposé.

Exemples :

```
short int x ;
unsigned long y ; /* int implicite */
Long double z ;
unsigned char ch ;
```

4. Déclaration de variables

Les variables doivent toutes être déclarées avant d'être utilisées bien que certaines soient faîtes implicitement par le contexte. A une variable correspond un espace en mémoire et un mécanisme permettant d'adresser cet emplacement. En C une variable est caractérisé par :

- Son nom (un identificateur)
- Son type (type de base ou type définie par l'utilisateur)
- Sa classe d'allocation (extern, static, auto, register)

Une déclaration de variables a la forme :

classe d'allocation	**type**	**liste d'identificateurs**

Exemples :

```
extern int var_globale1, var_globale2;
static float  var_statique ;
auto int i,j,k ;
register char caractere1, caractere2  ;
```

La classe d'allocation par défaut est auto.

La classe d'allocation d'un objet spécifie le type d'espace mémoire où l'emplacement correspondant à cet objet sera alloué, sa durée de vie et sa visibilité. Il existe 4 classes d'allocation en C :

- Les variables "externes" (extern) : l'espace correspondant est alloué en zone de données statique dès la compilation. Ce type de spécification permet de partager des variables entre des fichiers sources différents.
- Les variables "statiques" (static) : l'espace est alloué en zone de donnée statique dès la compilation. Ce type d'allocation permet de définir des variables globales à un fichier (en dehors de toutes fonction mais accessible uniquement dans les fonctions définies dans le fichier source ou elles sont déclarées) et des variables locale à des fonctions (ou à des blocs) telles qu'à toute exécution du bloc l'adresse de la variable reste la même. La spécification *static* peut être appliqué à une fonction, dans ce cas la fonction n'est appelable que par des fonctions définies dans le même fichier source.
- Les variables "automatiques" (auto) ; l'espace est alloué dynamiquement (zone dynamique dans la pile) à chaque appel de fonction ou entrée dans le bloc correspondant
- Les variables "registres" (register) : l'espace associé est, dans la mesure du possible, un registre du processeur. Ce type d'allocation n'est possible que pour des types simples (caractère, entier..).

Une déclaration indique le nom, le type (parfois la valeur) et regroupe derrière une ou plusieurs variables de même type.

On peut initialiser les variables lors de la déclaration :

Exemples :

```
char backslash='\\' ;
int i=0 ;
float eps=1.0 e-5 ;
```

Si la variable est externe ou statique, l'initialisation est faite une fois pour toutes avant que le programme ne commence l'exécution.

Les variables "automatiques" dont l'initialisation est explicite seront initialisées à chaque fois qu'on appellera la fonction dont elle font partie. Les variables automatiques qui ne sont pas initialisées explicitement ont des valeurs indéfinies. Les variables externes et statiques sont mises à zéro par défaut.

Chapitre 3 : La syntaxe du langage C (les opérateurs)

Objectifs

- *Connaître les différents opérateurs du C*

Eléments de contenu

- *L'affectation*
- *Les opérateurs arithmétiques*
- *Les opérateurs relationnels*
- *Les opérateurs logiques booléens*
- *Les opérateurs logiques bit à bit*
- *Les opérateurs d'affectation composés*
- *Les opérateurs d'incrémentation et de décrémentation*
- *L'opérateur virgule*
- *L'opérateur conditionnel ternaire*
- *L'opérateur de conversion de types*
- *L'opérateur adresse*
- *Règles de priorité des opérateurs*

1. L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe =. Sa syntaxe est la suivante :

variable = expression ;

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau, une structure…etc. Cette expression a pour effet d'évaluer « expression » et d'affecter la valeur obtenue à « variable ». De plus, cette expression possède une valeur, qui est celle de « expression ». Ainsi, l'expression i = 5 vaut 5.

L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant :

```
main()
{
int i, j = 2;
float x = 2.5;
i = j + x;
x = x + i;
printf(" %f ",x);
 }
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction i = j + x;, l'expression j +x a été convertie en entier.

2. Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires

+ addition
- soustraction
* multiplication
/ division
% reste de la division (modulo) (pour les entiers)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple :

```
float x;
x = 3 / 2;
affecte à x la valeur 1. Par contre
x = 3 / 2.0;
/* affecte `a x la valeur 1.5. */
```

- L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

- Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction pow(x,y) de la librairie math.h pour calculer x^y

3. Les opérateurs relationnels

> Strictement supérieur
>= supérieur ou égal
< Strictement inférieur
<= inférieur ou égal
== égal
!= différent

Leur syntaxe est **: expression1 op expression2**

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut 0 si la condition est faux, et une valeur différente de 0 sinon. Attention à ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =. Ainsi, le programme

```
main()
{
   int a = 0;
   int b = 1;
  if (a = b)
```

```
        printf("a et b sont égaux");
    else
        printf("a et b sont differents");
}
```

imprime à l'écran : a et b sont égaux

4. Les opérateurs logiques booléens

&& et logique
|| ou logique
! négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 0 si la condition est faux.

Dans une expression de type : expression1 op1 expression2 op2 ...expressionN , l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i,j;
if ((i >= 0) && (i <= 9) && !(j == 0))
```

la dernière clause ne sera pas évaluée si **i** n'est pas entre 0 et 9.

5. Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

& et (a&b=1 ssi a=1 et b=1)
| ou inclusif (a|b=1 ssi a=1 ou b=1), ici OU est dit inclusif car si a=b=1 alors a|b=1
^ ou exclusif (a^b=1 ssi b=1 ou a=1)
~ complément à 1
<< décalage à gauche
>> décalage à droite

En pratique, les opérateurs &, | et ~ consistent à appliquer bit à bit les opérations suivantes

&	0	1
0	0	0
1	0	1

a.b

|	0	1
0	0	1
1	1	1

a+b (a ou b)

^	0	1
0	0	1
1	1	0

$a.\overline{b} + \overline{a}.b$

L'opérateur unaire ~ change la valeur de chaque bit d'un entier. Le décalage à droite et à gauche effectue respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît). Considérons par exemple les entiers a=77 et b=23 de type unsigned char (i.e. 8 bits). En base 2 il s'écrivent respectivement 01001101 et 00010111.

Tableau 3.1 : les opérateurs logiques bit à bits

valeur	binaire	décimale	remarques
a	01001101	77	
b	00010111	23	
a&b	00000101	5	
a|b	01011111	95	
a^b	01011010	90	
~ a	10110010	178	
b << 2	01011100	92	Multiplication par 4
b << 5	11100000	112	Ce qui dépasse disparaît
b >> 1	00001011	11	Division entière par 2

6. Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

+= -= *= /= %= &= ^= |= <<= >>=

Pour tout opérateur **op**, l'expression :

expression1 **op=** expression2 ⇔ expression1 = expression1 **op** expression2

Toutefois, avec l'affection composée, expression1 n'est évaluée qu'une seule fois.

7. Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en suffixe (i++) qu'en préfixe (++i). Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple :

```
int a = 3, b, c;
b = ++a; /* a et b valent 4 */
c = b++; /* c vaut 4 et b vaut 5 */
```

8. L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

```
expression1, expression2, ... , expressionN
```

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme

```
main()
{
    int a, b;
    b = ((a = 3), (a + 2));
    printf("\n b = %d \n",b);
}
```

imprime b = 5.

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie. Par exemple l'instruction composée

```
{
  int a=1;
  printf("\n%d \n%d",++a,a);
}
```

(compilée avec gcc) produira la sortie 2 1 sur un PC Intel/Linux et la sortie 2 2 sur un DEC Alpha/OSF1. (Problème de portabilité)

9. L'opérateur conditionnel ternaire

L'opérateur conditionnel **?** est un opérateur ternaire. Sa syntaxe est la suivante :

```
condition ? expression1: expression2
```

Cette expression est égale à expression1 si condition est satisfaite, et à expression2 sinon. Par exemple, l'expression x >= 0 ? x : -x correspond à la valeur absolue d'un nombre. De même l'instruction m = ((a > b) ? a : b); affecte à m le maximum de a et de b.

10. L'opérateur de conversion de type

L'opérateur de conversion de type, appelé **cast**, permet de modifier explicitement le type d'un objet. On écrit **(type) objet** Par exemple :

```
main()
{
int i = 3, j = 2;
printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

11. L'opérateur adresse

L'opérateur d'adresse **&** appliqué à une variable retourne l'adresse mémoire de cette variable. La syntaxe est **&objet**

Eemple :

```
Main()
{
    int i=5 ;
    printf("l'adresse mémoire de i est :",&i) ;
}
```

imprime l'adresse mémoire de i sous forme hexadécimale : exemple FFF4

12. Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs

opérateurs de même priorité, l'ordre dévaluation est définie par la flèche de la seconde colonne du tableau. On préfèrera toutefois mettre des parenthèses en cas de doute.

Tableau 3.2 : règles de priorité des opérateurs

opérateurs	
() [] -> .	→
! ~ ++ -- -(unaire) (type) *(indirection) &(adresse) sizeof	←
* / %	→
+ -(binaire)	→
<< >>	→
< <= > >=	→
== !=	→
&(et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	←
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions.

Par exemple, il faut écrire if ((x ^ y) != 0)

Chapitre 4 : Les types de données

Objectifs

- *Connaître les principaux types de données du C et leurs caractéristiques*
- *Savoir déclarer de nouveaux types*

Eléments de contenu

- *Les types caractères*
- *Les types entiers*
- *Les types flottants*

1. Les types prédéfinis

Le C est un langage typé. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire. Cette dernière se décompose en une suite continue de mots (>= 1octet). Chaque mot de la mémoire est caractérisé par son adresse, qui est un entier. Deux mots contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type. La taille mémoire correspondant aux différents types dépend des compilateurs. Toutefois, la norme ANSI spécifie un certain nombre de contraintes.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

char int float double
short long unsigned

1.1. Les types caractères

Le mot-clef char désigne un objet de type caractère. Il est codé sur un octet et il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier. Par exemple, si c est de type char, l'expression c + 1 est valide. Elle désigne le caractère suivant dans le code ASCII.

Exemple : le programme suivant affiche la lettre B.

```
#include <stdio.h>
Main()
{
   char c='A' ;
   printf ("%c", c+1) ;
}
```

Suivant les implémentations, le type char est signé ou non. En cas de doute, il vaut mieux préciser unsigned char ou signed char. Notons que tous les caractères imprimables sont positifs.

1.2. Les types entiers

Le mot-clef désignant le type entier est int. Un objet de type int est représenté par un mot de 32 bits pour un DEC alpha ou un PC Intel.

Le type int peut être précédé d'un attribut de précision (short ou long) et/ou d'un attribut de représentation (unsigned). Un objet de type short int a au moins la taille d'un char et au plus la taille d'un int. En général, un short int est codé sur 16 bits. Un objet de type long int a au moins la taille d'un int (64 bits sur un DEC alpha, 32 bits sur un PC Intel).

Tableau 4.1 : les types entiers selon l'architecture

	DEC Alpha	PC Intel (Linux)	
char	8 bits	8 bits	Caractère
short	16 bits	16 bits	Entier court
int	32 bits	32 bits	Entier
long	64 bits	32 bits	Entier long

Le bit de poids fort d'un entier est son signe. Un entier positif est donc représenté en mémoire par la suite de 32 bits dont le bit de poids fort vaut 0 et les 31 autres bits correspondent à la décomposition de l'entier en base 2. Par exemple, pour des objets de type char (8 bits), l'entier positif 12 sera représenté en mémoire par 00001100. Un entier négatif est, lui, représenté par une suite de 32 bits dont le bit de poids fort vaut 1 et les 31 autres bits correspondent à la valeur absolue de l'entier représentée suivant la technique dite du complément à 2. Ainsi, pour des objets de type signed char (8 bits), -1 sera représenté par 11111111, -2 par 11111110, -12 par 11110100. Un int peut donc représenter un entier entre -2^{31} et $(2^{31}-1)$. L'attribut unsigned signifie que l'entier n'a pas de signe. Un unsigned int peut donc représenter un entier entre 0 et $(2^{32}-1)$. Sur un DEC alpha, on utilisera donc un des types suivants :

signed char	$[-2^7..2^7[$
unsigned char	$[0..2^8[$
short int	$[-2^{15}..2^{15}[$
unsigned short int	$[0..2^{16}[$
int	$[-2^{31}..2^{31}[$
unsigned int	$[0..2^{32}[$
long int	$[-2^{63}..2^{63}[$
unsigned long int	$[0..2^{64}[$

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard limits.h.

Le mot-clef sizeof a pour syntaxe sizeof(expression) où expression est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet. Par exemple :

```
unsigned short x;
taille = sizeof(unsigned short);
```

```
taille = sizeof(x);
```

→ Dans les deux cas, taille vaut 4.

Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de limits.h ou le résultat obtenu en appliquant l'opérateur sizeof.

1.3. Les types flottants

Les types float, double et long double servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

Tableau 4.1 : les types flottant selon l'architecture

	DEC Alpha	PC Intel	
float	32 bits	32 bits	flottant
double	64 bits	64 bits	Flottant double précision
Long double	64 bits	128 bits	Flottant quadruple précision

Les flottants sont généralement stockés en mémoire sous la représentation de la virgule flottante normalisée. On écrit le nombre sous la forme « signe, mantisse B exposant ». En général, B = 2. Le digit de poids fort de la mantisse n'est jamais nul. Un flottant est donc représenté par une suite de bits dont le bit de poids fort correspond au signe du nombre. Le champ du milieu correspond à la représentation binaire de l'exposant alors que les bits de poids faible servent à représenter la mantisse.

2. Définition de nouveaux types

Le langage C fournit une fonction appelée **typedef** qui crée de nouveaux types de données.

Exemple :

```
typedef int entier ; /* fait du nouveau nom 'entier' un synonyme de int. Ce type peut
être utilisé dans des déclaration, des 'casts' … exactement comme int. */

tyedef char* STRING ;  /* fait de string un synonyme de char*  */

typedef  struct noeud {
     char* mot ;
     int  quantite ;
     }ARBRE ; /*crée un nouveau type appelé ARBRE (une structure) */
```

On trouve aussi un constructeur de type par énumération :

```
enum <identificateur> { liste de valeurs symboliques }
```

Exemple :

```
enum quadrillatere { carre, rectangle, losange, parallellogramme } q1,q2 ;
```

définit le type par enumération nommé quadrillatere et déclare deux objets q1 et q2 de ce type. (q1=carre ; q2=losange ; …..).

Chapitre 5 : Les fonctions d'E/S standards

Objectifs

- *Se familiariser avec les fonctions assurant l'échange d'information entre la mémoire centrale et les périphériques standard.*

Eléments de contenu

- *Les types caractères*
- *Les types entiers*
- *Les types flottants*

1. Introduction

Les fonctions d'E/S standards sont les fonctions assurant l'échange d'information entre la mémoire centrale et les périphériques standard, principalement le clavier et l'écran.

Dans ce chapitre, nous allons examiner les fonctions permettant de lire les données à partir du clavier et d'afficher les résultats sur l'écran.

2. Les fonctions d'entrées

2.1. La fonction scanf

Cette fonction lit à partir de l'entrée standard (clavier) une liste de variables en mémoire selon un format donné.

```
int  scanf ( const char *format, liste d'adresses);
```

liste d'adresse représente une liste d'adresses de variables déclarées auxquelles les données lues seront attribuées.
format représente le format de lecture des données. C'est une chaîne de caractères (donc entouré par des guillemets). Elle est composée de spécificateurs qui indique le type de variables qu'on va lire. Chaque spécificateur correspond à un type de variable et doit donc apparaître dans le même ordre que la variable dans la liste. Les principaux spécificateurs de format sont résumé dans le tableau suivant.

spécificateur	**signification**
%d ou %x ou %o	Pour une variable de type int
%u	Pour une variable de type unsigned int
%h	Pour une variable de type short int
%f	Pour une variable de type float
%lf	Pour une variable de type double
%e	Pour une variable de type float mise sous forme scientifique
%c	Pour une variable de type char
%s	Pour une variable de type texte

Exemple :

```
Int a = -1;
Unsigned int b = 25;
Char c = 'X';
Scanf("%d%u%c",&a,&b,&c);
```

Remarque:

- On peut placer la longueur de la variable entre le signe % et la lettre spécificateur. Par exemple « %3d » indique qu'on va lire un entier de 3 chiffres.
- Si l'on sépare les spécificateurs de format par des espaces ou par des virgules alors les valeurs à lire seront séparées par les mêmes séparateurs.

2.2. La fonction gets

Elle renvoie une chaîne de caractère lue dans le flux en entrée stdin. Sa déclaration est la suivante :

```
Char * gets (char * s) ;
```

Lorsqu'on lit une chaîne de caractères avec scanf, la lecture s'arrête dès la rencontre d'un blanc. Avec la fonction gets, la lecture se termine à la réception d'un retour chariot '\n'.

Exemple :

```
#include <stdio.h>
void main()
{
      char * CH;
      gets(CH);
}
```

2.3. Les fonction getch(), getche() et getchar()

La première fonction renvoie un caractère lu au clavier sans écho à l'écran alors que la deuxième renvoie un caractère lu au clavier avec écho à l'écran. La fonction getchar renvoie un caractère depuis stdin. Leur déclarations sont les suivantes :

```
int getch(void) ;
int getchar(void);
int getche();
```

Exemple :

```
char  C1,C2,C3;
C1 = getch();
C2 = getchar();
C3 = getche()
```

3. Les fonctions de sorties

3.1. La function printf

Elle permet la traduction de quantité alphanumérique en chaîne de caractères ainsi qu'une présentation formatée de éditions.

```
int  printf ( const char *format, liste d'expression);
```

format : format de représentation.
liste d'exprexssion : variables et expressions dont les valeurs sont à éditer.

La première partie est en fait une chaîne de caractères qui peut contenir

- du texte
- des séquences d'échappement qui permettent de contrôler l'affichage des données. Exemple '\n' qui permet le passage à la ligne suivante
- des spécificateurs de format ou de conversion

Elle contient exactement un spécificateur de format pour chaque expression.

Exemple :

```
int i ;
printf("entrer une valeur :\n");
scanf("%d",&i);
i = i *2
printf("la nouvelle valeur de I est : %d",i) ;
```

3.2. La fonction puts

Envoie une chaîne de caractères vers stdout et ajoute un saut de ligne (newline).

```
int puts (const char * s) ;
```

Exemple :

```
puts (" ceci est un exemple ");
```

3.3. La fonction putchar

Envoie un caractère vers stdout (écran)

```
int putchar (int c) ;
```

Exemple :

```
char c ;
c= 'A'
putchar (c);
putchar ('B');
```

chapitre 6 : Les instructions de branchement conditionnel

Objectifs

- *Se familiariser avec les instructions de branchement conditionnel et comprendre leur utilité.*

Eléments de contenu

- *L'instruction if--else*
- *L'instruction switch*

1. If – else

algorithmique

```
si ( expression logique)
alors
    bloc d'instructions 1
sinon
    bloc d'instructions 2
fsi
```

- Si l'**expression** logique a la valeur logique *vrai*, alors le **bloc d'instructions 1** est exécuté
- Si l'**expression** logique a la valeur logique *faux*, alors **le bloc d'instructions 2** est exécuté

Langage C

```
if ( expression )
        bloc d'instructions 1
else
        bloc d'instructions 2
```

- Si l'**expression** fournit une valeur différente de zéro, alors le **bloc d'instructions 1** est exécuté
- Si l' **expression** fournit la valeur zéro, alors le **bloc d'instructions 2** est exécuté

La partie **expression** peut désigner :

- une variable d'un type numérique,
- une expression fournissant un résultat numérique.

La partie **bloc d'instructions** peut désigner :

- un (vrai) bloc d'instructions compris entre accolades.
- une seule instruction terminée par un point-virgule.

On peut avoir des « **if »** imbriqués

Exemples :

```
if (a > b)
   max = a;
else
   max = b;
```

```
if (A-B)
        printf("A est différent de B\n");
else
        printf("A est égal à B\n");
```

```
if (temperature < 0)
{
        printf("glace");
}
else
{
     if (temperature < 100)
     {
         printf("eau");
      }
     else
     {
         printf("vapeur");
     }
}
```

2. switch

C'est un moyen qui permet de décider dans une structure à plusieurs cas. Cette instruction teste si une expression prend une valeur parmi un ensemble de constante et fait le branchement en conséquence.

algorithmique

```
selon cas  faire
    cas1 : liste d'instructions 1
    cas1 : liste d'instructions 1
    …
    defaut : liste instructions N
finselon
```

Langage C

```
switch   (expression)
{
  case expr_cst1 : liste d'instructions1
  case expr_cst1 : liste d'instructions2
  …
  default : liste instructions N
}
```

- Le choix de l'ensemble d'instructions à exécuter est calculé par l'évaluation de **expression** qui doit envoyer un type entier.
- expr_cste doit être un **int** et doit être unique

Exemple :

```
int mois ;
scanf(" %d" ,&mois) ;
switch   ( mois )
 {
       case 1 : printf(" janvier" ) ; break ;
       case 2 : printf(" fevrier" ) ; break ;
       …
       case 12 : printf(" décembre" ) ; break ;
       default : printf(“erreur”)
 }
```

L’instruction break permet de sortir de l’instruction switch. Elle est importante car si on ne la met pas après chaque cas d’exécution alors toutes les instructions après ce cas seront exécutée (bien sur s’ils ne sont pas suivie d’une autre instruction break).

Remarque :

Il existe des instructions de branchement non conditionnels : goto, continue et break.

chapitre 7 : Les structures répétitives

Objectifs

- *Se familiariser avec les structures répétitives*
- *Comprendre l'importance de ces structures dans l'écriture de programmes.*

Eléments de contenu

- *La structure while*
- *La structure do--while*
- *La structure for*

1. Introduction :

En C, nous disposons de trois structures qui nous permettent la définition de boucles conditionnelles:

- la structure : **while**
- la structure : **do - while**
- la structure : **for**

Théoriquement, ces structures sont interchangeables, c.-à-d qu'il serait possible de programmer toutes sortes de boucles conditionnelles en n'utilisant qu'une seule des trois structures. Mais il est recommandé de choisir toujours la structure la mieux adaptée au cas du traitement à faire.

2. Les structures répétitives

2.1. While

algorithmique

```
tant que (expression logique) faire
        bloc d'instructions
ftantque
```

Tant que l'**expression logique** fournit la valeur *vrai*, le **bloc d'instructions** est exécuté.

Si l'**expression logique** fournit la valeur *faux*,

Langage C

```
while (expression)
        bloc d'instructions
```

Tant que l'expression fournit une valeur différente de zéro, le bloc d'instructions est exécuté.

l'exécution continue avec l'instruction qui suit *ftanquet*.

Le **bloc d'instructions** est exécuté zéro ou plusieurs fois.

expression peut désigner :

- une variable d'un type numérique.
- une expression fournissant un résultat numérique.

La partie **bloc d'instructions** peut désigner :

- un bloc d'instructions compris entre accolades.
- Une seule instruction terminée par un point-virgule.
- Rien (boucle infinie)

Exemple 1:

```
/* Afficher les nombres de 0 à 9 */
int I = 0;
while (I<10)
{
      printf("%d \n", I);
      I++;
}
```

```
/* Afficher les nombres de 0 à 9 */
int I = 0 ;
while (I<10)
printf("%d\n",I++);
```

```
/* Afficher les nombres de 1 à 10 */
int I = 0 ;
while (I<10)
printf("%d\n",++I);
```

Exemple2:

```
/* faire la somme des N premiers terme  entier*/
int somme=0, i = 0;
while (i<N)
{
     somme += i;
     i++ ;
}
```

Exemple3:

```
/* Afficher des caractères */
unsigned char c=0;
while (c<255)
     printf("%c \n", c++);
```

2.2. do-While

algorithmique

```
Répeter
    bloc d'instruction
jusqu'à (expression)
```

Langage C

```
do
      bloc d'instructions
while ( expression );
```

Le bloc d'instructions est exécuté au moins une fois et aussi longtemps que l'expression fournit une valeur différente de zéro.

do - while est comparable à la structure répéter du langage algorithmique si la condition finale est inversée logiquement.

La structure **do - while** est semblable à la structure **while**, avec la différence suivante :

- while évalue la condition avant d'exécuter le bloc d'instructions.
- do - while évalue la condition après avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois.

En pratique, la structure do - while n'est pas si fréquente que while; mais dans certains cas, elle fournit une solution plus élégante. Une application typique de do - while est la saisie de données qui doivent remplir une certaine condition.

Exemple1:

```
float N;
do
  {
      printf("Introduisez un nombre entre 1 et 10 :");
       scanf("%f", &N);
  }
while (N<1 || N>10);
```

Exemple2 :

```
int n, div;
printf("Entrez le nombre à diviser : ");
scanf("%i", &n);
do
{
   printf("Entrez le diviseur ( != 0) : ");
   scanf("%i", &div);
}
while (!div);
printf("%di / %d = %f\n", n, div, (float)n/div);
```

Exemple3 :

```
float N;
do
{
   printf("Entrer un nombre (>= 0) : ");
   scanf("%f", &N)
}
while (N < 0);
   printf("La racine carrée de %.2f est %.2f\n", N, sqrt(N));
```

2.3. for

La structure pour en langage algorithmique est utilisées pour faciliter la programmation de boucles de comptage. La structure for en C est plus générale et beaucoup plus puissante.

```
for ( expr1 ; expr2 ; expr3 )
       bloc d'instructions
```

Est équivalente à :

```
expr1;
while (expr2 )
{
      bloc d'instructions
      expr3 ;
}
```

expr1 est évaluée une fois avant le passage de la boucle. Elle est utilisée pour initialiser les données de la boucle.

expr2 est évaluée avant chaque passage de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.

expr3 est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

En pratique, les parties ***expr1*** et ***expr2*** contiennent souvent plusieurs initialisations ou réinitialisations, séparées par des virgules.

Exemple1:

```
/* affichage des carrés des nombres entire compris entre 0 et 20 */
int I;
for (I=0 ; I<=20 ; I++)
    printf("Le carré de %d est %d \n", I, I*I);
```

Exemple2 :

```
int n, tot;
for (tot=0, n=1 ; n<101 ; n++)
     tot+=n;
printf("La somme des nombres de 1 à 100 est %d\n", tot);
```

Exemple3 : affichage du code binaire d'un caractère lue au clavier

Notation utilisant la structure while

```
int C, I;
C=getchar();
I=128;
while (I>=1)
{
      printf("%d ", C/I);
      C%=I;
      I/=2;
}
```

Notation utilisant for - très lisible –

```
int C, I;
C=getchar();
for (I=128 ; I>=1 ; I/=2)
{
      printf("%i ", C/I);
      C%=I;
}
```

Notation utilisant for - plus compacte

```
int C, I;
C=getchar();
for (I=128 ; I>=1 ; C%=I, I/=2)
      printf("%i ", C/I);
```

Notation utilisant for - à déconseiller -

```
int C, I;
for(C=getchar(),I=128; I>=1 ;printf("%i ",C/I),C%=i,i/=2);
```

Chapitre 8 : les tableaux

Objectifs

- *Assimiler le concept des tableaux*
- *Connaître les différents traitements qu'on peut exécuter sur les tableaux*
- *Comprendre l'importance des tableaux dans la structuration des programmes.*

Eléments de contenu

- *Les tableaux à une dimension*
  - *Déclaration et mémorisation*
  - *Initialisation et réservation automatique*
  - *Affichage des éléments d'un tableau*
  - *affectation de valeurs aux éléments du tableau*
- *Les tableaux à deux dimensions*
  - *Déclaration et mémorisation*
  - *Initialisation et réservation automatique*
  - *Affichage des éléments d'un tableau*
  - *affectation de valeurs aux éléments du tableau*

1. Les tableaux à une dimension

1.1. Déclaration et mémorisation

- **Déclaration**

Déclaration de tableaux en langage algorithmique :

```
TypeSimple tableau NomTableau [Dimension]
```

Déclaration de tableaux en C :

```
TypeSimple NomTableau[Dimension];
```

Les noms des tableaux sont des identificateurs.

Exemples :

```
int notes [8] ;  /* déclaration d'un tableau nommé notes, de type int et de  dimension 8 */
char tab [100] ; /* déclaration d'un tableaux nommé tab, de type char et de dimension 100 */
float moy[40] ; /* déclaration d'un tableau nommé moy, de type float et de dimention 100 */
```

- **Mémorisation**

En C, le nom d'un tableau est le représentant de ***l'adresse du premier élément*** du tableau. Les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse.

Exemple:

```
char A[5] = {'A', 66, 70, 'C', 240};
```

……	65	66	70	67	240	……..

1E02

A

Si un tableau est formé de N composantes et si une composante a besoin de M octets en mémoire, alors le tableau occupera de N*M octets.

1.2. 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 :

```
int A[5]   = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int C[10]  = {1, 0, 0, 1, 1};
```

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.

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

Exemples

```
int A[] = {10, 20, 30, 40, 50};
```

==> réservation de 5*sizeof(int) octets (dans notre cas: 10 octets)

```
float B[] = {-1.05, 3.33, 87e-5, -12.3E4};
```

==> réservation de **4*sizeof(float)** octets (dans notre cas: 16 octets)

```
int C[] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

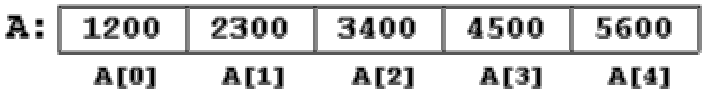
==> réservation de **10*sizeof(int)** octets (dans notre cas: 20 octets)

1.3. Accès aux composantes

En déclarant un tableau par **int A[5];** nous avons défini un tableau A avec cinq composantes, auxquelles on peut accéder par: A[0], A[1], ... , A[4]

Exemple :

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```



1.4. Affichage et affectation

La structure for se prête particulièrement bien au travail avec les tableaux. La plupart des applications se laissent implémenter par simple modification des exemples-types de l'affichage et de l'affectation.

- **Affichage du contenu d'un tableau : exemple**

```
main()
{
 int A[5];
 int I;  /* Compteur */
 for (I=0; I<5; I++)
    printf("%d ", A[I]);
 return 0;
 printf("\n");
}
```

- **Affectation avec des valeurs provenant de l'extérieur :exemple**

```
main()
{
 int A[5];
 int I;  /* Compteur */
 for (I=0; I<5; I++)
    scanf("%d", &A[I]);
 return 0;
```

2. Les tableaux à deux dimension

2.1. Déclaration et mémorisation

- **déclaration**

Déclaration de tableaux à deux dimensions en lang. algorithmique

```
TypeSimple tableau NomTabl[DimLigne,DimCol]
```

Déclaration de tableaux à deux dimensions en C

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

Exemples :

```
double A[2][5] ; /* déclaration d'un tableau à deux dimension nommé A et de type double */
char B[4][2] ; /* déclaration d'un tableau à deux dimension nommé B et de type char */
```

- **Mémorisation**

Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de ***l'adresse du premier élément*** du tableau (c.-à-d. l'adresse de la première ***ligne*** du tableau). Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

Exemple*:* Mémorisation d'un tableau à deux dimensions

```
short A[3][2] = {{1,   2  },
                 {10,  20 },
                 {100, 200}};
```

…….	1	2	10	20	100	200	……..

A → 1E08

2.2. 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. A 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.

Exemples

```
int A[3][10] ={{ 0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               { 1,12,23,34,45,56,67,78,89,90}};

float B[3][2] = {{-1.05,   -1.10  },
                 {86e-5,    87e-5 },
                 {-12.5E4, -12.3E4}};
```

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.

Si le nombre de **lignes L** n'est pas indiqué explicitement lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

```
int A[][10] = {{ 0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               { 1,12,23,34,45,56,67,78,89,90}};
```

2.3. Accès aux composantes

L'accès à un élément d'un tableau à deux dimensions se fait selon le schéma suivant :

```
NomTableau[Ligne][Colonne]
```

Les éléments d'un tableau de dimensions L et C se présentent de la façon suivante:

```
/                                                          \
|  A[0][0]    A[0][1]    A[0][2]    . . .    A[0][C-1]   |
|  A[1][0]    A[1][1]    A[1][2]    . . .    A[1][C-1]   |
|  A[2][0]    A[2][1]    A[2][2]    . . .    A[2][C-1]   |
|   . . .      . . .      . . .     . . .      . . .     |
| A[L-1][0]  A[L-1][1]  A[L-1][2]   . . .   A[L-1][C-1] |
\                                                          /
```

2.4. Affichage et affectation

Lors du travail avec les tableaux à deux dimensions, nous utiliserons deux indices (p.ex: I et J), et la structure **for**, souvent imbriquée, pour parcourir les lignes et les colonnes des tableaux.

- **Affichage du contenu d'un tableau à deux dimensions**

```
main()
{
 int A[5][10];
 int I,J;
 /* Pour chaque ligne ... */
 for (I=0; I<5; I++)
   {
    /* ... considérer chaque composante */
    for (J=0; J<10; J++)
          printf("%7d", A[I][J]);
    /* Retour à la ligne */
    printf("\n");
   }
 return 0;
}
```

- **Affectation avec des valeurs provenant de l'extérieur**

```
main()
{
 int A[5][10];
 int I,J;
 /* Pour chaque ligne ... */
 for (I=0; I<5; I++)
   /* ... considérer chaque composante */
   for (J=0; J<10; J++)
          scanf("%d", &A[I][J]);
return 0;
}
```

Chapitre 9 : Les chaînes de caractères

Objectifs

- *Connaître la convention de représentation des chaînes de caractères en C*
- *Comprendre les entrées -sorties de chaînes*
- *Manipuler les fonctions de traitement et de conversion de chaînes*

Eléments du contenu

- *La convention de représentation de chaînes de caractères en C*
- Les *entrées sorties de chaînes*
- *Les fonctions de concaténation de chaînes*
- *Les fonctions de comparaison de chaînes*
- *Les fonctions de copie de chaînes*
- *Les fonctions de recherche dans une chaîne*
- *Les fonctions de conversion*

Il n'existe pas de type spécial *chaîne* ou *string* en C. Une chaîne de caractères est traitée comme un *tableau à une dimension de caractères.*

1. Déclaration :

La déclaration d'une chaîne de caractère se fait par l'une des méthodes suivantes :

- ✓ **char NomChaine [Longueur];**
- ✓ **char * NomChaine ;**

Exemple :

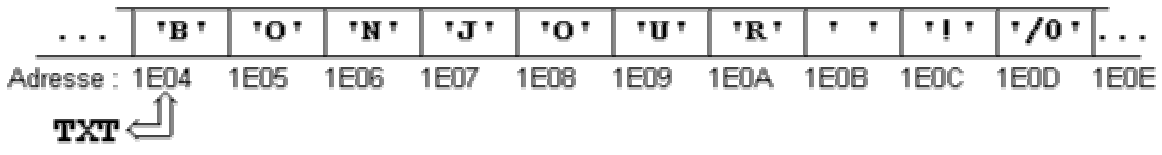
```
char NOM [20];                        /* déclaration sans initialisation  */
char PRENOM [] = " stream";       /* déclaration avec initialisation */
char adresse1 [] = { 'c', 'h','a', 'r' ,'g','u','i','a',\0'} ;  /* déclaration avec initialisation */
char adresse2 [20] = { 'c', 'h','a', 'r' ,'g','u','i','a','\0'} ;  /* déclaration avec initialisation */
char * CH1 ;                          /* déclaration sans initialisation */
char * CH2= " voila" ;              /* déclaration avec initialisation */
CH1 = " affectation permise"  /* affectation permise mais NOM = " chaine" est non permise  */
char PHRASE [300];                /* déclaration sans initialisation  */
```

2. Mémorisation :

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 :

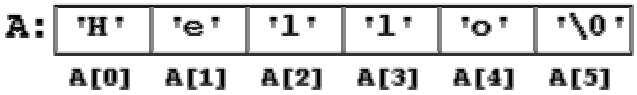
```
char TXT [10] = "BONJOUR !";
```



3. Accès aux éléments :

Exemple :

```
char TXT [6] = "hello";
```



4. Utilisation des chaînes de caractères :

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

Exemples 1: les fonctions de stdio.h

```
char TXT [6] = "hello";
char CH[5] ;
printf ("%s",TXT) ;       /* affiche hello sans retour à la ligne*/
printf ("%.3s", TXT) ;    /* affiche hel   */
scanf ("%s", CH) ;        /* CH contient l'adresse du premier caractère de la chaine */
puts ( TXT) ;             /* affiche hello avec un retour à la ligne */
puts (" bonjour") ;       /* affiche bonjour avec un retour à la ligne */
gets(CH) ;     /* lit une ligne de de caractères de stdin et la copie à l'adresse indiquée par CH */
```

Exemples 2: les fonctions de string.h

```
char CH1 [] = "hello";
char CH2 [] = "bonjour";
char * chaine = "bonsoir" ;
char * sous_chaine = "soir" ;
int k = 2 ;
char C='a' ;
strlen (CH1) ;                /* fournit la longueur de CH1 */
strcpy (CH1, CH2) ;           /* copie CH1 dans CH2 */
strncpy (CH1, CH2,k) ;        /* copie au plus k caractère de CH1 vers CH2 */
strcat (CH1, CH2) ;           /* ajoute CH2 à la fin de CH1 */
strcmp ( CH1, CH2 ) ;         /* compare CH1 et CH2 lexico graphiquement */
stricmp ( CH1, CH2 ) ;        /* compare CH1 et CH2 lexico graphiquement  sans tenir min maj*/
strnicmp( CH1, CH2 ) ;        /* compare CH1 et CH2 sans tenir compte de la différence entre  */
strncat (CH1, CH2,k) ;        /* miniscule et majiscule */
strrev (chaine) ;             /*inverse la chaîne et, renvoie l'adresse de la chaine inversée*/
strchr(chaine,C) ;            /*recherche la première occurrence du caractère dans la chaîne*/
strrchr (chaine,C) ;          /*idem en commençant par la fin*/
strstr (chaine,sous_chaine) ;/*recherche la première occurrence de sous_chaine ds la chaine*/
strpbrk (CH1,CH2) ;/*recherche, dans CH1 la première occurrence d'un caractère  de CH2 */
```

strcmp, stricmp et strcmpi renvoient un nombre:

- positif si la chaîne1 est supérieure à la chaine2 (au sens de l'ordre lexicographique)
- négatif si la chaîne1 est inférieure à la chaîne2
- nul si les chaînes sont identiques.

Les 4 dernières fonctions renvoient l'adresse de l'information recherchée en cas de succès, sinon le pointeur NULL (c'est à dire le pointeur de valeur 0 ou encore le pointeur faux).

Remarque:

- ✓ Comme le nom d'une chaîne de caractères représente une adresse fixe en mémoire, on ne peut pas 'affecter' une autre chaîne au nom d'un tableau. Il faut bien copier la chaîne caractère par caractère ou utiliser la fonction **strcpy** respectivement **strncpy**:

```
strcpy(CH1, "bonsoir");
```

Exemples 3: les fonctions de stlib.h

La bibliothèque <*stdlib*> contient des déclarations de fonctions pour la conversion de nombres en chaînes de caractères et vice-versa.

```
char CH [] = "125";
atoi ( CH);      /* retourne la valeur numérique représenté par CH comme int */
atol (CH) ;      /* retourne la valeur numérique représenté par CH comme long */
atof (CH) ;     /* retourne la valeur numérique représenté par CH comme double */
```

Règles générales pour la conversion:

- ✓ Les espaces au début d'une chaîne sont ignorés
- ✓ Il n'y a pas de contrôle du domaine de la cible
- ✓ La conversion s'arrête au premier caractère non convertible
- ✓ Pour une chaîne non convertible, les fonctions retournent zéro

Remarque:

- ✓ Le standard ANSI-C ne contient pas de fonctions pour convertir des nombres en chaînes de caractères. Si on se limite aux systèmes fonctionnant sous DOS, on peut quand même utiliser les fonctions **itoa**, **ltoa** et **ultoa** qui convertissent des entiers en chaînes de caractères.

```
char CH [20] ;
int base = 10; /* base est compris entre 2 et 36 */
int x=5 ;
long y =20;
unsigned long z = 300 ;
itoa ( x, CH, base);
ltoa ( y, CH, base);
ultoa(z, CH, base);
```

Exemples 3: les fonctions de ctype.h

Les fonctions de <*ctype*> servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés. Les fonctions de <*ctype*> sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes. Dans la suite, c représente une valeur du type **int** qui peut être représentée comme caractère.

```
isupper(c) ;  /* retourne une valeur différente de zéro si c est une majiscule */
islower(c) ;  /* retourne une valeur différente de zéro si c est une miniscule */
isdigit  (c) ;  /* retourne une valeur différente de zéro si c est un chiffre décimal */
isalpha(c) ;  /* retourne une valeur différente de zéro si  isupper(c)  ou islower(c) */
isalnum(c) ;  /* retourne une valeur différente de zéro si isalpha(c) ou isdigit  (c) */
isxdigit(c) ;   /* retourne une valeur différente de zéro si c est un chiffre hexadécimal */
isspace(c) ;  /* retourne une valeur différente de zéro si c est un signe d'espacement */
```

Les fonctions de ***conversion*** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de **c** reste inchangée:

```
tolower(c) ;  /* retourne  c convrti en miniscule si c est une majiscule */
toupper(c) ;  /* retourne  c convrti en majiscule si c est une miniscule */
```

5. Tableaux de chaîne de caractères :

Souvent, il est nécessaire de mémoriser une suite de mots ou de phrases dans des variables. Il est alors pratique de créer un tableau de chaînes de caractères, ce qui allégera les déclarations des variables et simplifiera l'accès aux différents mots (ou phrases).

- ✓ Declaration : elle se fait par l'une des deux méthodes

```
char JOUR [7] [9];
char * mois [12] ;
```

- ✓ Initialisation : lors de la déclaration.

```
char JOUR [7] [9]= {"lundi", "mardi", "mercredi","jeudi", "vendredi", "samedi","dimanche"};
char * MOIS [12] = {"jan", "fev", "mars","avr", "may", "juin","juil","aout", "sep", "oct","nov", "dec"};
```

- ✓ Accès aux différents composantes :

```
Printf ("%s", JOUR [2]);  /* affiche mercredi */
Printf ("%s", MOIS [0]);  /* affiche jan */
```

- ✓ Accès aux caractères : (similaire au cas des tableaux)

```
for(I=0; I<7; I++)
      printf("%c  ", JOUR[I][0]);
```

- ✓ Affectation : pas d'affectation directe (genre JOUR [2] = "vendredi" ;). Utiliser strcpy

```
strcpy(JOUR[4], "Friday");
```

Chapitre 10 : Les types de variables complexes (Structures, union et énumération)

Objectifs

- *Connaître la syntaxe de la déclaration des structures* en langage *C*
- *Comprendre l'utilité de l'utilisation d'une structure*
- *Connaître la déclaration, des* types *synonymes en C*
- *Différencier entre les structures, les unions et les énumérations*

Eléments du contenu

- *Exemples de déclaration de structures*
- *Utilisation d'une structure*
- *La déclaration de types synonymes*
- *Les champs de bits*
- *Les énumérations*
- *Les unions*

1. Notion de structure

C'est un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité en utilisant le concept d'enregistrement. Un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des *champs*. Le langage C possède le concept d'enregistrement appelé structure.

1.1. Déclaration de structure :

Méthode 1 : déclaration en précisant un nom pour la structure

```
struct personne
  {
  char nom[20];
  char prenom[20];
  int  n_cin;
  };
```

Déclare l'identificateur **`personne`** comme étant le nom d'un type de structure, composée de trois membres. On peut ensuite utiliser ce type structure pour déclarer des variables, de la manière suivante :

```
struct personne p1,p2;   /* qui déclare deux variables de type struct
personne */
                                                    /*  de noms
p1 et p2 */
```

Méthode 2 : déclaration sans préciser un nom pour la structure

```
struct
  {
  char nom[20];
  char prenom[20];
  int  n_cin;
  } p1,p2;
```

Déclare deux variables de noms p1 et p2 comme étant deux structures de trois membres, mais elle ne donne pas de nom au type de la structure. L'inconvénient de cette méthode est qu'il sera par la suite impossible de déclarer une autre variable du même type. De plus, si plus loin on écrit :

```
struct
  {
  char nom[20];
  char prenom[20];
  int  n_cin;
  } p3;
```

Les deux structures, bien qu'elles ont le même nombre de champs, avec les mêmes noms et les mêmes types, elles seront considérées de types différents. Il sera impossible en particulier d'écrire `p3 = p1;`

Méthode 3 : déclaration en précisant un nom pour la structure et en déclarant des variables (combiner)

```
struct  personne
  {
        char nom[20];
        char prenom[20];
        int  n_cin;
  } p1,p2;
```

Déclare les deux variables **`p1`** et **`p2`** et donne le nom **`personne`** à la structure. Là aussi, on pourra utiliser ultérieurement le nom **`struct personne`** pour déclarer d'autres variables :

```
struct personne pers1, pers2, pers3;   /*sont du même type que p1 et p2 */
```

→ De ces trois méthodes c'est la première qui est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.

Méthode 4 : déclaration en utilisant **typedef**

```
typedef struct                  /* On définit un type struct */
{
      char nom[10];
      char prenom[10];
      int age;
      float moyenne;
}fiche;
/* ou bien */
typedef struct  fiche          /* On définit un type struct */
```

```
{
      char nom[10];
      char prenom[10];
      int age;
      float moyenne;
};
```

Utilisation : On déclare des variables par exemple :

```
fiche f1,f2;
```

1.2. Accès aux membres d'une structure :

Pour désigner un membre d'une structure, il faut utiliser l'opérateur de sélection de membre qui se note **.** (point).

Exemple 1 :

```
struct personne
  {
  char nom[20];
  char prenom[20];
  int  n_cin;
  };
struct personne p1,p2;
p1.n_cin=15 ; /* accès au troisième champs + affectation */
p1.nom ;        /* accès au premier champs */
p1.prenom ;    /* accès au deuxième champs */
```

Les membres ainsi désignés se comportent comme n'importe quelle variable et par exemple, pour accéder au premier caractère du nom de **p2**, on écrira : **`p2.nom[0]`**.

1.3. Initialisation d'une structure :

Exemple1 :

```
struct personne pr1 = {"jean", "Dupond", 500};
```

Exemple2 :

```
struct personne pr2;        /* déclaration de pr2 de type struct personne */
strcpy ( pr2.nom, "jean" ) ;   /* affectation de  "jean" au deuxième champs */
strcpy ( pr2.prenom, "Dupont" ) ;
pr2.n_cin = 164 ;
```

Remarque :

On ne peut pas faire l'initialisation d'une structure lors de sa déclaration.

1.4. Affectation de structures :

On peut affecter une structure à une variable structure de même type.

```
pr1 = pr2 ;
```

1.5. Comparaison de structures :

Aucune comparaison n'est possible sur les structures, même pas les opérateurs **==** et **!=**.

1.6. Tableau de structures :

Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple. Supposons que l'on ait déjà déclaré la **`struct personne.`**

```
struct personne t[100]; /* dec d'un tableau de 100 structures de type struct personne */
```

Pour référencer le nom de la personne qui a l'index **`i`** dans **`t`** on écrira : **`t[i].nom`**.

1.7. Composition de structures :

Une structure permet de définir un type. Ce type peut être utilisé dans la déclaration d'une autre structure comme type d'un de ses champs.

Exemple :

```
struct date
  {
      unsigned int jour;
       unsigned int mois;
       unsigned int annee ;
  };
struct personne
  {
  char nom[20];
  char prenom[20];
  struct date  d_naissance;
  };
```

2. Les champs de bits

Il est parfois nécessaire pour un programmeur de décrire en termes de bits la structure d'une ressource matérielle de la machine. Un exemple typique est la programmation système qui nécessite de manipuler des registres particuliers de la machine. Il existe dans le langage C un moyen de réaliser de telles descriptions, à l'aide du concept de structure. En effet, dans une déclaration de structure, il est possible de faire suivre la définition d'un membre par une indication du nombre de bits que doit avoir ce membre. Dans ce cas, le langage C appelle ça un *champ de bits*. Par exemple, un registre d'état décrit comme suit :

```
struct registre
  {
        unsigned int : 3;          /*   inutilisé   */
```

```
        unsigned int extend : 1;
        unsigned int overflow : 1;
        unsigned int carry : 1;
};
```

On voit que le langage C accepte que l'on ne donne pas de nom aux champs de bits qui ne sont pas utilisés.

- ✓ Les seuls types acceptés pour les champs de bits sont **`int`**, **`unsigned int`** et **`signed int`**.
- ✓ L'ordre dans lequel sont mis les champs de bits à l'intérieur d'un mot dépend de l'implémentation, mais généralement, dans une machine *little endian* les premiers champs décrivent les bits de poids faibles et les derniers champs les bits de poids forts, alors que c'est généralement l'inverse dans une machine *big endian*.
- ✓ Un champ de bits n'a pas d'adresse, on ne peut donc pas lui appliquer l'opérateur adresse &.

3. Les énumérations :

On sait qu'on pouvait déclarer des constantes nommées de la manière suivante :

```
enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

Qui déclare les identificateurs **`LUNDI`**, **`MARDI`**, etc. comme étant des constantes entières de valeur 0, 1, etc. Les énumérations fonctionnent syntaxiquement comme les structures : après le mot-clé **`enum`** il peut y avoir un identificateur appelé *étiquette d'énumération* qui permettra plus loin dans le programme de déclarer des variables de type énumération.

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

Exemple1 :

```
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
enum jour j1, j2;
j1 = LUNDI;
j2 = MARDI;
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE} d1, d2;
enum {FAUX, VRAI} b1,b2;  /* mauvaise programmation */
```

4. Les unions :

Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type struct. Dans l'exemple suivant, la variable hier de type union jour peut être soit un entier soit un caractère

```
union jour
  {
  int numero;
  char lettre;
  } ;
union jour hier, demain ; /* déclaration de deux variables de type union jour */
hier.numero = 5 ;
hier.lettre='D' ; /* écrase la valeur précédente */
```

Remarque :

La différence sémantique entre les struct et les unions est la suivante : alors que pour une variable de type structure tous les membres peuvent avoir en même temps une valeur, une variable de type union ne peut avoir à un instant donné qu'un seul membre ayant une valeur.

4.1. Utilisation des unions

Lorsqu'il manipule des variables de type union, le programmeur n'a malheureusement aucun moyen de savoir à un instant donné, quel est le membre de l'union qui possède une valeur. Pour être utilisable, une union doit donc toujours être associée à une variable dont le but sera d'indiquer le membre de l'union qui est valide. En pratique, une union et son indicateur sont généralement englobés à l'intérieur d'une structure. Dans l'exemple, on procédera de la manière suivante :

```
enum type {ENTIER, FLOTTANT};

struct arith
  {
  enum type typ_val;   /*   indique ce qui est dans u   */
  union
    {
        int i;
        float f;
    } u;
  };
```

La **struct arith** a deux membres typ_val de type int, et u de type union d'int et de float. On déclarera des variables par :

```
struct arith a1,a2;
```

Puis on pourra les utiliser de la manière suivante :

```
a1.typ_val = ENTIER;
a1.u.i = 10;
a2.typ_val = FLOTTANT;
a2.u.f = 3.14159;
```

Si on passe en paramètre à une procédure un pointeur vers une struct arith, la procédure testera la valeur du membre typ_val pour savoir si l'union reçue possède un entier ou un flottant.

5. La déclaration de types synonymes : typedef

La déclaration *typedef* permet de définir "types synonymes". A priori, elle s'applique à tous les types et pas seulement aux structures. C'est pourquoi nous commencerons par l'introduire sur quelques exemples avant de montrer l'usage que Ion peut on faire avec les structures.

Exemple1 :

```
Typedef int entier ; /* entier est synonyme de int, entier i,j ; est correcte  */
```

Exemple2 : application aux structures

```
Struct enreg
{     int numéro ;
      int qte ;
      float  prix ;
 } ;
typedef struct enreg s_enreg;
s_enreg art1,art2;
/*  ou encore, plus simplement */
typedef struct
{
    int numéro ;
    int qte;
} s_enreg;
s_enreg art1,art2;
```

L'emploi de typdef permet ainsi de déclarer des variables du type correspondant à noter modèle de structure enreg, à l'aide d'un seul mot de notre choix (ici s_enreg). La deuxième forme montre comment le nom de modèle peut être omis, sans compromettre d'éventuelles déclarations ultérieures de variable de type que l'on a ainsi définit.

Chapitre 11 : Les pointeurs

Objectifs

- *Connaître la notion de pointeur en C*
- *Connaître les caractéristiques des variables de type pointeur en C*
- *Utiliser les pointeurs pour la résolution de divers problèmes*

Eléments du contenu

- *Adresse et valeur d'un objet*
- *Notion de pointeur*
- *Arithmétique des pointeurs*
- *Allocation dynamique*
- *Pointeurs et tableaux*
- *Pointeurs et chaîne de caractères*
- *Pointeurs et structures*
- *Les listes chaînées*

1. Introduction

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée de mots qui sont identifiés de manière univoque par un numéro qu'on appelle adresse. Pour retrouver une variable, il suffit donc de connaître l'adresse du mot où elle est stockée ou (s'il s'agit d'une variable qui recouvre plusieurs mots contigus) l'adresse du premier de ces mots. Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs et non par leur adresses. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

2. Adresse et valeur d'un objet :

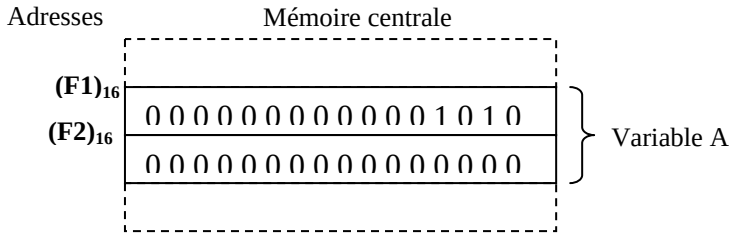
On appelle Lvalue (Left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :

- son adresse. C'est à dire l'adresse mémoire à partir de laquelle l'objet est stocké.
- sa valeur, c'est à dire ce qui est stocké à cette adresse.

✓ **Adressage direct :** Le nom de la variable nous permet d'accéder *directement* à sa valeur.
→ ***Adressage direct:*** Accès au contenu d'une variable par le nom de la variable.

Exemple : (mot mémoire de taille 2 octets. type **int** codé sur 4 octets)

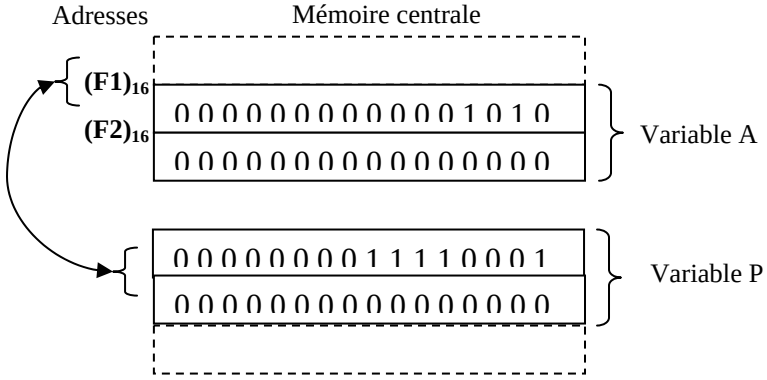
```
int A=10 ;
```



- ✓ **Adressage indirect :** Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur. Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

→ ***Adressage indirect:*** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple :
Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



3. Notion de pointeur :

Un pointeur est un objet (variable) 'Lvalue' dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

Type * nom_du_pointeur ;

Type de l'objet pointé

Identificateur dont la valeur est l'adresse d'un objet de type **Type**

Exemple :

```
char * pc;      /*pc est un pointeur pointant sur un objet de type char*/
int *pi;        /*pi est un pointeur pointant sur un objet de type int*/
float *pf;      /*pf est un pointeur pointant sur un objet de type float*/
```

L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé. Ainsi si p est un pointeur vers un entier i, *p désigne la valeur de i. Par exemple, le programme :

```
Main()
{
    int  i =3;
    int * p ;          /* p est un pointeur sur un objet de type entier */
    p=&I;              /* p contient l'adresse de la variable i */
    printf ("contenu de la case mémoire pointé par p est :  %d ", *p) ;
}
```

Imprime : **contenu de la case mémoire pointé par p est : 3**

Dans ce programme, les objets **i** et ***p** sont identiques. Ils ont même adresse et valeur. Cela signifie en particulier que toute modification de *p modifie i.

Remarque :

- La valeur d'un pointeur est toujours un entier.
- Le type d'un pointeur dépend du type de la variable à laquelle il pointe

4. Arithmétique des pointeurs :

La valeur d'un pointeur étant un entier. On peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur le départ.
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ.
- la différence de deux pointeurs p et q pointant tous deux vers des objets de même type. Le résultat est un entier dont la valeur est égale à **(p-q)/sizeof(type)**.

Notons que la somme de deux pointeurs n'est pas autorisée.

Si **k** est un entier et **p** est un pointeur sur un objet de type ***type.*** L'expression **p + k** désigne un pointeur sur un objet de type ***type*** dont la valeur est égale à la valeur de p incrémentée de **k*sizeof(type)**. Il en va de même pour la soustraction d'un entier à un pointeur et pour les opérateurs d'incrémentation et de décrémentation ++ et --.

Exemple : (on suppose sizeof(int)=2, mots mémoire=2octets et i est rangé à l'adresse F1).

```
Void main()
{
    int i = 3 ;
    int   *p1,  *p2 ;      /* p1 et  p2 sont deux pointeur vers deux objets de type int */
    p1 = &I ;              /* p1 contient l'adresse de l'entier  I */
    p2 = p1 + 2;           /* p2 contient la valeur rangé dans p1 incrémenté de  2* sizeof (int) */
    printf ( " p1 = %x  et p2 = %x ", p1,p2) ;    /* imprime p1= F1  et  p2 = F5 */
}
```

Exemples : (mots mémoire=1octets et sizeof(int)=4).

```
int *pi;                /*  pi pointe sur un objet de type entier */
float *pr;              /*  pr pointe sur un objet de type réel */
char *pc;               /*  pc pointe sur un objet de type caractère */

*pi = 4;                /*  4 est le contenu de la case mémoire p et des 3 suivantes */
*(pi+1) = 5;            /* on range 5  4 cases mémoire plus loin   */
*(pi+2) = 0x1f;         /* on range 0x1f  8 cases mémoire plus loin */
*pr = 45.7;             /* 45,7 est rangé dans la case mémoire r et les 3 suivantes  */
pr++;                   /* incrémente la valeur du pointeur pr (de 4 cases mémoire) */
*pc = 'j';              /* le contenu de la case mémoire c est le code ASCII de 'j' */
```

Remarque :

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

5. Allocation dynamique

Lorsque l'on déclare une variable char, int, float .... Un nombre de cases mémoire bien défini est **réservé** pour cette variable. Il n'en est pas de même avec les pointeurs. Avant de manipuler un pointeur, il faut l'**initialiser soit par une allocation dynamique soit par une affectation d'adresse p=&i**. Sinon, par défaut la valeur du pointeur est égale à une constante symbolique notée NULL définie dans « stdio.h » En général, cette constante vaut 0.

On peut initialiser un pointeur **p** par une affectation sur **p**. Mais il faut d'abord réserver à **p** un espace mémoire de taille adéquate. L'adresse de cet espace mémoire sera la valeur de **p**. Cette opération consistant à réserver un espace mémoire pour stocker l'objet pointé s'appelle allocation dynamique. Elle se fait en C par la fonction **malloc** de la librairie standard **stdlib.h** Sa syntaxe est la suivante :

malloc (nombre_cases_mémoire)

Cette fonction retourne un pointeur de type char* pointant vers un objet de taille nombre_cases_mémoire. Pour initialiser des pointeurs vers des objets qui ne sont pas de type char*, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.

Exemples : la fonction malloc

```
char *pc;
int *pi,*pj,*pk;
float *pr;
pc = (char*)malloc(10);       /* on réserve 10 cases mémoire, soit la place pour 10 caractères */
pi = (int*)malloc(16);         /* on réserve 16 cases mémoire, soit la place pour 4 entiers  */
pr = (float*)malloc(24);       /* on réserve 24 places en mémoire, soit la place pour 6 réels */
pj = (int*)malloc(sizeof(int));  /* on réserve la taille d'un entier en mémoire */
pk = (int*)malloc(3*sizeof(int));          /* on réserve la place en mémoire pour 3 entiers */
```

Exemple 2 : la fonction malloc

Le programme suivant :

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

Void main()
{
```

```
        int   *p ;                  /* p est un  pointeur vers un objet de type int */
        printf( " la valeur de p avant initialisation est :%d ",p) ;
        p = (int*) malloc (sizeof(int)) ;
        printf( " \n la valeur de p après initialisation est :%x ",p) ;
        *p = 2;                     /* p contient la valeur 2 */
        printf( " \n la valeur de *p est :%d ",p) ;

}
```

Imprime :

la valeur de p avant initialisation est : 0
la valeur de p après initialisation est : ff11
la valeur de *p est : 2

Analyse de l'exemple :

Avant l'allocation dynamique, *p n'a aucun sens et toute manipulation de *p génèrerait une erreur 'segmentation fault'.
L'allocation dynamique a pour résultat d'attribuer une valeur à p et de réserver à cette adresse un espace mémoire composé de 4 octets pour stocker la valeur de *p.

Remarque1 :

Dans le programme :

```
#include <stdio.h>
main()
{
    int i ;
    int * p ;
    p=&i ;
}
```

i et *p sont identiques et on n'a pas besoin d'allocation dynamique puisque l'espace mémoire à l'adresse &i est déjà réservé pour un entier.

Remarque 2 :

La fonction calloc de la librairie stdlib.h a le même rôle que la fonction malloc mais elle initialise en plus l'objet pointé *p à 0. Sa syntaxe est la suivante :

Calloc(nb_objets, taille_objet)

Exemple :

```
Int n=10;
Int *p ;
p= (int*) calloc (n, sizeof(int));
```

Est equivalent à:

```
Int n=10;
Int *p ;
p= (int*) malloc (n * sizeof(int));
For (i=0;i<n;i++)  *(p+i)=0;
```

Enfin, lorsque l'on n'a plus besoin de l'espace mémoire allouée dynamiquement c'est-à-dire quand on n'utilise plus le pointeur p, il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction free qui a pour syntaxe :

free(nom_du_pointeur) ;

Libération de la mémoire : la fonction free

```
pi = (int*)malloc(16);          /* on réserve 16 cases mémoire, soit la place pour 4 entiers  */
pr = (float*)malloc(24);        /* on réserve 24 places en mémoire, soit la place pour 6 réels */

free(pi);                       /* on libère la place précédemment réservée pour i */
free(pr);                       /* on libère la place précédemment réservée pour r */
```

6. Pointeurs et tableaux :

6.1. Pointeur et tableau à une dimension :

Tout tableau en C est en fait un pointeur constant. Dans la déclaration :

```
int tab[10] ;
```

tab est un pointeur constant non modifiable dont la valeur est l'adresse du premier élément du tableau. Autrement dit **tab** a pour valeur **&tab[0]**. On peut donc utiliser un pointeur initialisée à tab pour parcourir les éléments du tableau.

Exemple :

```
main()
{
      int tab[5] = {2,1,0,8,4} ;
      int *p ;
      p= tab ;       /*  ou p= &tab[0]; */
      for (int i=0 ; i<5 ; i++)
      {
           printf ( " %d\n",*p);
           p++ ;
       }
}
```

Qui est équivalent à :

```
main()
{
      int tab[5] = {2,1,0,8,4} ;
      int *p ;
      p= tab ;
      for (int i=0 ; i<5 ; i++)
                printf ( " %d\n",p[i] );      /* car *(p+i) = p[i]   */
 }
```

Qui est équivalent à :

```
main()
{
      int tab[5] = {2,1,0,8,4} ;
      int *p ;
      for ( p=&tab[0] ; p<&tab[N]; p++)
            printf ( "%d \n",*p);
 }
```

6.2. Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions est par définition un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions définie par :

```
int tab[m][n] ;
```

tab est un pointeur qui pointe vers un objet qui est lui même de type pointeur d'entier. tab a une valeur constante égale à l'adresse du premier élément du tableau, &tab[0][0]. De même tab[i] pour i entre 0 et M-1, est un pointeur constant vers un objet de type entier qui est le premier élément de la ligne d'indice i. tab[i] a donc une valeur constante qui est égale à &tab[i][0]. ***(à voir)***

On déclare un pointeur qui pointe sur un objet de type (type *) de la même manière qu'un pointeur, c'est-à-dire :

type ** nom_du_pointeur ;

Exemple : (création d'une matrice k,n avec des pointeur sur des pointeur)

```
main()
{
      int  k,n ;
      int **tab ;
      tab = (int**) malloc(k* sizeof(int*)) ;
      for ( i=0;i<n;i++)
            tab[i] = (int*) malloc(sizeof(int));
       …….
       for ( i=0;i<n;i++)
             free(tab[i] ;
             free(tab)
 }
```

7. Tableau de pointeurs :

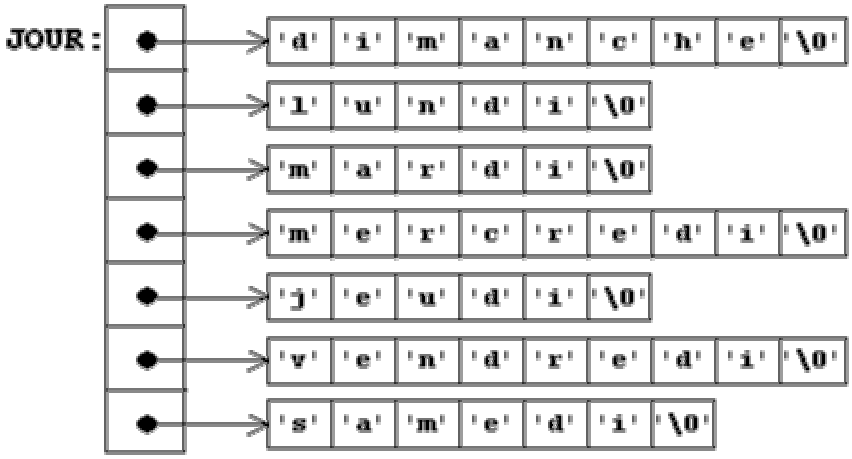
La déclaration d'un tableau de pointeur se fait comme suit :

Type * NomTableau [N]

C'est la déclaration d'un tableau NomTableau de N pointeurs sur des données du type **Type**.

Exemple :

```
char *JOUR[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};
/* affichage des 7 chaine de caractères */
for (int I=0; I<7; I++) printf("%s\n", JOUR[I]);
/* affichage des premières lettres des jours de la semaine */
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
/* affichage de la 3 eme lettre de chaque jourde la semaine */
for (I=0; i<7; I++) printf("%c\n",*(JOUR[I]+2));
```

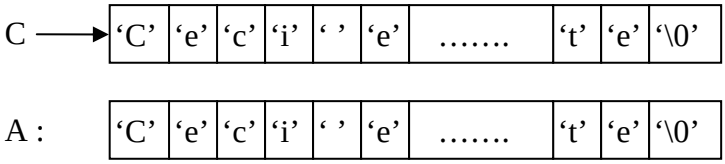


8. Pointeur et chaîne de caractère:

Une chaîne de caractères est un tableau à une dimension d'objets de type char se terminant par le caractère nul '\0'. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type char.

Exemple :

```
char *C;
char  A[] = " bonjour" ;
C = "Ceci est une chaîne de caractères constante";
```



Exemple :

```
main()
{
      char * chaine ;
      chaine = " chaine de caractères"
      for (int i=0 ; *chaine != '\0' ;
          chaine++ ;
      printf ( " la longueur de la chaine est %d\n",*p);
}
```

9. Pointeur et structures :

Supposons que l'on ait défini la `struct personne` à l'aide de la déclaration :

```
struct personne
  {
       char nom[20] ;
       unsigned int age ;
};
```

On déclarera une variable de type pointeur vers une telle structure de la manière suivante :

```
struct personne *p ;
```

On pourra alors affecter à p des adresses de `struct personne`. Exemple :

```
struct personne pers;   /*  pers est une variable de type struct personne   */
struct personne *p;     /*   p est un pointeur vers une struct personne      */
p =  &pers;
(*p).age=20 ;  /* parentheses obligatoires */
p -> age=22 ; /* -> opérateur d'accès */
```

9.1. Structures dont un des membres pointe vers une structure du même type

Une des utilisations fréquentes des structures, est de créer des listes de structures chaînées. Pour cela, il faut que chaque structure contienne un membre qui soit de type pointeur vers une structure du même type. Cela se fait de la façon suivante :

```
struct personne
{
    char nom[20] ;
    unsigned int age ;
    struct personne * suivant ;
};
```

Le membre de nom **`suivant`** est déclaré comme étant du type pointeur vers une **`struct personne`**. La dernière structure de la liste devra avoir un membre **`suivant`** dont la valeur sera le pointeur **`NULL`**.

9.2. Allocation et libération d'espace pour les structures

Quand on crée une liste chaînée, c'est parce qu'on ne sait pas à la compilation combien elle comportera d'éléments à l'exécution (sinon on utiliserait un tableau). Pour pouvoir créer des listes, il est donc nécessaire de pouvoir allouer de l'espace dynamiquement. On dispose pour cela de deux fonctions **malloc** et **calloc**.

La fonction **`malloc`** admet un paramètre qui est la taille en octets de l'élément désiré et elle rend un pointeur vers l'espace alloué. Utilisation typique :

```
struct personne *p ;
p = malloc(sizeof(struct personne));
```

10. Allocation d'un tableau d'éléments : fonction `calloc`

Elle admet deux paramètres : le premier est le nombre d'éléments désirés ;le second est la taille en octets d'un élément. son but est d'allouer un espace suffisant pour contenir les éléments demandés et de rendre un pointeur vers cet espace.

```
struct personne *p;
int nb_elem;
p = calloc(nb_elem,sizeof(struct personne));
```

On peut alors utiliser les éléments **`p[0]`**, **`p[1]`**, ... `p[nb_elem-1]`.

11. Libération d'espace : procédure free

On libère l'espace alloué par **malloc** ou **calloc** au moyen de la procédure **free** qui admet un seul paramètre : un pointeur précédemment rendu par un appel à **malloc** ou **calloc**.

```
struct personne *p;
p = malloc(sizeof(struct personne));
free(p);
```

Chapitre 12 : Les fichiers

Objectifs

- *Apprendre à manipuler des fichiers à partir de programme écrits en langage C*
- *Comprendre l'utilité des fichiers stockés sur des supports permanents par rapport aux fichiers standards.*

Elément de contenu

- *Définitions et propriétés des fichiers*
- *Manipulation des fichiers*
  - *Déclarer*
  - *Ouvrir*
  - *Fermer*
  - *Détruire*
  - *Renommer*
  - *Se positionner à l'intérieur du fichier*
  - *Changer le mode d'accès*
  - *Détection de la fin du fichier*
- *Lecture et écriture dans les fichiers*

1. Introduction

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, (clavier) et ils ont écrit leurs résultats dans le fichier de sortie standard (l'écran). Nous allons voir dans ce chapitre, comment nous pouvons créer, lire et modifier nous-mêmes des fichiers sur les périphériques disponibles.

2. Définition et propriétés :

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

2.1. Types d'accès :

- **Accès séquentiel** (surtout pour les bandes magnétiques)
  - ✓ Pas de cellule vide.
  - ✓ On accède à une cellule quelconque en se déplaçant depuis la cellule de départ.
  - ✓ On ne peut pas détruire une cellule.
  - ✓ On peut par contre tronquer la fin du fichier.
  - ✓ On peut ajouter une cellule à la fin.

- **Accès direct** (disques, disquettes, Cd-rom où l'accès séquentiel est possible aussi).
  - ✓ Cellule vide possible.
  - ✓ On peut directement accéder à une cellule.
  - ✓ On peut modifier (voir détruire) n'importe quelle cellule.

2.2. Codage

- En binaire : Fichier dit « binaire », Ce sont en général des fichiers de nombres. Ils ne sont pas listables.

- En ASCII : Fichier dit « texte », les informations sont codées en ASCII. Ces fichiers sont listables. Le dernier octet de ces fichiers est EOF (caractère ASCII spécifique).

2.3. Fichiers standard

Il existe deux fichiers spéciaux qui sont définis par défaut pour tous les programmes:

- ***stdin*** le fichier d'entrée standard, lié en général au clavier
- ***stdout*** le fichier de sortie standard, lié au clavier

3. La mémoire tampon

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une ***mémoire tampon***. La mémoire tampon est une zone de la mémoire centrale de la machine réservée à un ou plusieurs enregistrements du fichier. L'utilisation de la mémoire tampon a l'effet de réduire le nombre d'accès à la périphérie d'une part et le nombre des mouvements de la tête de lecture/écriture d'autre part.

4. Manipulation des fichiers :

Les opérations sur les fichiers les plus courant sont: Créer - Ouvrir - Fermer - Lire - Ecrire - Détruire - Renommer. Le langage **C** ne distingue pas les fichiers à accès séquentiel des fichiers à accès direct, certaines fonctions de la bibliothèque livrée avec le compilateur permettent l'accès direct. Les fonctions standard sont des fonctions d'accès séquentiel.

4.1. Déclaration :

```
FILE *fichier ;  /* majuscules obligatoires pour FILE */
```

On définit un pointeur qui va pointer vers une variable de type FILE qui est une srtucture (struct).

4.2. Ouverture : fopen

```
FILE *fopen(char *nom_fichier, char *mode_ouverture);
```

nom_fichier est une chaîne de caractères représentant le nom du fichier à ouvrir.
mode_ouverture est une chaîne représentant le mode d'ouverture du fichier. Elle peut être l'une des chaînes suivantes :

Mode (pour les fichiers TEXTES) :

- "r" ouverture en lecture seule.
- "w" ouverture en écriture seule.
- "a" ouverture en écriture à la fin.
- "r+" ouverture en lecture/écriture.
- "w+" ouverture en lecture/écriture.
- "a+" ouverture en lecture/écriture à la fin.

Mode (pour les fichiers BINAIRES) :

- "rb" ouverture en lecture seule.
- "wb" ouverture en écriture seule.
- "ab" ouverture en écriture à la fin.
- "r+b" ou "rb+" ouverture en lecture/écriture.
- "w+b" ou "wb+" ouverture d'un en lecture/écriture.
- "a+b" ou "ab+" ouverture en lecture/écriture à la fin.

A l'ouverture, le pointeur est positionné au début du fichier (sauf "a+" et "ab+")

Exemple :

```
FILE *fichier ;
fichier = fopen( "a :\fich.dat","  rb ") ;
```

4.2.1. Valeur rendue

La fonction **`fopen`** retourne une valeur de type pointeur vers **`FILE`**, où **`FILE`** est un type prédéfini dans le fichier **`stdio.h`**.

- Si l'ouverture a réussi, la valeur retournée permet de repérer le fichier, et devra être passée en paramètre à toutes les procédures d'entrées-sorties sur le fichier.

- Si l'ouverture s'est avérée impossible, **`fopen`** rend la valeur **`NULL`**.

4.2.2. Conditions particulières et cas d'erreur

- Si le mode contient la lettre **`r`**, le fichier doit exister, sinon c'est une erreur.

- Si le mode contient la lettre **`w`**, le fichier peut exister ou pas. Si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est perdu.
- Si le mode contient la lettre **`a`**, le fichier peut exister ou pas. Si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est conservé.

- Si un fichier est ouvert en mode « écriture à la fin », toutes les écritures se font à l'endroit qui est la fin du fichier au moment de l'exécution de l'ordre d'écriture. Cela signifie que si plusieurs processus partagent le même **`FILE *`**, résultat de l'ouverture d'un fichier en écriture à la fin, leurs écritures ne s'écraseront pas mutuellement. D'autre

part, si un processus ouvre un fichier en écriture à la fin, fait un `fseek` pour se positionner à un endroit du fichier, puis fait une écriture, celle-ci aura lieu à la fin du fichier (pas nécessairement à l'endroit du **`fseek`**).

Exemple typique :

```
FILE *fp;
if ((fp = fopen("donnees","r")) == NULL)
{
    fprintf(stderr,"Impossible d'ouvrir le fichier données en lecture\n");
    exit(1);
}
else  fprintf(stderr,  "ouverture avec succès\n " ) ;
```

4.3. Fermeture : fclose

```
int fclose(FILE *fichier);
```

Retourne 0 si la fermeture s'est bien passée, EOF en cas d'erreur.
Il faut toujours fermer un fichier à la fin d'une session. mode (pour les fichiers TEXTE) :

Exemple :

```
FILE *fichier ;
fichier = fopen( "a :\\ fich.dat ", " rb") ;
/* Ici instructions de traitement */
fclose(fichier) ;
```

4.4. Destruction : remove

```
int unlink (const char* nom_fichier) ;  /* ou bien */
int remove(const char *nom_fichier); /* macro qui appelle  unlink */
```

Permet de supprimer un fichier fermé. Retourne 0 en cas de succès.

Exemple :

```
remove ( "a :\\ fich.dat ");
```

4.5. Renommer: rename

```
int rename(char *oldname, char *newname);
```

Retourne 0 si la fonction s'est bien passée. Sinon la valeur -1.

4.6. Changer le mode d'accès : chmod

```
int chmod(const char *nom_fichier, int mode);
```

Modifie ou définie les droit d'accès à un fichier. « mode » est une constante (S_IREAD, S_IWRITE, S_IREAD,S_IWRITE)
Retourne la valeur 0 en cas de succès. Sinon la valeur -1

4.7. Positionnement du pointeur au début du fichier : rewind

```
void rewind(FILE *fichier);
```

Repositionne le pointeur du fichier sur le début d'un flux.

4.8. Positionnement du pointeur dans un fichier : fseek

```
Int fseek (FILE *fichier, long offset, int whence);
```

Fseek fait pointer le pointeur de fichier à la position situé offset octets au-delà de l'emplacement indiqué par whence (SEEK_SET (0), SEEK_CUR (1) et SEEK_END (2)) qui signifie respectivement : recherche à partir du début de fichier, position courante et fin de fichier.

4.9. Détection de la fin d'un fichier séquentiel : feof

Exemple :

```
FILE *fichier ;
fichier = fopen( "a :\\ fich.dat ", " r") ;
while (!feof(fichier)) ........
```

5. Lecture et écriture dans les fichiers séquentiels

Les fichiers que nous employons dans ce chapitre 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**, **fputc** et **fgetc** qui correspondent à **printf**, **scanf**, **putchar** et **getchar** si nous indiquons *stdout* respectivement *stdin* comme fichiers de sortie ou d'entrée.

5.1. Traitement par caractères

Lecture par caractère :

```
int fgetc(FILE *fichier); /*ou */
int getc(FILE *fichier) ; /* version macro de fgetc */
int getchar(void) ; /* renvoie un c depuis stdin, équivalente à getc(stdin) */
```

Renvoie le prochain caractère depuis le flux en entrée après conversion en int mais sans extension de signe. Si la fin de fichier est atteinte elle renvoie EOF.

Exemple :

```
FILE *fichier ; char c ;
fichier = fopen( "a :\\ fich.dat ", " rb") ;
while ((c = fgetc(fichier)) != EOF)   /* ou bien c=getc(fichier)  */
{
   ...   /*   utilisation de c   */
}
```

Ecriture par caractère :

```
int fputc(int c, FILE *fichier); /*ou */
int putc(int c, FILE *fichier) ; /* version macro de fgetc */
int putchar(int c) ; /* équivalente à putc(c, stdout) */
```

Injecte un caractère dans un flux. Renvoie le caractère injecté en cas de succès et EOF en cas d'erreur.

Exemple : copier un fichier dans un autre

```
FILE *fichier_source, *fichier_dest;
Char c ;
Fichier_source = fopen("c:\\autoexec.bat","r");
Fichier_dest = fopen("c:\\copie_autoexec.bat","w");
while ((c = fgetc(fichier_src)) != EOF)
   fputc(c,fichier_dest);
```

5.2. Traitement par chaîne de caractères :

Lecture par chaîne de caractère :

```
char *fgets(char * chaine, int n,FILE *fichier); /*ou */
char *gets(char * chaine);
```

fgets lit des caractères depuis le flux d'entrée fichier et les place dans la chaîne « chaine ». fgets cesse la lecture soit lorsque (n-1) caractères ont été lues ou jusqu'à rencontrer \n (qui est mis dans la chaine), ou jusqu'à ce qu'il ne reste plus qu'un seul caractère libre dans le tableau, ou rencontre de fin de fichier. fgets complète alors les caractères lus par un caractère '\0'.

gets lit des caractères depuis stdin et les place dans la chaîne « chaine »

Exemple :

```
char ligne[20];
FILE *fichier;
Fichier= fopen (a :\\ fich.dat ", " rb") ;
while (fgets(ligne,20,fichier) != NULL)   /*  stop sur fin de fichier ou erreur  */
{
   ...   /*   utilisation de ligne   */
}
```

Ecriture par chaîne de caractère :

```
int fputs(char * chaine , FILE *fichier);
int puts(char * chaine); /*  équivalente à fputs sur stdout */
```

`fputs` écrit sur le fichier le contenu du tableau de caractères dont la fin est indiquée par un caractère'\0'. Le tableau de caractères peut contenir ou non un'\n'. `fputs` peut donc servir indifféremment à écrire une ligne ou une chaîne quelconque. La fonction `fputs` rend une valeur non négative si l'écriture se passe sans erreur, et `EOF` en cas d'erreur

puts envoie une chaîne de caractères vers stdout et fait un saut de ligne. La fonction `fputs` rend une valeur non négative si l'écriture se passe sans erreur, et `EOF` en cas d'erreur.

Exemple :

```
char ligne[20];
FILE *fichier;
Fichier= fopen ("a :\\ fich.dat ", " a+") ;
fputs ("un texte",fichier) ;
```

5.3. E/S formatées sur les fichiers :

Lecture formatée à partir d'un fichier:

```
int  fscanf (FILE *fichier, const char *format, liste d'adresses);
int  scanf ( const char *format, liste d'adresses); /* lect à partir de stdin */
```

`fscanf` lit une suite de caractères du fichier défini par *fichier* en vérifiant que cette suite est conforme à la description qui en est fait dans *format*. Cette vérification s'accompagne d'un effet de bord qui consiste à affecter des valeurs aux variables pointées par les différents *paramètres* spécifiés par la liste d'adresse.

`fscanf` retourne le nombre de $param_i$ affectés. Si il y a eu rencontre de fin de fichier ou erreur d'entrée-sortie avant toute affectation à un $param_i$, `fscanf` retourne `EOF`.

La lecture d'un paramètre s'arrête quand on rencontre un caractère blanc (*espace, tab, line feed, new line, vertical tab* et *form feed*).

Lecture formatée à partir d'une chaine:

```
int  sscanf (const char *buffer, const char *format, liste d'adresses);
```

La fonction `sscanf` réalise le même traitement que la fonction `fscanf`, avec la différence que les caractères lus par `sscanf` ne sont pas lus depuis un fichier, mais du tableau de caractères *chaîne*. La rencontre du *null* terminal de *chaîne* pour `sscanf` est équivalente à la rencontre de fin de fichier pour `fscanf`.

Ecriture formatée dans un fichier:

```
int fprintf(FILE *fichier, char *format, liste d'expressions);
int printf(char *format, liste d'expressions); /* écriture sur stdout */
```

fprintf permet d'écrire dans le fichier la liste des expression selon les format spécifiés. Elle retourne le nombre de caractères écrits, ou une valeur négative si il y a eu une erreur d'E/S.

Ecriture formatée dans une chaine:

```
int  sprintf (const char *buffer, const char *format, liste d'adresses);
```

La fonction **sprintf** réalise le même traitement que la fonction `fprintf`, avec la différence que les caractères émis par `sprintf` n'est pas écrite dans un fichier, mais dans le tableau de caractères *chaîne*. Un *null* est écrit dans *chaîne* en fin de traitement.

Exemple : copie d'un fichier dans un autre. Le premier fichier contient deux colonnes (nom, moyenne)

```
#include <stdio.h>
#include <process.h>
void main()
{
  FILE * fichier_src, *fichier_dest;
  char nom[20];
  float moyenne;
  if( (fichier_src=fopen("c:\\test.txt","r"))==NULL)
  {
          printf("erreur d'ouverture u fichier source\n");
          exit (0);
  }
  if( (fichier_dest=fopen("c:\\test2.txt","w"))==NULL)
  {
          printf("erreur d'ouverture du fichier destination\n");
          exit (0);
  }
  while (!feof(fichier_src))
  {
          fscanf(fichier_src,"%s%f",&nom,&moyenne);
          fprintf(fichier_dest,"%s%f\n",nom,moyenne);
  }
}
```

Chapitre 13 : Les fonctions

Objectifs

- *Comprendre la notion de fonction en C*
- *Connaître la notion de prototype en C*
- *Différencier entre les variables globales et les variables locales*
- *Comprendre la notion de transmission d'arguments entre fonctions*
- *Connaître les classes d'allocation des variables*
- *Définir la portée d'une variable*

Eléments du contenu

- *Modularisation de programme*
- *Définition de fonctions*
- *Déclaration de fonctions*
- *Variable locales et variables globales*
- *Paramètres d'une fonction*
- *Passage de paramètres par valeur et passage de paramètre par adresse*

1. Introduction

La structuration de programmes en sous-programmes se fait en C à l'aide de ***fonctions***. Les fonctions en C correspondent aux fonctions *et* procédures en langage algorithmique. Nous avons déjà utilisé des fonctions prédéfinies dans des bibliothèques standard (**printf**, **pow…**). Dans ce chapitre, nous allons découvrir comment nous pouvons définir et utiliser nos propres fonctions.

2. Modularisation de programmes

Jusqu'ici, nous avons résolu nos problèmes à l'aide de fonctions prédéfinies et d'une seule fonction : la fonction principale **main()**. Pour des problèmes plus complexes, nous obtenons ainsi 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.

Pour remédier à ces problèmes, on subdivise nos programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts appelés modules. Dans ce contexte, un ***module*** désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

2.1. Avantages de la modularisation :

- Meilleure lisibilité,
- Diminution du risque d'erreurs,
- Possibilité de tests sélectifs,
- Réutilisation de modules déjà existants,
- Simplicité de l'entretien (changer un module),
- Favorisation du travail en équipe,
- Hiérarchisation des modules.

Un programme peut d'abord être résolu globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite. De cette façon, nous obtenons une hiérarchie de modules.

3. Définition de fonctions

Définition d'une fonction en algorithmique :

```
fonction  NomFonct  ( NomPar1 , NomPar2 , ...) : TypeResultat
   déclarations des paramètres
   déclarations locales
   instructions
finfonction
```

Définition d'une fonction en ANCI-C :

```
TypeRésultatRetourné  NomFonction (TypePar1  NomPar1, TypeParN NomParN,… )
{
    déclarations locales
    instructions
 }
```

Définition d'une fonction en K&R-C :

```
TypeRésultatRetourné  NomFonction (NomPar1, NomParN,… )
TypePar1  NomPar1, TypeParN NomParN,… ;
{
    déclarations locales
    instructions
 }
```

La définition des fonctions est la plus importante différence entre ANSI et K&R. Dans la version K&R du langage, le prototype (c'est à dire la liste-de-déclarations-de-paramètres) est remplacé par une liste d'identificateurs, qui sont les noms des paramètres. La déclaration des types des paramètres se fait juste après.

Exemple 1 : définition avec K&R-C

```
int somme_carre(i,j)    /*  liste des noms des paramètres formels     */
int i, int j;           /*  déclaration du type des paramètres        */
{
    int resultat;       /*  declaration locale */
    resultat = i*i + j*j;
    return(resultat);   /*  la valeur retournée, de type int */
}
```

Exemple 2 : définition avec ANCI-C

```
int somme_carre( int i , int j )   /*  liste des noms et types des paramètres formels     */
{
    int resultat;       /*  declaration locale */
    resultat = i*i + j*j;
    return(resultat);   /*  la valeur retournée, de type int */
}
```

Dans la définition d'une fonction, nous indiquons:

- le nom de la fonction
- le type 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

Remarque :

- Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction cache la fonction prédéfinie.
- Une fonction peut fournir comme résultat:
  - un type arithmétique
  - une structure (définie par **struct**),
  - une réunion (définie par **union**),
  - un pointeur,
  - **void** (la fonction correspond alors à une 'procédure')
- Une fonction **ne peut pas** fournir comme résultat des tableaux, des chaînes de caractères ou des fonctions. Mais, il est cependant possible de renvoyer un pointeur sur le premier élément d'un tableau ou d'une chaîne de caractères.
- Si une fonction ne fournit pas de résultat, il faut indiquer **void** (vide) comme type du résultat.
- Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme **(void)** ou simplement comme **()** .
- Le type par défaut est **int**; autrement dit: si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type **int**.

- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée ou définie avant d'être appelée.

4. Déclaration de fonctions

En C, 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. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects. Si dans le texte du programme la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

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.

```
TypeRésultat  NomFonction (TypePar1, TypePar2, ...);
```

Ou bien

```
TypeRésultat  NomFonction (TypePar1 NomPar1, TypePar  NomPar2, ...);
```

Remarque :

- Lors de la *déclaration*, le nombre et le type des paramètres doivent nécessairement correspondre à ceux de la *définition* de la fonction.
- **Noms des paramètres** : On peut facultativement inclure les *noms des paramètres* dans la déclaration, mais ils ne sont pas considérés par le compilateur. Les noms fournissent pourtant une information intéressante pour le programmeur qui peut en déduire le rôle des différents paramètres.
- **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.
  - **Déclaration multiple:** Une fonction peut être déclarée *plusieurs fois* dans le texte d'un programme, mais les indications doivent concorder.

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

5. Notion de bloc et portée des identificateurs

Les fonctions en C sont définies à l'aide de blocs d'instructions. Un bloc d'instructions est encadré d'accolades et composé de deux parties:

```
{
  déclarations locales
   instructions
}
```

Par opposition à d'autres langages de programmation, ceci est vrai pour ***tous*** les blocs d'instructions, non seulement pour les blocs qui renferment une fonction. Ainsi, le bloc d'instructions d'une commande **if**, **while** ou **for** peut théoriquement contenir des déclarations locales de variables et même de fonctions.

Exemple :

La variable d'aide I est déclarée à l'intérieur d'un bloc conditionnel. Si la condition (N>0) n'est pas remplie, I n'est pas défini. A la fin du bloc conditionnel, I disparaît.

```
if (N>0)
  {
   int I;
   for (I=0; I<N; I++)
     ...
  }
```

5.1. Variables locales :

Les variables déclarées dans un bloc d'instructions sont *uniquement visibles à l'intérieur de ce bloc*. On dit que ce sont des ***variables locales*** à ce bloc. Une variable déclarée à l'intérieur d'un bloc ***cache*** toutes les variables du même nom des blocs qui l'entourent.

5.2. Variables globales :

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.

6. Renvoi d'un résultat :

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, nous disposons de la commande **return**:

```
return <expression>;
```

Elle a les effets suivants:

- évaluation de l'<expression>
- conversion automatique du résultat de l'expression dans le type de la fonction
- renvoi du résultat
- terminaison de la fonction

7. Paramètres d'une fonction

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

8. Passage des paramètres par valeur

En langage C, les fonctions n'obtiennent que les *valeurs* de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes. Donc une fonction ne peut pas modifier la valeur des variables locales à main() ou à une autre fonction. Cependant, elle peut modifier le contenu de l'adresse de cette variable.

Les paramètres d'une fonction sont à considérer comme des ***variables locales*** qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel.

A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Exemple: fonction qui calcule le factoriel d'un nombre N>0

Au moment de l'appel, la *valeur* de L est copiée dans N. La variable N peut donc être décrémentée à l'intérieur de factoriel, sans influencer la valeur originale de L.

```
Long int  factoriel (int N)
{
    long int fact = 1 ;
while (N>0)
  {
   fact=fact*N;
   N - - ;
  }
 return (fact);
}
/* fonction qui calcule et affiche le factoriel des 10 premiers entiers */
void calcfact(void)
{
 int L;
 for (L=1; L<10; L++)
     printf("%d",factoriel(L));
}
```

9. Passage de l'adresse d'une variable (par valeur)

Comme nous l'avons constaté ci-dessus, une fonction n'obtient que les valeurs de ses paramètres. *Pour changer la valeur d'une variable de la fonction appelante*, il faut procéder comme suit:

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

Exemple: Fonction permettant d'échanger la valeur de 2 variables :

Syntaxe qui conduit à une erreur :

```
#include <stdio.h>
void PERMUTER (int X,int Y)
{
        int tampon;
        tampon = X;
        X = Y;
        Y = tampon;
}

void main()
{
        int A = 5 , B = 8;
        PERMUTER(A,B);
        printf(« A=%d\n », A) ;
        printf(« B=%d\n », B) ;
}
PASSAGE DES PARAMETRES
PAR VALEUR
```

Syntaxe correcte :

```
#include <stdio.h>
void PERMUTER (int *px,int *py)
{
        int tampon;
        tampon = *px;
        *px = *py;
        *py = tampon;
}

void main()
{
        int A = 5 , B = 8 ;
        PERMUTER(&A,&B);
        printf(« A=%d\n », A) ;
        printf(« B=%d\n », B) ;
}
PASSAGE DES PARAMETRES
PAR ADRESSE
```

Explication1: Lors de l'appel, les *valeurs* de A et B sont copiées dans les paramètres X et Y. PERMUTER échange bien contenu des variables *locales* X et Y, mais les valeurs de A et B restent les mêmes.

Explication2: Lors de l'appel, les *adresses* de A et B sont copiées dans les *pointeurs* px et py. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.

9.1. Passage de l'adresse d'un tableau à une dimension

Cela consiste à fournir ***l'adresse d'un élément du tableau***. En général, on fournit l'adresse du premier élément du tableau, qui est donnée par ***le nom du tableau***.

Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivi de crochets ou simplement par un pointeur sur le type des éléments du tableau.

Lors d'un appel, l'adresse d'un tableau peut être donnée par le nom du tableau, par un pointeur ou par l'adresse d'un élément quelconque du tableau.

Exemple:

```
int strlen(char *CH)
{
    int N;
    for (N=0; *CH != '\0'; CH++)
        N++;
    return N;
}
```

Exemple:

```
void LIRE_TAB(int N, int *PTAB)
{
     printf("Entrez %d valeurs : \n", N);
     while(N)
     {
          scanf("%d", PTAB++);
          N--
     }
}
```

Remarque :

Pour qu'une fonction puisse travailler correctement avec un tableau qui n'est pas du type **char**, il faut aussi fournir la dimension du tableau ou le nombre d'éléments à traiter comme paramètre, sinon la fonction risque de sortir du domaine du tableau.

9.2. Passage de l'adresse d'un tableau à deux dimensions

Une solution praticable consiste à faire en sorte que la fonction reçoive un pointeur (de type float*) sur le début de la matrice et de parcourir tous les éléments comme s'il s'agissait d'un tableau à une dimension N*M.

Exemple:

```
float SOMME(float *A, int N, int M)
{
 int I;
 float S;
 for (I=0; I<N*M; I++)
     S += A[I];
 return S;
}
```

Voici un programme faisant appel à notre fonction SOMME:

```
#include <stdio.h>
main()
{
 /* Prototype de la fonction SOMME */
 float SOMME(float *A, int N, int M);
 /* SOMME(&T[0][0], 3, 4) ); */
 /* Déclaration de la matrice */
 float T[3][4] = {{1, 2, 3, 4},
                  {5, 6, 7, 8},
                  {9,10,11,12}};
 /* Appel de la fonction SOMME */
 printf("Somme des éléments : %f \n",
                   SOMME((float*)T, 3, 4) );
 return 0;
}
```