Support de cours de Langage C

Parcours Sciences de l'Ingénieur
L2 S.I. – L3 E.E.A.et L3 I.E.

GENERALITES

☞ Historique

- Développé dans les années 1970 par Kernighan et Ritchie aux laboratoires Bell d'AT &T.
- Conçu pour réécrire en langage évolué le système d'exploitation UNIX (1er système d'exploitation) de manière à assurer sa portabilité.
- Actuellement utilisé pour développé des systèmes d'exploitation et pour écrire aussi bien des applications de calcul scientifique que de gestion.
- Le 1er langage à être quasiment disponible sur tout système programmable (PC, station, microcontrôleur …).
- Syntaxe utilisée par plusieurs langage de programmation comme Java par exemple.
- Extension au C++ en 1985 permettant la programmation objet.

☞ Un langage multi-niveaux

C'est un langage très utilisé dans l'industrie car il cumule les avantages d'un langage de haut-niveau (portabilité, modularité, etc...) et ceux des langages assembleurs proches du matériel.

- **Langage de haut niveau**
  - Programmation structurée (conçu pour traiter les tâches d'un programme en les mettant dans des blocs).
  - Programmation sous forme de fonctions (sous-programmes).

- **Langage de bas-niveau** (niveau matériel)
  - Conçu pour être facilement traduit en langage machine.
  - Gestion de la mémoire "à la main".

☞ Création d'un programme

- **Edition du programme** : écriture du programme source sous un éditeur de texte soit quelconque, soit spécialisé (l'éditeur généralement associé à l'environnement C utilisé) → création d'un ou de plusieurs fichiers source avec une extension ".c".

- **Compilation du programme** : traduction du programme source en langage machine ou code objet interprété par le processeur. Cette compilation s'effectue en deux étapes :

  - Le préprocesseur : cette phase examine toutes les lignes commençant par le caractère # et réalise des manipulations sur le code source du programme (substitution de texte, inclusion de fichiers, compilation conditionnelle).

  - Le compilateur : cette étape est effectuée par le compilateur qui réalise en fait une vérification syntaxique du code source et s'il n'y a pas d'erreur, il crée un fichier avec une extension ".obj". Le fichier objet est incomplet pour être exécuter car il contient par exemple des appels de fonctions ou des références à des variables qui ne sont pas définies dans le même fichier.

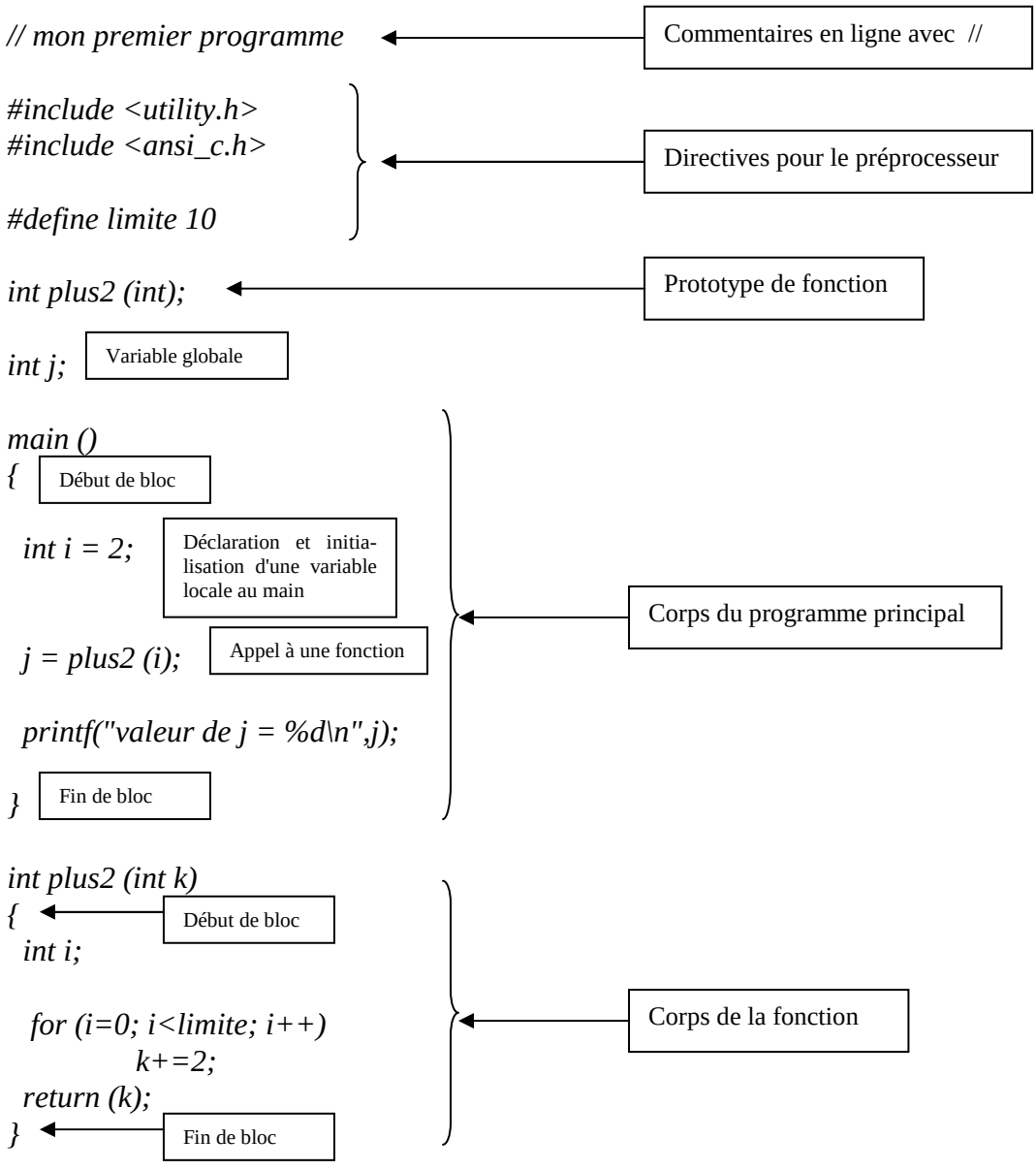
- **Edition des liens** : cette étape est effectuée par le linkeur qui réalise les liens entre différents programmes objets pour obtenir un programme exécutable. Plusieurs fichiers objets sont mis ensemble pour se compléter mutuellement pour produire un seul fichier exécutable (extension .exe).

☞ Structure d'un programme C

Afin d'écrire des programmes en langage C bien lisibles, il est important de respecter un certain nombre de règles de présentation qui sont :

- ne jamais placer plusieurs instructions sur une même ligne
- utiliser des identificateurs significatifs
- laisser une ligne blanche entre la dernière ligne des déclarations de variables et la première ligne des instructions
- une accolade fermante est seule sur une ligne et fait référence, par sa position horizontale, au début du bloc qu'elle ferme
- aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces
- commenter judicieusement les programmes

Exemple de programme C :



☞ Principales caractéristiques

- Programmation modulaire multi-fichiers : un ensemble de programmes déjà mis au point pourra être réuni pour constituer une librairie.

- Langage déclaratif c'est-à-dire que tout objet (variables, fonctions) doit être déclaré avant d'être utilisé.

- Langage transportable c'est-à-dire séparation entre ce qui est algorithmique (déclarations, instructions) et tout ce qui est en interaction avec le système (allocation mémoire, entrées-sorties).

- Jeu d'opérateurs très riche.

- Faible contrôle des types de données.

I – ELEMENTS DE BASE

I - Commentaires

Il est possible et vivement conseillé de documenter un programme en y intégrant des commentaires. Les commentaires sont non seulement utiles mais nécessaires à la compréhension du programme.

Deux types de commentaires :

- Commentaires par bloc en utilisant les caractères suivants :/* ….. */
- Commentaires par ligne en utilisant le caractère suivant : //

Les commentaires ne peuvent pas être imbriqués et ne sont pas pris en compte par le compilateur.

II – Les variables

Les variables servent à stocker des données manipulées par le programme. Elles sont nommées par des identificateurs alphanumériques.

On distingue :

- Les variables simples ou scalaires (entières, réelles et pointeurs).
- Les variables structurées (tableaux, structures, unions et listes chaînées).

II.1 - Déclaration de variables

Le langage C étant un langage déclaratif, toutes les variables doivent être déclarées avant utilisation. La déclaration se fait à l'aide de 2 paramètres :

<type> <identificateur>

II.1.1 - Identificateur

Un identificateur est un nom donné aux diverses composantes d'un programme ; variables, tableaux, fonctions.

- Il est formé de lettres et de chiffres. Le 1er caractère doit obligatoirement être une lettre ou bien le caractère _.
- Il peut contenir jusqu'à 31 caractères minuscules et majuscules mais pas de caractères accentués (codes ASCII étendus).
- Il est d'usage de réserver les identificateurs en majuscules aux variables du préprocesseur.
- Il y a un certain nombre de noms de variables réservés. Leur fonction est prévue par la syntaxe du langage C et ils ne peuvent pas être utilisés dans un autre but.

☞ **Liste des mots réservés**

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

II.1.2 – Types de variables

Toutes les variables sont typées. Il y a principalement deux types de base que l'on identifient à l'aide de mots clés :

- Le type entier (char, short, int, long)
- Le type flottant (float, double)

A partir de ces types de base, il est possible de définir des types dérivés tels que les tableaux, les pointeurs, les structures, les unions et les listes chaînées.

L'utilisation de ces mots clés influe directement sur la taille de la zone mémoire qui est allouée et qui est destinée à stocker la valeur qui est affectée à la variable. Les tableaux suivant donne les informations sur le codage des entiers et des flottants en fonction de la déclaration utilisée pour un environnement sous windows. Atention, ce codage dépend de l'architecture matérielle utilisée.

☞ **Les entiers**

Les deux mots clés **unsigned** et **signed** peuvent s'appliquer aux types "caractère" et "entier" pour indiquer si le bit de poids fort doit être considérer ou non comme un bit de signe. Les entiers sont signés par défaut. Selon la valeur que l'on est amené à affecter à un entier et la présence ou non du signe, on peut opter pour déclarer des **int, short, unsigned int** …

Le langage C distingue donc plusieurs types d'entiers :

TYPE	DESCRIPTION	TAILLE MEMOIRE	VALEURS
int	entier standard signé	4 octets	$[-2^{31} , 2^{31}-1]$
unsigned int	entier positif strictement	4 octets	$[0 , 2^{32}-1]$
short	entier court signé	2 octets	[-32768 , 32767]
unsigned short	entier court non signé	2 octets	[0 , 65535]
char	caractère signé	1 octet	[- 128 , 127]
unsigned char	caractère non signé	1 octet	[0 , 255]
long	entier long signé	8 octets	$[-2^{63} , 2^{63}-1]$
unsigned long	entier long non signé	8octets	$[0 , 2^{64}-1]$

Attention, pour certains systèmes ou compilateurs, le type int par exemple possède le même codage qu'un type short. Cela peut donc poser un problème de portabilité : le même programme, compilé sur deux machines distinctes, peut avoir des comportements différents.

Le type **char** définit des variables de type caractère et ne stocke qu'un seul caractère à chaque fois. Un caractère peut être une lettre, un chiffre ou tout autre élément du code ASCII et peut être un caractère non imprimable dit de contrôle : **0 < ASCII <31**. Un **CHAR** est codé sur **1 octet** et ne peut prendre que des valeurs.

- positives entre **0 et 255** s'il est non signé
- négatives entre **-127 et +127** s'il est signé

A propos des booléens : en langage C, il n'existe donc pas de type booléen spécifique. Il faut savoir que lorsqu'une expression (typiquement, dans des instructions comme if ou while) est vraie alors elle est considérée comme non nulle et dans le cas contraire, elle produit la valeur 0.

☞ **Les réels ou les flottants**

Rappel : un réel (généralement sur 4 octets) est composé d'un signe, d'une mantisse et d'un exposant. La mantisse correspond à la partie fractionnaire du nombre (généralement <1) et l'exposant correspond à la puissance à laquelle est élevée une base pour être multipliée par la mantisse.

Selon la valeur que l'on est amené à affecter à un nombre à virgule flottante simple ou double précision, on peut opter pour **float, double** ou **long double**. Le langage C distingue donc 3 types de réels :

TYPE	DESCRIPTION	TAILLE MEMOIRE	MANTISSE (nombre de chiffres significatifs)	VALEURS min et max positives
float	réel standard	4 octets	6	$3.4 * 10^{-38}$ $3.4 * 10^{38}$
double	réel double précision	8 octets	15	$1.7 * 10^{-308}$ $1.7 * 10^{308}$
long double	réel long double précision	10 octets	19	$3.4 * 10^{-4932}$ $3.4 * 10^{4932}$

II.1.3 – Exemples de déclaration

Une valeur initiale peut être spécifiée dès la déclaration de la variable.

```
main()
{
 int i;              // i : variable entière signée
 int i, j;           // i et j : variables entières signées
 unsigned long k;    // i : variable entière de type long non signé
 float x;            // x : variable flottante simple précision
 double y;           // y : variable flottante double précision
 unsigned char c;    // c : variable de type caractère non signé
}
```

II.2 - Initialisation de variables

Il y a deux manières d'affecter une valeur à la variable :

☞ **utilisation du signe "="** : le signe " = " désigne l'opération d'assignation ou d'affectation d'une valeur à une variable. Ce n'est pas une égalité au sens mathématique. Le sens exact est : prends le terme qui est à droite du signe "=" est mets la dans la variable qui est à gauche. En fait la valeur est introduite dans la case mémoire associée à la variable.

Exemple :

```
main()
{
 int i = 2;                 // i : variable entière initialisée à la valeur 2
 float x = 4.3;             // x : variable flottante initialisée à la valeur 4.3
 float  y = 54e-01;         // y : variable flottante initialisée avec un format scientifique (5.4)
 double z = 5.4;            // z : variable flottante double précision initialisée à la valeur 5.4
 unsigned char c = 'A';     // c : variable de type caractère initialisée au caractère 'A'
}
```

☞ **utilisation d'instructions spécifiques (scanf, getchar, gets)** : ces instructions permettent l'affectation de données à des variables par saisie au clavier.

Il est possible d'enchaîner plusieurs assignations en écrivant : i = j = 2; (2 dans j puis dans i).
Si on déclare : const int n=20; alors cela empêche toute modification future de la variable n.

II.3 – Conversion de type

☞ **conversion implicite**

Le langage C possède ses règles de conversion de type de données à l'intérieur d'une expression. Le but de ces conversions est d'obtenir une meilleure précision du résultat. Le principe général est que la conversion implicite se fait lors d'assignations vers le type "le plus grand" mais cette règle est parfois dangereuse !!!

Exemple :

```
main ()
{
 short i = 259, j = 10;
 char ch;
 float x;

ch = i;          // entier tronqué, perte de l'octet le plus significatif (MSB) du short (
                 // 257 = 0x0103  ->  dans ch, on a : 0x03
ch = 100;
 i = ch;         // pas de perte d'information, le char se retrouve dans l'octet le moins
                 // significatif (LSB) du short, i = 100

x = i;           // pas de perte d'informations, x = 100.0
x = 100.2;
i = x;           // réel tronqué car perte de la partie décimale, i = 100
}
```

Autres exemples de conversions automatiques :

- int ← long : conservation du bit de signe et donc la valeur de l'entier
- long ← int : résultat tronqué, perte des 2 octets les plus significatifs
- float ← double : aucun problème
- double ← float : perte de précision

☞ **conversion explicite**

On peut explicitement demander une conversion dans un type désiré. Cette opération s'appelle casting ou transtypage. Une expression précédée par un nom de type entre ( ) provoque la conversion de celle ci dans le type désiré.

Exemple :

```
main ()
{
 short i = 259, j = 10;
float x;

x = i / j;              // x = 25.0, le résultat de 2 entiers et un entier qui est mis dans un float
x = (float) i / j;      // x = 25.9, le contenu de i est converti au format d'un float pour l'opération
}
```

III – Les constantes

Le langage C permet d'utiliser des constantes numériques dans le code source. Chaque constante a une valeur et un type. Les expressions constantes sont évaluées à la compilation.

Les formats de ces constantes sont précisés ci-dessous.

III.1 - Les constantes entières

notation	syntaxe	
décimal	123 -123	succession de chiffres éventuellement précédée d'un signe mais **pas de suffixe**
octal	**0**777	succession de chiffres commençant par **0**
hexadécimal	**0x**ff ou **0X**ff	succession de chiffres précédée des caractères **0x** ou **0X**
long	65538**L**	on suffixe la variable par **l** ou **L**.
non signé	255**U** 255**UL**	on suffixe la variable par **l** ou **L**

III.2 - Les constantes flottantes

Les constantes réelles sont par défaut de type double. Elles peuvent être exprimées sous les formes suivantes :

Notation	syntaxe	
Double	3.14159 314159e-1	double précision, notation décimale notation scientifique
long double	3.14159**L**	impose le type long double par le suffixe **L**
simple précision	3.14159**F**	simple précision, notation décimale

III.3 - Les constantes caractères

Une constante caractère s'écrit entourée du signe '. Par exemple, la constante caractère correspondant au caractère A s'écrit 'A'. Les cas particuliers sont traités par une séquence d'échappement introduite par le caractère '\'. Comme pour les constantes entières, on peut utiliser les notations octales ou hexadécimales par exemple '\008' ou '\x3A'.

Il existe des caractères non affichables dits caractères de contrôle indiqués ci-dessous :

Caractère	code ASCII correspondant	Sémantique
'\n'	0x0a	LF (line feed) : saut de ligne
'\t'	0x09	HT tabulation horizontale
'\v'	0x0b	VT : tabulation verticale
'\r'	0x0d	CR (carriage return) : retour charriot
'\b'	0x08	BS (back space) : espace arrière
'\f'	0x0c	FF (feed form) : saut de page
'\a'	0x07	BELL : signal sonor
' \ '	0x5c	\ : backslash
' " '	0x22	" : guillemets

III.4 - Les constantes chaînes de caractères

Une constante chaînes de caractères est une suite de caractères entourée du signe ". Toutes les notations de caractères (normale, octale, hexadécimale) sont utilisables dans les chaînes de caractères.

Exemple : "Bonjour".

On verra que les constantes de type chaînes de caractères sont considérées comme des tableaux. Un caractère null '\0' est ajouté automatiquement à la fin de la chaîne.

III.5 - Les constantes nommées

Il y a deux façons de donner un nom à une constante : soit en utilisant les possibilités du préprocesseur, soit en utilisant des énumérations.

☞ #define

Lorsque le préprocesseur lit une ligne du type **#define < identificateur > < valeur >**, il effectue dans tout le code source une substitution de l'identificateur par la valeur indiquée.

Par exemple on peut écrire **: #define PI 3.14159**

Ainsi pour développer le programme, on pourra utiliser le nom PI pour désigner la constante 3.14159. Il s'agit d'une commande du préprocesseur et par conséquent il n'y a pas de ";" à la fin de la ligne.

☞ Les énumérations

On peut définir des constantes de la manière suivante : enum { liste des identificateurs }. Par exemple :

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

définit les identificateurs LUNDI, ... DIMANCHE comme étant des constantes de type int, et leur donne les valeurs 0, 1, ... 6. Par défaut, le premier identificateur est associé à la valeur 0.

On peut donner des valeurs particulières aux constantes en écrivant par exemple :

enum {LUNDI = 20, MARDI = 30, MERCREDI, JEUDI = 40, VENDREDI, SAMEDI, DIMANCHE};

Il n'est pas nécessaire de donner une valeur à toutes les constantes. Dans ce cas, LUNDI est associé à la valeur 20, MARDI à 30, MERCREDI à 31, JEUDI à 40, VENDREDI à 41, SAMEDI à 42 et DIMANCHE à 43.

IV – Les opérateurs

Le langage C comporte de très nombreux opérateurs.

IV.1 - Les opérateurs arithmétiques

Il existe 5 opérateurs arithmétiques, l'addition (+), la soustraction (-), la multiplication (*), la division (/) et le reste de la division entière, l'opérateur modulo (%).

Leurs opérandes peuvent être des entiers ou des flottants hormis pour l'opérateur % qui agit uniquement sur des entiers.

Remarque : lorsque les types des deux opérandes sont différents alors il y a conversion implicite dans le type le plus fort suivant certaines règles.

IV.2 - Les opérateurs logiques

Le type booléen n'existe pas. Le résultat d'une expression logique vaut 1 si elle est vraie et 0 sinon. Réciproquement, toute valeur non nulle est considérée comme vraie et la valeur nulle comme fausse. Les opérateurs logiques comprennent :

☞ 4 opérateurs relationnels

- inférieur à (<)
- inférieur ou égal à (<=)
- supérieur à (>)
- supérieur ou égal à (>=)
- l'opérateur de négation (!)

☞ 2 opérateurs de comparaison

- identique à (= =)
- différent de(!=)

☞ 3 opérateurs de conjonction

- le ET logique (&&)
- le OU logique (||)
- le NON (!)

expr1	expr2	! expr1	(expr1) && (expr2)	(expr1) || (expr2)
fausse → 0	fausse → 0	vraie →1	0	0
0	non nulle	1	0	1
non nulle	0	0	0	1
non nulle	non nulle	0	1	1

En effet,

- (!expr1) est vrai si expr1 est fausse et faux si expr1 est vraie.
- (expr1) && (expr2) est vraie si les deux expressions expr1 et expr2 sont vraies et fausse sinon. L'expression expr2 n'est évaluée que dans le cas où l'expression expr1 est vraie.
- (expr1) || (expr2) est vraie si l'une au moins des expressions expr1 ou expr2 est vraie et fausse sinon. L'expression expr2 n'est évaluée que dans le cas où l'expression expr1 est fausse.

Exemples :

```
main()
{
 int i = 5;
 float f = 5.5;
 char c = 'A';

 f > 3 → vrai (1)
 (i + f) <= 1 → faux (0)
 c != 'A' → faux (0)
 (i >= 2) && (c == 'A') → vrai (1)
 (i >= 2) || (c == 'A') → vrai (1)
}
```

IV.3 - Les opérateurs de traitement binaire

Ces opérateurs opèrent bit à bit et s'appliquent à des opérandes de type entier (ou caractère) et de préférence unsigned (sinon, ils risquent de modifier le bit de signe). Ils procurent des possibilités de manipulation de bas-niveau de valeurs, traditionnellement réservées à la programmation en langage assembleur.

opérateurs	opérations
~	négation bit à bit (unaire) complément à 1
<<	décalage à gauche
>>	décalage à droite
&	"ET" bit à bit
|	"OU" inclusif bit à bit
^	"OU" exclusif bit à bit

Exemple :

```
main()
{
 char ch = 0x81;
 ch = ch >> 4;          →   décalage du contenu de ch de 4 bits à droite
 ch = ch << 4;          →   décalage du contenu de ch de 4 bits à gauche
}

main()
{
 char ch = 0x81;
 ch = ch & 0xf0;        →   positionnement d'un masque pour éliminer les 4 bits de poids
}                           faibles de la variable ch
```

Remarques :

Dans le cas d'un décalage à gauche les bits les plus à gauche sont perdus. Les positions binaires rendues vacantes sont remplies par des 0.

Dans le cas d'un décalage à droite, les bits les plus à droite sont perdus. Si l'entier à décaler est non signé, les positions binaires rendues vacantes sont remplies par des 0. S'il est signé le remplissage s'effectue à l'aide du bit de signe.

IV.4 - Les opérateurs d'affectation

A part le signe "=", il y a des opérateurs d'affectation combinés qui modifient la valeur courante de la variable intervenant dans l'évaluation de l'expression.

opérateurs	opérations équivalentes
+=	i = i + 20, on peut écrire i += 20 i = i + k, on peut écrire i += k
-=	i = i - 20, on peut écrire i -= 20
*=	i = i * 3, on peut écrire i *= 3
/=	i = i / 3, on peut écrire i /= 3
i++	peut s'écrire i = i+1 (post-incrémentation) incrémentation après utilisation de la valeur
++i	pré-incrémentation c'est-à-dire incrémentation avant utilisation de la valeur
i--	peur s'écrire i = i-1 (post-décrémentation)
--i	pré-décrémentation

Les opérateurs d'incrémentation (++) et de décrémentation (--) sont des opérateurs unaires permettant respectivement d'ajouter et de retrancher 1 au contenu de leur opérande. Cette opération est effectuée après ou avant l'évaluation de l'expression suivant que l'opérateur suit ou précède son opérande.

IV.5 – L'opérateur conditionnel

L'opérateur conditionnel (? :) est un opérateur ternaire. Ses opérandes sont des expressions. La syntaxe est: expr1 ? expr2 : expr3

La valeur de l'expression expr1 est interprétée comme un booléen. Si elle est vraie, c'est-à-dire non nulle, seule l'expression expr2 est évaluée sinon c'est l'expression expr3 qui est évaluée. La valeur de l'expression conditionnelle est la valeur de l'une des expressions expr2 ou expr3 suivant que l'expression expr1 est vraie ou fausse.

IV.6 - L'opérateur de taille

L'opérateur sizeof renvoie la taille en octets de son opérande. L'opérande est soit une expression soit une expression de type. La syntaxe est : **sizeof (expression)**. L'opérateur sizeof appliqué à une chaîne de caractères vaudra sa taille y compris le caractère de fin de chaîne ('\0').

Exemple :

sizeof(int) : retourne en octet le codage d'une variable de type int c'est-à-dire retourne la valeur 4.

IV.7 - Priorités des opérateurs

La priorité et l'associativité des opérateurs du langage C sont donnés dans le tableau ci-dessous. Les opérateurs de priorité 1 sont les opérateurs de plus forte priorité. Les parenthèses permettent d'outrepasser les priorités.

CATEGORIE		OPERATEURS	ASSOCIATIVITE
Opérateurs arithmétiques		* / % + -	→
Opérateurs logiques	Opérateurs Relationnels	< <= > >=	→
	Opérateurs de Comparaison	Identique à : = = Diférent de : !=	→
	Opérateurs de Conjonction	ET logique : && OU logique : || NON : !	→
Opérateurs de traitement binaire		Négation : ~ Décalage à droite : >> Décalage à gauche : << ET binaire : & OU binaire : | OU binaire exclusif : ^	→
Opérateurs d'affectation		= += -= *= /= %= &= ^= |= <<= >>= ++ --	←
Opérateur conditionnel		? :	→
Opérateurs unaires		Conversion : (cast) Taille de : sizeof (…)	←
Opérateurs de référence		( ) [ ] ->	→
Opérateur séquentiel		,	→

Exemples :

- a & b == c ⇔ a & (b == c) et non (a & b) == c.
- a << 4 + b ⇔ a << (4 + b) et non (a << 4) + b

V – Les entrées – sorties standards

La manière standard de rentrer des données dans un programme est d'utiliser le clavier et la manière standard de récupérer des résultats consiste à utiliser l'écran. En fait, il s'agit d'un cas particulier d'entrées/sorties sur des fichiers que l'on verra par la suite.

☞ sortie formatée – printf

La fonction printf permet d'afficher à l'écran des chaînes de caractères et des données numériques. Sa syntaxe est la suivante : *int printf(const char *format, arguments)*

Elle admet un nombre quelconque d'arguments. char *format est une chaîne de caractères qui contient le texte d'accompagnement et le format de la variable à afficher (caractère %) avec éventuellement un caractère de contrôle. Les types de conversion les plus usuels sont donnés dans le tableau ci-dessous.

DECLARATION	LECTURE	AFFICHAGE	FORMAT EXTERNE
int i	scanf("%d", &i)	printf("%d", i)	décimal
int i	scanf("%x", &i)	printf("%x", i)	hexadécimal
unsigned int i	scanf("%u", &i)	printf("%u", i)	décimal
short i	scanf("%hd", &i)	printf("%d", i)	décimal
short i	scanf("%hx", &i)	printf("%x", i)	hexadécimal
unsigned short i	scanf("%hu", &i)	printf("%u", i)	décimal
long i	scanf("%ld", &i)	printf("%ld", i)	décima
long i	scanf("%lx", &i)	printf("%lx", i)	hexadécimal
unsigned long i	scanf("%lu", &i)	printf("%lu", i)	décimal
float x	scanf("%f", &x)	printf("%f", x)	point décimal
float x	scanf("%e", &x)	printf("%e", x)	scientifique
float x		printf("%g", x)	la plus courte des deux
double x	scanf("%lf", &x)	printf("%f", x)	point décimal
double x	scanf("%le", &x)	printf("%e", x)	scientifique
double x		printf("%g", x)	la plus courte des deux
long double x	scanf("%lf", &x)	printf("%lf", x)	point décimal
long double x	scanf("%le", &x)	printf("%le", x)	scientifique
long double x		printf("%lg", x)	la plus courte des deux
char ch	scanf("%c", &ch) ch = getchar()	printf("%c", ch) putchar(ch)	caractère
char tab[10]	scanf("%s", tab) gets(tab)	printf("%s", tab) puts(tab)	chaîne de caractères

☞ entrée formatée – scanf

La fonction scanf fournit des possibilités de conversions semblables à printf mais pour des entrées. Sa syntaxe est: *int scanf(const char *format, &arguments)*

scanf effectue à la fois la saisie de la donnée au clavier (codes ASCII) et la conversion. Les arguments (autres que format) de scanf doivent être des adresses de variables (ou des pointeurs) d'où utilisation de l'opérateur adresse &.

Exemple

```
#include<stdio.h>

main()
{ int i = 3;
  float x;

  printf("Donner une valeur pour x : "); scanf("%f",&x);
  printf("i = %d    x = %f  : \n", i, x);
}
```

☞ cas particulier pour les char

L'unité de base est le caractère codé sur un octet. Les deux fonctions présentées sont souvent des macro-expressions du préprocesseur mais on les considère comme des fonctions pour simplifier.

int getchar(void) est une fonction qui permet de lire un caractère dans le buffer associé au clavier. getchar met le programme en attente de lecture si aucun caractère n'est disponible dans le tampon d'entrée. Sinon, il lit les caractères dans ce buffer. Ce caractère est considéré comme étant du type unsigned char. La fonction retourne le caractère lu après conversion en entier. En cas d'erreur, elle retourne EOF.

int putchar(int) est une fonction qui permet d'écrire un caractère à l'écran. Elle est définie comme recevant un entier pour être conforme à getchar(). Ceci permet d'écrire putchar(getchar()). Elle retourne la valeur du caractère écrit toujours considéré comme un entier. En cas d'erreur, la fonction retourne EOF.

☞ cas particulier pour les chaînes de caractères

Une chaîne de caractères est considérée comme une suite de caractères terminée par un caractère de fin de ligne "\0".

char *gets(char *) est une fonction qui permet la lecture d'une chaîne de caractères contenant des espaces et des tabulations. La lecture ne s'arrête qu'à la réception d'un retour-chariot, lequel est remplacé par le caractère nul ('\0'). Le programmeur doit s'assurer que la zone mémoire désignée par l'argument d'entrée (char *) soit de taille suffisante pour mémoriser les caractères lus. Elle renvoie l'adresse du 1er octet de la zone mémoire où se trouvent les caractères.

int puts(char *) est une fonction qui permet de saisir une chaîne de caractères suivie d'un retour chariot. Elle retourne e nombre de caractères transmis et en cas d'erreur, elle retourne la valeur EOF.

II - STRUCTURES DE CONTROLE

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

I – Les instructions conditionnelles

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

I.1 – L'instruction if else

Syntaxe :

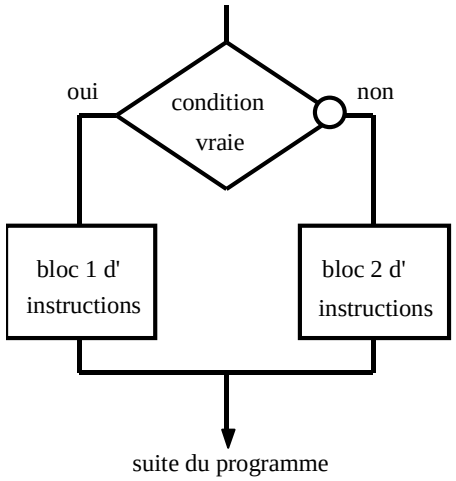
```
if (expression conditionnelle)
{
   .....;          /* bloc 1 d'instructions */
   .....;
}
else
{
   .....;          /* bloc 2 d'instructions */
   .....;
}
```

La valeur de l'expression conditionnelle est évaluée. Si elle est vraie (non nulle), alors le bloc 1 d'instructions est exécuté. Si elle est fausse (nulle), le bloc 2 d'instructions est exécuté.

La clause else est facultative c'est-à-dire que l'on peut se passer d'alternative à notre condition. Dans le cas d'imbrication de plusieurs **if**, la clause **else** se rapporte au **if** le plus proche.

Les {} ne sont pas nécessaires lorsque les blocs ne comportent qu'une seule instruction.

Organigramme :



Rappels sur les opérateurs

- *Egalité* if (a == b) ***« si a égal b »***
- *Non égalité* if (a != b) ***« si a différent de b »***
- *Relation d'ordre* if (a < b), if (a <= b), if (a > b), if (a >= b)
- *ET condition* if ((expression1) && (expression2))
  "si l'expression1 ET l'expression2 sont vraies"
- *OU condition* if ((expression1) || (expression2))
  "si l'expression1 OU l'expression2 est vraie"
- *Non Logique* if (!(expression1))
  "si l'expression1 est fausse"

Toutes les combinaisons sont possibles entre ces tests.

Il est possible d'effectuer des incrémentations et des décrémentations dans une condition.

Exemples :

if (--i==5) : i est d'abord décrémenté et le test i==5 est effectué
i=4;
if (i++==5) printf("EGALITE (%d) ",i);
else printf("INEGALITE (%d) ",i);
le test i==5 est d'abord effectué, i est incrémenté avant même d'exécuter le printf;

I.2 – L'instruction switch - case

Pour éviter les imbrications d'instructions **if**, le langage C possède une instruction qui crée une table de branchement : c'est l'instruction **switch**. L'instruction **switch** permet de faire un choix multiple et donc d'effectuer un aiguillage direct vers les instructions en fonction de l'évaluation d'une expression.

```
Syntaxe :        switch (expression à évaluer) {
                     case val1 :    bloc 1 d'instructions
                                    break;
                     case val2 :    bloc 2 d'instructions
                                    break;
                     default :      bloc 3 d'instructions
                 }
```

L'évaluation de l'expression doit impérativement donner un résultat de type entier. Les termes val1, val2 … doivent être des expressions constantes de type entier ou char. L'exécution des instructions se fait à partir du **case** dont la valeur correspond à l'expression jusqu'à une instruction **break**.

Le bloc 3 d'instructions qui suit la condition **default** est exécuté lorsqu'aucune constante des **case** n'est égale à la valeur retournée par l'expression.

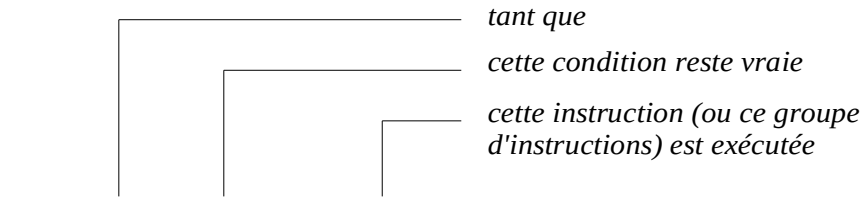
II – Les instructions itératives

Les instructions itératives sont gérées par trois types de boucles : le **while**, le **do while** et le **for**.

II.1- La boucle tant que (while)

Cette instruction permet de répéter une instruction (ou un bloc d'instructions) tant qu'une condition est vraie.

Syntaxe :




```
while (condition) instruction;
ou
while (condition) { instruction 1;
                    . . . .
                    instruction N; }
```

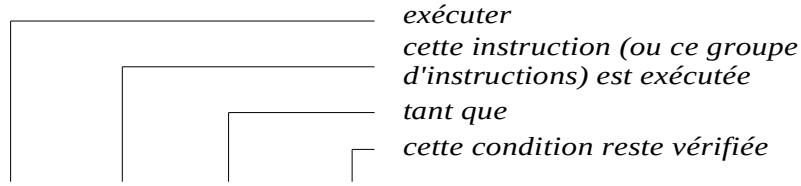
Ici, l'instruction est exécutée de façon répétitive aussi longtemps que le résultat de expression est vraie c'est-à-dire non nul.

La condition est examinée avant. La boucle peut ne jamais être exécutée.

II.2- La boucle faire .... tant que (do while)

Cette instruction permet de répéter une instruction (ou un bloc d'instructions) jusqu'à ce qu'une condition soit vraie.

Syntaxe :

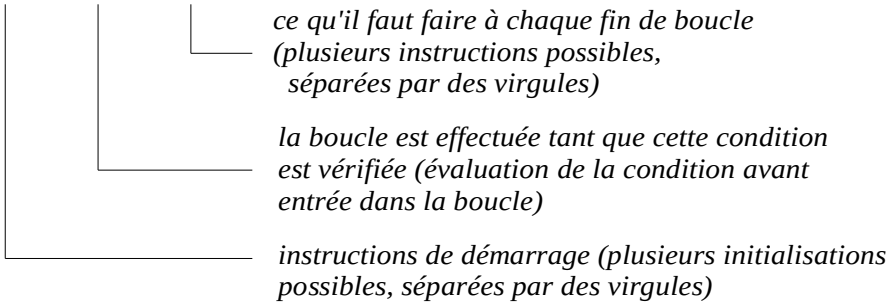



```
do instruction; while (condition);
ou
do { instruction 1;
     . . . .
     instruction N; }
while (condition) ;
```

La différence avec le **while** réside dans le fait que la boucle **do while** est exécutée au moins une fois.

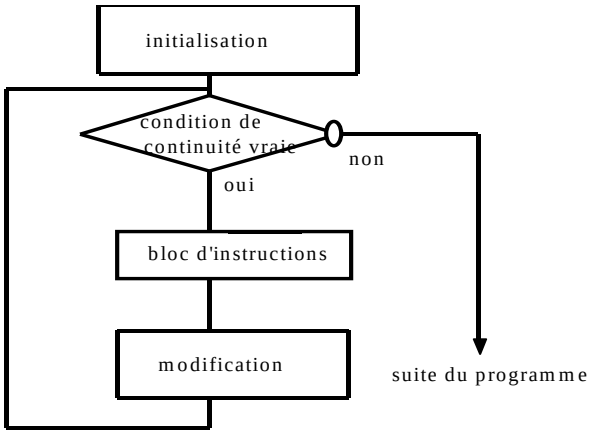
II.3- La boucle pour (for)

Syntaxe : `for (expr 1; expr 2; expr 3) instructions;`



L'instruction **for** s'utilise avec trois expressions, séparées par des points virgules, mais qui peuvent être vides. Si expr2 n'existe pas, elle sera supposée vraie.

Organigramme :



II.4- Ruptures de séquence

Dans le cas où une boucle commande l'exécution d'un bloc d'instructions, il peut être intéressant de vouloir sortir de cette boucle alors que la condition de passage est encore valide. Ce type d'opération est appelé une rupture de séquence. Les ruptures de séquence sont réalisées par quatre instructions qui correspondent à leur niveau de travail : continue, break, exit et return.

- **break** : - interruption de l'exécution d'une boucle **for**, **while** et **do while**.
  - pour un **for**, la sortie s'effectue sans exécuter les instructions de fin de boucle.
  - fait également sortir d'un **switch**.
  - ne fait pas sortir d'une structure conditionnelle **if**

- **continue** : permet de passer à l'itération suivante de la boucle en sautant à la fin du bloc. Elle provoque la non exécution des instructions qui la suivent à l'intérieur du bloc.

- **exit (n)** : termine l'exécution d'un programme et retour au système d'exploitation. La valeur de retour est n.
- **return (n)** : permet de sortie d'un sous-programme (fonction) avec la valeur n

III – LES TABLEAUX STATIQUES

Un tableau est une variable structurée mais qui contient un ensemble d'éléments tous de même type et rangés consécutivement en mémoire.

I - Tableau statique unidimensionnel

I.1 – Déclaration d'un tableau unidimensionnel

On déclare un tableau en indiquant :

- le type des éléments qu'il contient.
- son identificateur.
- le nombre d'éléments qu'il contient entre crochets.

La déclaration se met sous la forme : < type > < identificateur > < taille >

Exemple :

```
int tab [10];   // déclaration d'un tableau de 10 entiers
float x [20];   // déclaration d'un tableau de 20 réels
```

L'expression < taille > qui définit la dimension par défaut doit être une constante entière de manière à être connue au moment de la compilation. Ainsi, le compilateur peut évaluer et connaître la dimension de l'espace nécessaire pour loger les éléments du tableau. Il est impossible donc d'écrire int tab[n]; où "n" serait une variable.

En pratique, il est recommandé d'utiliser un mnémonique pour définir la taille du tableau de manière à faciliter sa gestion.

Exemple :

```
#define dim 10

short tab[dim];      // une allocation mémoire de 10*sizeof(short) = 20 octets est réalisée
float x[dim];        // une allocation mémoire de 10*sizeof(float) = 40 octets est réalisée
```

I.2 – Accès aux éléments du tableau

On accède à chaque élément du tableau par l'intermédiaire d'un indice.

- L'indice peut être valeur entière, une variable ou une expression arithmétique.
- L'indice **0** donne accès au premier élément → tab[0].
- L'indice **dim-1** donne accès au dernier élément → tab[dim-1].

Exemple :

```
int i = 4;
tab[i];      // accès au 5ème élément du tableau
tab[i*2-1];  // accès au 8ème élément du tableau
```

Il n'y a **aucun contrôle de dépassement de bornes** à la compilation et au linkage.

I.3 - Initialisation d'un tableau unidimensionnel

☛ initialisation à la déclaration

On peut initialiser un tableau au moment de sa déclaration en indiquant entre accolades la liste des valeurs de ses éléments. On constitue une liste d'expressions constantes séparées par l'opérateur "virgule".

Exemple :

```
#define dim 4

short tab[dim] = {1, 10, 20, 100};    // affecte à tab[0] la valeur 1, à tab[1] la valeur 10, etc.
```

On peut donner moins d'expressions constantes que le tableau ne comporte d'éléments. Dans ce cas, les premiers éléments du tableau seront initialisés avec les valeurs indiquées, les autres seront initialisés à zéro.

Exemple :

```
#define dim 4

short tab[dim] = {1, 10};      // affecte à tab[0] et tab[1] les valeurs 1 et 10, les autres sont à 0.
```

☛ initialisation par saisie des valeurs au clavier

Il est possible d'initialiser les éléments du tableau en introduisant l'instruction scanf dans une boucle.

Exemple :

```
#define dim 4

int tab[dim];  // déclaration d'un tableau de 4 entiers

for (i = 0;i < dim; i++) scanf("%d", &tab[i]);       // initialisation des 4 éléments du tableau
```

I.4 – Affichage des éléments d'un tableau unidimensionnel

L'affichage des éléments d'un tableau se fait élément par élément (sauf s'il s'agit d'un type char) et nécessite une expression itérative.

Exemple :

```
#define dim 4

int tab[dim];  // déclaration d'un tableau de 4 entiers

for (i = 0;i < dim; i++) printf("%d\n", tab[i]);       // affichage des 4 éléments du tableau
```

Remarque : on peut afficher l'adresse de début de la zone mémoire où se trouvent les éléments du tableau en écrivant : *printf("%x\n", &tab[0]);*

Attention, il est impossible d'effectuer des opérations sur la totalité du tableau car les tableaux ne sont pas des types de base du langage. Les opérateurs ne peuvent s'appliquer qu'élément par élément. En particulier:

tab1 = tab2 ne recopie pas le tableau tab2 dans le tableau tab1.
tab1 == tab2 ne compare pas le tableau tab2 et le tableau tab1.

I.5 – détermination automatique de la taille d'un tableau unidimensionnel

Il est possible de détermination automatique de la taille d'un tableau unidimensionnel en utilisant l'opérateur sizeof(). Ainsi, pour obtenir la dimension du tableau en nombre d'éléments, on peut écrire :

sizeof(tab) / sizeof(tab[0]) ou sizeof(tab) / sizeof(int)

Exemple d'utilisation pour afficher les elements du tableau :

```
int tab[10] = {1, 2, 3}
for (i = 0;i < sizeof tab / sizeof(int); i++)  printf("%d\n ", tab[i]);
```

II - Tableau statique multidimensionnel

Il existe en langage C la possibilité de déclarer et d'utiliser des tableaux à plusieurs dimensions. Il s'agit en fait de tableaux, de tableaux, de tableaux...

☞ La déclaration se met sous la forme : < type > < identificateur > < taille1 > < taille2 > < taille3 > …

☞ On accède à un élément du tableau en donnant un indice par dimension (entre crochets).

Exemple :

```
#define ligne 2                  // matrice de 2 lignes et 3 colonnes
#define colonne 3

int i, j;                        // i est l'indice de ligne et j est l'indice de colonne
int tab[ligne] [colonne];        // déclaration d'un tableau de 6 entiers

for (i = 0; i < ligne; i++)
  for (j = 0; j < colonne; j++) printf("%d\n", tab[i][j]); // affichage des 6 éléments du tableau
```

Remarque : le compilateur va réserver 2*3 cases mémoire pour ce tableau. L'implantation en mémoire se fait de la manière suivante :

t[0][0]
t[0][1]
t[0][2]
t[1][0]
t[1][1]
t[1][2]

Les colonnes sont placées les une à la suite des autres, ce qui peut permettre d'identifier les éléments avec un seul indice si l'on utilise les adresses.

☞ On peut aussi initialiser ces tableaux soit par saisie des valeurs au clavier soit au moment de la déclaration. Par exemple, la déclaration suivante crée et initialise un tableau de 2 lignes et 3 colonnes:

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

L'élément tab[0][0] prend la valeur 1, l'élément tab[0][1] prend la valeur 2, etc...

III – Chaînes de caractères

Les chaînes de caractères sont stockées sous forme de tableaux de caractères. Le compilateur complète la chaîne avec un caractère NULL ('\0') qui est ajouté à la suite du dernier caractère utile constituant la chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

III.1 – Déclaration d'une chaîne de caractères

Une chaîne de caractères peut donc être déclarée de la manière suivante :

char ch[10]; // chaîne de 9 caractères + caractère NULL

III.2 – Initialisation de la chaîne de caractères

☞ initialisation à la déclaration

- Un tableau de caractères peut être initialisé par une liste de constantes caractères :

  char ch[3] = {'A', 'B', 'C'};

  C'est une méthode à proscrire car très lourde.

- Un tableau de caractères peut être initialisé par une chaîne littérale de la manière suivante :

  char ch[8] = "bonjour";
  char ch[20] = "bonjour"; // 7 caractères + '\0' et les autres sont à 0)

☞ initialisation à la saisie au clavier

On peut utiliser la fonction **scanf** et le format %s mais on utilisera de préférence la fonction **gets** non formatée.

```
char texte[10];
printf("Entrer un texte : ");
scanf("%s", texte);     est équivalent à        gets(texte);
```

Une chaîne étant un pointeur, on n'écrit pas le symbole &.

Remarque: **scanf** ne permet pas la saisie d'une chaîne comportant des espaces. Les caractères saisis à partir de l'espace ne sont pas pris en compte (l'espace est un délimiteur au même titre que LF) mais ils sont rangés dans le buffer d'entrée. Pour saisir une chaîne avec des espaces, il faut utiliser l'instruction **gets**. A l'issue de la saisie d'une chaîne de caractères, le caractère de retour chariot est remplacé par le caractère de fin de chaîne '\0'.

III.3 – Affichage d'une chaîne de caractères

On peut utiliser la fonction **printf** et le format %s:

```
char texte[10] = "Bonjour";
printf ("Texte saisi : %s\n",texte);
```

On utilisera si possible la fonction puts non formatée: *puts(texte)* est équivalent à : *printf("%s\n",texte);* La fonction ajoute automatiquement le caractère de saut de ligne **\n** en fin d'affichage.

III.4 - Fonctions permettant la manipulation des chaînes

Une chaîne de caractères est un tableau de caractères. Comme n'importe quel type, les opérateurs = et == ne permettent pas de copier une chaîne dans une autre ni de comparer le contenu de deux chaînes. Par contre, il existe des fonctions de traitement de chaîne de caractères dont les prototypes sont déclarés dans les fichiers **string.h ou stdlib.h.** En voici quelques exemples :

- **int strcmp(const char *chaîne1, const char * chaîne2)** compare deux chaînes de caractères. Elle retourne un nombre entier:
  - *nul si les 2 chaînes contiennent les mêmes caractères y compris '\0'*
  - *négatif si chaîne1 < chaîne2*
  - *positif si chaîne1 > chaîne2*

- **char * strcpy (char *destination, const char *source)** copie de chaînes de caractères. Elle copie le contenu d'une chaîne (source) dans une autre (destination). Le second argument peut être une variable de type chaîne de caractères ou une constante de ce type.

- **unsigned int strlen (const char *)** détermine la taille d'une chaîne de caractères. Elle retourne le nombre de caractères présents dans une chaîne sans le caractère NULL.

- **char * strcat(char *destination, const char *source)** réalise la concaténation de deux chaînes de caractères. Cette fonction recopie la seconde chaîne (source) à la suite de la première (destination) après avoir effacé le caractère '\0' de la chaîne **destination**. Elle renvoie l'adresse de la chaîne destination. Il est donc nécessaire que l'emplacement réservé pour la première chaîne soit suffisant pour y recevoir la partie à lui concaténer. Cette fonction est très utilisée pour les noms de fichier avec des extensions différentes.

- **char *strstr(const char *chaîne1, const char *chaîne2)** entraîne la recherche de la chaîne chaîne2 dans la chaîne chaîne1.

IV – LES POINTEURS

I – Définition et déclaration

Une variable est en fait une petite zone mémoire de x octets en fonction de type qui est allouée et où l'on va stocker les informations. En fait, il est possible d'accéder à une variable en utilisant :

- son nom ou identificateur
- son adresse en utilisant un pointeur

Un pointeur est une variable qui contient l'adresse d'un autre objet (variable ou fonction).

Les pointeurs sont définis par un type et se déclarent de la façon suivante :

*type *nom ou type* nom*

Le type correspond au type des variables pointées et le nom correspond au nom du pointeur.

Exemple : *int *p; // déclaration d'un pointeur p sur un entier quelconque*

II – Utilisation d'un pointeur

La manipulation des pointeurs nécessite l'utilisation des deux opérateurs suivants :

- L'opérateur **&** : cet opérateur, qui doit être obligatoirement suivi du nom d'une variable, indique la valeur de l'adresse d'une variable.

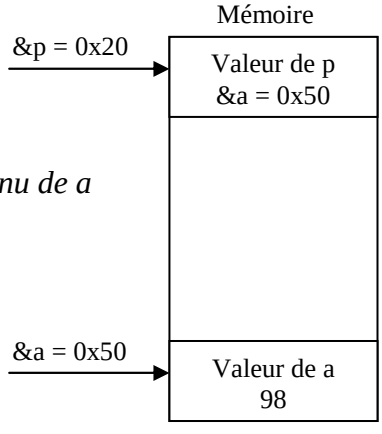
- L'opérateur ***** : cet opérateur désigne le contenu de la case mémoire pointée par le pointeur.

Exemple :

*int a, *p; // déclaration de 2 variables **a** et ***p. p** est un pointeur sur un entier quelconque.*
*a = 98; // initialisation de la variable **a** mémoire de 4 octets allouée par le système.*
*// dont l'adresse est **0x50** par exemple.*

***p = &a;** // initialisation du pointeur **p** : **p** pointe l'entier **a** dont l'adresse est 0x20.*

a désigne le contenu de a
&a désigne l'adresse de a
p désigne le contenu du pointeur c'est-à-dire l'adresse de a
**p désigne la valeur pointée par le pointeur c'est-à-dire le contenu de a*
&p désigne l'adresse du pointeur
**a est illégal (a n'est pas un pointeur)*



Le contenu du pointeur affiché en hexadécimal en utilisant printf avec le format %p ou %x.

☞ **cas particulier : le pointeur nul**

Pour certains besoins, il peut être utile de vouloir comparer un pointeur avec la valeur nulle. On utilisera alors la constante NULL prédéfinie dans <stdio.h>.

Exemple : *int *p;*
if (p == NULL) …

III - Pointeurs et tableaux

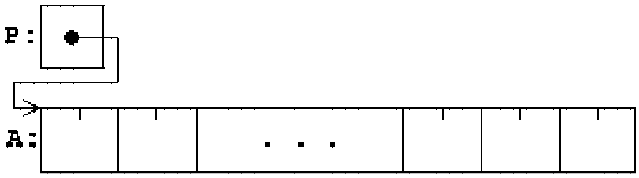
III.1 – Tableau unidimensionnel

Jusqu'à présent, on a utilisé les tableaux de manière intuitive par une déclaration du type int tab[10]. L'opérateur d'indexation noté [ ] permet d'accéder à un élément du tableau via un indice. En fait, le nom d'un tableau est un pointeur constant sur le premier élément de celui-ci. Les pointeurs sont donc particulièrement utilisés pour gérer des tableaux.

adresse d'un tableau = nom du tableau = adresse du 1er élément

Exemple :

int A[10];
*int *p;*
p = A; est équivalente à p = &A[0];

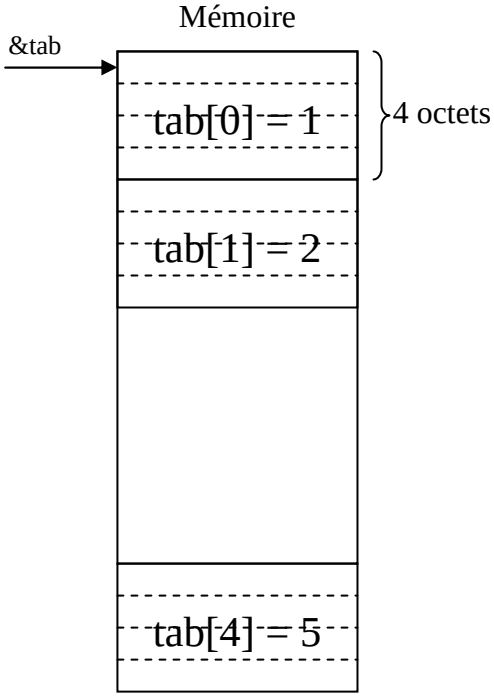


Si **p** pointe sur une composante quelconque d'un tableau, alors **p+1** pointe sur la composante suivante. Plus généralement,

p + i pointe sur la i^{ème} composante derrière p.
p - i pointe sur la i^{ème} composante devant p.

Ainsi, après l'instruction, p = A; le pointeur p pointe sur A[0]

*(p + 1) désigne le contenu de A[1]
*(p + 2) désigne le contenu de A[2]
*(p + i) désigne le contenu de A[i].



Exemple :

main()
*{ int tab[10]={1,2,3,4,5}, *p, x;*

p = &tab[0]; *équivaut à **p = tab** :*
x= *tab; *valeur du 1er élément, x = 1*
x= *(tab + 0); *équivaut à **tab[0]** c'est-à-dire x = 1*
x = *(tab+i); *équivaut à **tab[i]***
tab+i *équivaut à **&tab[i]***
}

En conclusion, pour travailler sur un tableau, on a donc le choix de travailler avec un tableau en utilisant un indice ou de travailler avec un pointeur.

Un tableau peut donc être déclaré de 2 façons :

```
type identificateur [dimension]      // tableau statique
type *identificateur                 // pointeur seul
```

☞ **pointeurs et opérateurs**

On peut donc utiliser les opérateurs arithmétiques +, - pour déplacer un pointeur dans une zone mémoire définie. Le déplacement se fait modulo la taille en octets du type des éléments pointés.

Attention, ne pas confondre p + 1 et p++ ou ++p. Dans le deux derniers cas, le contenu du pointeur est modifié.

☞ **affichage d'un tableau de réels**

Deux méthodes possibles :

- indices entiers

```
#define dim 15
{int i;
 double x[dim];

 for (i = 0; i < dim; i++) printf("lf \n",  x[i]);
 où bien
 for (i = 0; i < dim; i++) printf("lf\n", *(x+i));
}
```

- indices pointeurs

```
#define dim 15
{double x[dim], *p;

 for (p = x; p < x + dim; p++) printf("lf\n", *p);
où bien
 for (p = x; p < x+dim;) printf("lf\n", *p++);
}
```

III.2 – Tableau multidimensionnel - tableau de pointeurs

Un tableau unidimensionnel peut donc se représenter grâce à un pointeur (le nom du tableau) et à l'aide d'un indice de déplacement.

Un tableau à plusieurs dimensions peut se représenter à l'aide d'une notation similaire, construite avec des pointeurs. Par exemple, un tableau de dimension 2, est en fait un ensemble de deux tableaux à une seule dimension. Il est ainsi possible de considérer ce tableau à deux dimensions comme un pointeur vers un groupe de tableaux unidimensionnels consécutifs.

La déclaration correspondante peut s'écrire de la manière suivante :

*type (*p) [expression 2];*

au lieu de la déclaration classique:

type p[expression l] [expression 2];

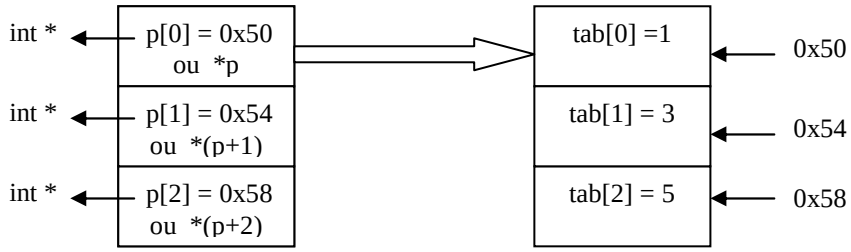
Il s'agit donc de tableaux de pointeurs.

Exemple :

```
int tab[3] = {1, 3, 5};       // déclaration impérative en globale pour le compilateur

main()
{
 int *p[3]  = {tab, tab+1, tab+2};
}
```

Ici, **p** est un tableau de 3 pointeurs pointant des zones mémoires contenant des éléments de type int.



On a :

*p[0][0] = 1 = *(*(p+0)+0) p[1][0] = 3 = *(*(p+1)+0) p[2][0] = 5 = *(*(p+2)+0)*
*p[0][1] = 3 = *(*(p+0)+1) p[1][1] = 5 = *(*(p+1)+1)*
*p[0][2] = 5 = *(*(p+0)+2)*

Les éléments étant places de manière consécutive dans la mémoire, au lieu d'écrire p[i][j], on peut écrire :

**(*(p+i)+j)*

☞ **cas particulier d'un tableau de chaînes de caractèress**

Un tableau de chaîne de caractères peut être défini comme un tableau à 2 dimensions soit :

char tab[10][20]; Cette définition réserve en mémoire 200 octets.

On préfère définir un tableau de pointeurs sur des chaînes :

*char *tab[10];*

Ce tableau de 10 pointeurs sur des chaînes (type char *) et permet à ces dernières d'être de longueur variable et d'occuper en mémoire que la place nécessaire.

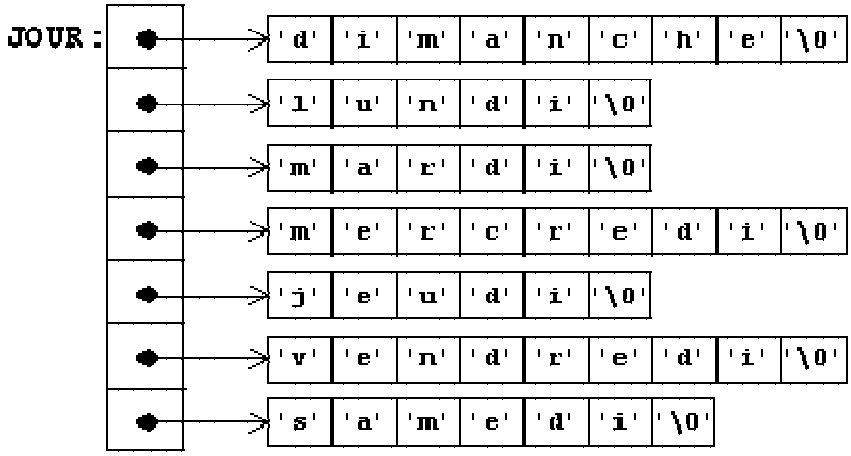
Les différentes chaînes peuvent donc avoir des tailles différentes, ce qui ne serait pas possible si l'on déclarait un tableau de caractères à 2 dimensions comme : *char tab[10][5]*.

Chaque élément du tableau pointe vers une chaîne de caractères. On conçoit qu'avec une telle représentation, une permutation de chaîne est très facile puisqu'il suffit de permuter les pointeurs correspondants.

Exemple :

*char *JOUR[7] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};*

déclare un tableau **JOUR[7] de 7 pointeurs sur char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.



JOUR est équivalent à **&JOUR[0]**. C'est l'adresse du premier élément du tableau qui contient les 7 pointeurs.

***JOUR** est équivalent à **JOUR[0]**. C'est l'adresse du premier caractère de la chaîne "dimanche".

(JOUR +1)** est équivalent à **JOUR [1].

On peut afficher les 7 chaînes de caractères en fournissant les adresses contenues dans le tableau JOUR à printf (ou puts) :

int i;
for (i = 0; i < 7; i++) printf ("%s\n", JOUR[i]); **Affiche les jours de la semaine.**

Comme JOUR[i] est un pointeur sur char, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur * de la manière suivante :

*** JOUR [0]** est le caractère pointé par **JOUR [0].** C'est le caractère "'d" de la chaîne "dimanche".

int i;
*for (i = 0; i < 7; i++) printf("%c\n", *JOUR[i]);* **Affiche la 1ère lettre de chaque jour (dlmmjvs).**

**** JOUR** est équivalent à *** JOUR[0].** C'est le caractère 'd' de la chaîne "dimanche".

****(JOUR +1)** est équivalent à ***JOUR [1].** C'est le caractère 'l' de la chaîne "lundi".

L'expression **JOUR[i]+j** désigne la **j-ième** lettre de la **i-ième chaîne**. On peut afficher la troisième lettre de chaque jour de la semaine par:

int i;
*for (i = 0; i < 7; i++) printf("%c\n", *(JOUR[i]+2));* Affiche (**mnrrunm**).

III.3 – Pointeurs de tableau

Un pointeur peut contenir l'adresse d'une variable assez complexe comme un tableau dans sa globalité. Il est ainsi possible de définir des pointeurs sur des tableaux à une ou plusieurs dimensions. Dans ce cas, la déclaration s'écrit de la manière suivante :

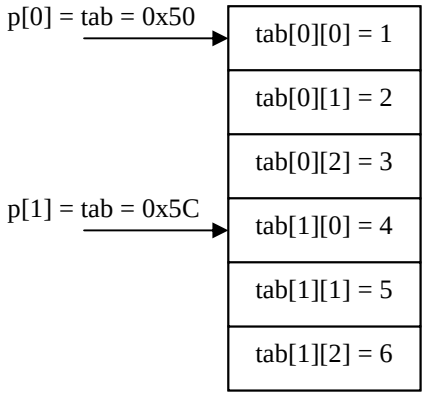
*type (*p) [taille];*

Exemple :

```
int tab[2][3]={{1,2,3},{4,5,6}} ;  // déclaration impérative en globale pour le compilateur

main()                                   Nombre d'éléments
{
 int (*p[2]) [3]   = {tab, tab+1} ;
}
```

p[0] = 0x50 → *(p[0]) [0] = 1
→ *(p[0]) [1] = 2
→ *(p[0]) [2] = 3
p[1] = 0x5C → *(p[1]) [0] = 4
p[0] = 0x50 → *(p[1]) [1] = 5
p[0] = 0x50 → *(p[1]) [2] = 6



III.4 – Pointeurs de fonction

Voir chapîte VI sur les fonctions paragraphe VIII.

V – ALLOCATION DYNAMIQUE

La déclaration de variables dans le programme principal ou en global réserve de l'espace mémoire pour ces variables et pour toute la durée de vie du programme. Elle impose par ailleurs de connaître, avant le début de l'exécution, l'espace nécessaire au stockage de ces variables et en particuliers la dimension des tableaux.

Or, dans de nombreuses applications, le nombre d'éléments d'un tableau peut varier d'une exécution du programme à l'autre. Allouer de la mémoire à l'exécution revient à réserver de manière dynamique la mémoire.

Principal intérêt de la déclaration dynamique : allouer ou libérer la mémoire en fonction des besoins et en cours d'exécution d'un programme.

☞ allocation d'un élément de taille définie – fonction malloc

void *malloc(unsigned size) : cette fonction alloue dynamiquement en mémoire, un espace de **size** octets et renvoie l'adresse du 1er octet de cet espace mémoire ou renvoie NULL si l'allocation n'a pu être réalisée.

- Pas d'initialisation à 0 de la variable.
- Mettre un "cast" si on veut forcer malloc à renvoyer un pointeur du type désiré.
- Utiliser la fonction sizeof(type) qui renvoie la taille en octets du type passé en paramètres.

Exemple :

```
int *p;                          // Déclaration du pointeur p
p = (int*) malloc (sizeof(int)); // Allocation dynamique de 4 octets en mémoire.
                                 // Initialisation du pointeur p
```

☞ allocation de plusieurs éléments consécutifs – fonction calloc

void *calloc (unsigned nb, unsigned size) réalise l'allocation dynamique de **nb** éléments de taille **size** octets et retourne l'adresse du 1er octet alloué.

- Initialise la mémoire à 0.
- Retourne un pointeur NULL si l'allocation n'a pas eu lieu.
- Mettre un "cast" si on veut forcer calloc à renvoyer un pointeur du type désiré.

Exemple :

```
int *p;
p = (int*) calloc (10, sizeof(int));   // Allocation dynamique de 40 octets en mémoire.
```

Remarque : on peut écrire : p = calloc(10, sizeof(int)) ou p = malloc(10*sizeof(int))

Exemple d'allocation dynamique pour un tableau bidimensionnel :

```
int **p;
p = (int**) calloc (3, sizeof(int*));
*(p+i) =(int*) calloc (2, sizeof(int));  // Allocation dynamique de 24 octets en mémoire.
                                         // tableau dynamique de 3 lignes et 2 colonnes
```

☞ **changement de taille d'un tableau alloué dynamiquement – fonction realloc**

void *realloc (void *p, unsigned size); cette fonction permet de modifier la taille d'une zone mémoire précédemment allouée dynamiquement. p est le pointeur sur le début du bloc mémoire concerné et size correspond à la nouvelle taille.

- realloc équivaut à malloc.
- Si l'allocation est réussie, la fonction renvoie un nouveau pointeur sur le début du nouveau bloc alloué. Sinon, elle retourne NULL.

☞ **libération de l'espace mémoire – fonction free**

L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin. La fonction **free()** permet de libérer la mémoire précédemment allouée dynamiquement et désignée par un pointeur **p**. Sa syntaxe est :

*void free (void *p);*

- Aucune valeur n'est retournée.
- n'a pas d'effet si le pointeur a la valeur NULL.

VI – LES FONCTIONS

I – Introduction

L'élément de base de la programmation modulaire en C est la notion de fonction. Les fonctions permettent de décomposer un programme en entités plus limitées et donc d'en simplifier à la fois la réalisation et la mise au point.

Une fonction peut :

- Se trouver dans le même fichier que le programme principal ou dans un autre.
- Etre appelée à partir du main(), d'une autre fonction ou d'elle-même (problème de récursivité).
- Admettre ou non des arguments en entrée.
- Retourner ou non une valeur spécifique.
- Posséder ses propres variables.

Une fonction est constituée d'un entête et d'un corps. Il faut distinguer la déclaration et la définition d'une fonction (bien que les deux puissent être effectuées en même temps).

II – Déclaration d'une fonction - Prototypes

La déclaration d'une fonction consiste à indiquer:

- Son identificateur c'est-à-dire son nom.
- Le type de la valeur retournée (**void** si la fonction ne retourne pas de valeur)
- Une liste de paramètres entre parenthèses séparés par des virgules avec pour chacun d'entre eux son type (cette liste peut être vide).

En norme ANSI, une déclaration de fonction s'écrit :

type identificateur (déclaration des paramètres d'entrée);

Comme toute déclaration de variables, la déclaration d'une fonction est suivie d'un point-virgule (;).

Exemples :

- *void fonc1 (int, float);* la fonction **fonc1** admet 2 arguments en entrée (le 1er de type int et le 2ème de type float) et ne retourne aucune valeur spécifique (void).
- *float fonc2 (int, int);* la fonction **fonc2** admet 2 arguments de type int et retourne un flottant.
- *char *fonc3 (void);* la fonction **fonc3** n'admet aucun argument en entrée (void) et retourne l'adresse d'un caractère ou d'une chaîne de caractères.

Ces déclarations définissent donc les prototypes des fonctions énumérées. Un prototype donne la règle d'usage de la fonction : nombre et type d'arguments d'entrée ainsi que le type de la valeur retournée pour l'utilisateur.

D'autre part, le prototype est indispensable pour le compilateur car il réalise une vérification syntaxique du code source de la 1ère à la dernière ligne du programme. Le prototype doit être placé en début de programme et la portée du prototype s'étend à tout le programme.

III – Définition d'une fonction

La définition d'une fonction est composée :

- D'une entête semblable et compatible avec son prototype.
- D'un bloc délimité par les caractères { et } contenant des déclarations de variables et des instructions qui la composent. Les variables sont dites locales à la fonction.
- D'une instruction **return** qui permet la sortie immédiate de la fonction et de transmettre la valeur de retour (s'il y en a une).

Attention, on ne peut pas définir une fonction à l'intérieur d'une autre fonction (toutes les fonctions sont "au même niveau").

IV – Appel d'une fonction

L'appel d'une fonction se fait simplement par son identificateur suivi de la liste des paramètres requis (entre parenthèses).

Exemple :

```
#include <utility.h>
#include <ansi_c.h>

int plus2 (int);

int j;

main ()
{
  int i = 2;

  j = plus2 (i);

  printf("valeur de j = %d\n",j);

}

int plus2 (int k)
{
  k+=2;
  return (k);
}
```

Prototype de fonction

Variable globale

Début de bloc

Déclaration et initialisation d'une variable locale au main

Corps du programme principal

Appel à une fonction

Fin de bloc

Argument d'entrée, "k" est initialisé à la valeur de "i"

Corps de la fonction

Renvoi de "k" au programme principal

Le programme principal appelle la fonction plus2. La variable "i" a été transmise à la fonction par l'intermédiaire de l'argument d'entrée "k". La fonction a ajouté 2 à la variable locale "k" puis et retournée au programme principal.. Cette valeur est "récupérée" par la variable "j".

On dit que les **arguments sont transmis par valeur**. Les valeurs de ces paramètres ne sont pas modifiées par la fonction. Les arguments, qui sont considérés comme des variables locales de la fonction, sont automatiquement initialisés aux valeurs passées en argument. Seul, l'ordre des arguments intervient, le nom des arguments ne jouant aucun rôle. Ici, "k" a pris automatiquement la valeur de "i".

V – Passage des paramètres

On vient de voir que les paramètres passés à une fonction le sont par valeur et ne peuvent pas être modifiés par l'exécution de la fonction. Ceci est très contraignant si l'on souhaite qu'une fonction renvoie plusieurs résultats par exemple.

Comment, par une fonction, peut-on modifier le contenu d'une variable déclarée dans le main() si la fonction ne retourne aucune valeur spécifique ? Pour répondre, on va s'appuyer sur le programme précédent en le modifiant de la manière suivante :

```
void plus2_val (int );          // prototypes de 2 fonctions ne retournant aucune valeur spécifique
void plus2_adr (int * );

main()
{
 int i=2;
 plus2_val(i);                  // appel de la fonction plus2_val
 printf ("après appel de la fonction plus2_val : %d\n", i);

 plus2_adr(&i);                 // appel de la fonction plus2_adr
 printf ("après appel de la fonction plus2_adr : %d\n", i);
}

int plus2_val(int nb)           // un argument d'entrée de type entier
{
 nb=nb+2;
}

int plus2_adr(int *nb)          // un argument d'entrée de type pointeur sur int
{
 *nb=(*nb)+2;
}
```

Dans les deux cas, ces fonctions sont destinées à ajouter 2 à la variable "i".

Résultat :

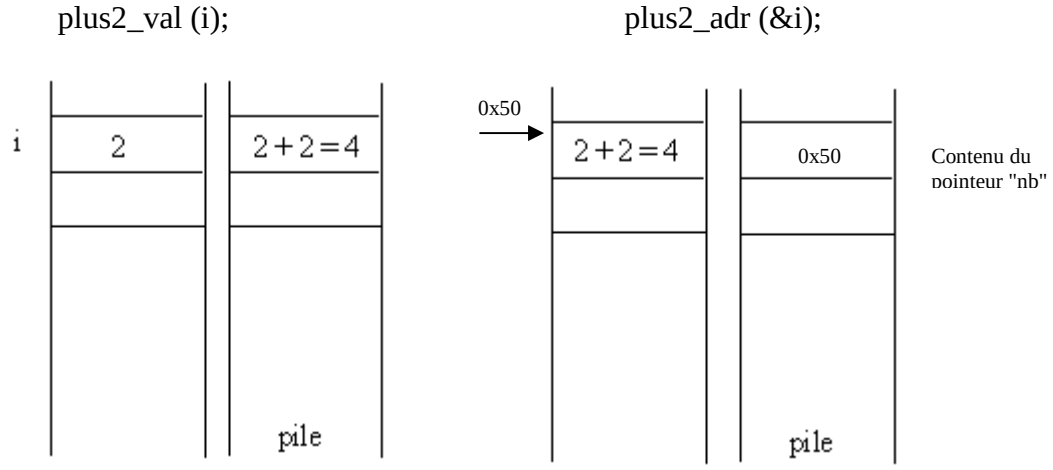
- après exécution de la fonction plus2_val, on a : i = 2
- après exécution de la fonction plus2_adr, on a : i = 4

Analyse :

La pile est une zone mémoire dans laquelle sont créées toutes les variables locales des fonctions. Ces variables locales apparaissent lors de l'entrée dans les fonctions. La dernière variable introduite dans la pile sera la première sortie.

Les variables locales sont donc créées lors de l'entrée dans la fonction. Elles sont détruites lors de la sortie de la fonction (sur instruction return ou après exécution de la dernière instruction de la fonction).

Etudions l'évolution de la pile avec l'exemple précédent :



Dans le premier cas, l'argument d'entrée **nb** a été automatiquement créé et il est placé dans la pile avec la valeur de i c'est à dire 2. La fonction **plus2_val** ajoute 2 à la valeur qui est dans la pile soit 4. Lorsque l'on sort de la fonction, **nb** est effacé puisque **nb** est une variable locale.

Dans le second cas, l'argument passé est l'adresse de **i** soit 0x50. **nb** est un pointeur qui contient l'adresse de **i**. La fonction **plus2_adr** ajoute 2 à la valeur pointée par le pointeur (***nb = *nb + 2)** c'est-à-dire le contenu à l'adresse 0x50. **i** prend donc la valeur 4

Pour modifier le contenu de variables déclarées dans le main par une fonction qui n'a pas d'arguments de retour, il est indispensable de passer les adresses de ces variables. Les arguments d'entrée de fonction devront donc être des pointeurs.

☞ passage d'un tableau en argument

Si l'on veut passer dans une fonction les éléments d'un tableau, il est inimaginable de passer ces éléments un à un. Recopier tout un tableau aurait un effet désastreux sur les performances et provoquerait peut-être une saturation de la pile.

En fait, puisque l'identificateur d'un tableau correspond à l'adresse du 1er élément du tableau, il suffit de récupérer cette adresse au moyen d'un pointeur local à la fonction qui sera initialisé à l'adresse de la zone mémoire concernée.

Exemple :

```
void affiche (int *);    // exemples de prototypes
void affiche (int []);
```

```
main()
{ int tab[10];

 affiche (tab);          // appel de la fonction
}
```

VI – Portée des variables

☞ variable locale

En général, les déclarations de variables sont faites dans le corps des fonctions (y compris la fonction main). On parle de variables locales ou automatiques. Elles sont créées au début de l'exécution de la fonction et détruite quand celle-ci se termine.

Une fonction ne peut donc avoir accès à des variables locales déclarées dans d'autres fonctions que par passage de paramètres. Ces paramètres contiennent une copie des valeurs passées en argument d'entrée au moment de l'appel de la fonction.

Ces arguments d'entrée sont :

- Automatiquement crées à l'appel de la fonction
- Automatiquement initialisés
- Automatiquement détruits à la fin de l'exécution de la fonction

Remarques :

- une variable peut être déclarée à l'intérieur d'un bloc d'instructions (entre {}). Dans ce cas, elle est locale au bloc d'instructions correspondant et n'est donc uniquement visible à l'intérieur de ce bloc.
- Une variable locale n'est pas initialisée sauf si elle est argument d'entrée d'une fonction.

☞ variable globale

Il est possible de déclarer des variables ayant une portée plus importante : ce sont les variables globales. Ces variables doivent être déclarées en début de fichier et sont visibles par toutes les fonctions définies dans le fichier.

On peut aussi accéder à des variables globales définies dans d'autres fichiers à condition de les redéclarer avec le mot clé **extern**.

Remarque : les variables globales sont, en général, déclarées immédiatemment après les #include et **initialisées à 0 au début de l'exécution du programme, sauf** si on les initialise à une autre valeur.

☞ variable static

On peut allonger la durée de vie d'une variable locale en la déclarant **static.** Lors d'un nouvel appel à la fonction, la variable garde la valeur obtenue à la fin de l'exécution précédente. Une variable **static** est initialisée à 0 lors du premier appel à la fonction.

On peut ainsi faire un compteur pour connaître le nombre de fois où la fonction a été appelée.

Exemple :

```
#include <utility.h>
#include <ansi_c.h>

void func(void);                // prototype

int count = 10;                 // variable globale

main()
  {
  Cls();
  while (count--) func();       // appel de la fonction 10 fois
  }

//*****************************************************************************
void func(void)
  {
  static int compteur=0;        // La varaiable 'compteur' est locale à la fonction func
  compteur++;
  printf(" compteur is %d and count is %d\n", compteur, count);
  }
```

VII – Bibliothèque de fonctions

Pour gérer efficacement des bibliothèques de fonctions, il est conseillé de maintenir des fichiers pas trop volumineux où les fonctions sont regroupées par thème. Pour construire une bibliothèque, il faut créer deux fichiers:

- Un fichier d'entête (avec l'extension ".h") qui contient les prototypes des fonctions. Il suffira ensuite de faire un #include du fichier ".h" dans le programme principal.
- Un fichier source (".c") qui contiendra les définitions de ces fonctions.
- Un fichier project (avec l'extension ".prj") indiquant au linkeur quels sont les fichiers à prendre en compte pour réaliser l'exécutable.

VIII – Pointeurs de fonctions

Les pointeurs de fonction sont des pointeurs qui, au lieu de pointer vers des données, pointent vers du code exécutable. La déclaration d'un pointeur de fonction ressemble à celle d'une fonction sauf que l'identificateur de la fonction est remplacé par l'identificateur du pointeur précédé d'un astérisque (*) le tout mis entre parenthèses.

```
int (* fpointer) (int, int);
```

déclare un pointeur vers une fonction retournant un entier et nécessitant deux paramètres de type entier. L'intérêt est, par exemple, de pouvoir passer en paramètre d'une fonction, une autre fonction.

Exemple :

```
#include <utility.h>
#include <ansi_c.h>

int (*fpointer) (int, int);

int add(int, int);
int sub(int, int);

main()
  {
  Cls();
  fpointer = add;                    // initialise le pointeur 'fpointer'
                                     // à l'addresse de 'add'

  printf("%d \n", fpointer(4, 5));   // Exécute 'add' et affichage

  fpointer = sub;                    // idem pour 'sub'
  printf("%d \n", fpointer(6, 2));
  }

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

int add(int a, int b)
  {
  return(a + b);
  }

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

int sub(int a, int b)
  {
  return(a - b);
  }
```

Il y a sélection de l'opération à réaliser par initialisation du pointeur à l'adresse de la fonction à exécuter.

VII – LES STRUCTURES ET LES UNIONS

I – Introduction

Une structure est un ensemble d'éléments de types différents réunis sous un même nom ce qui permet de les manipuler facilement. Elle permet de simplifier l'écriture d'un programme en regroupant des données liées entre elles.

Un exemple d'utilisation d'une structure est la représentation d'un nombre complexe qui est défini par deux éléments : la partie réelle et la partie imaginaire. Le regroupement de ces deux informations par un seul nom de variable structurée permet de faciliter la manipulation de ces nombres complexes.

II – Déclaration

On définit une structure à l'aide du mot réservé **struct** suivi d'un identificateur (nom de la structure) et de la liste des variables qu'elle doit contenir (type et identificateur de chaque variable). Cette liste est délimitée par des accolades. Chaque variable contenue dans la structure est appelée **champ**.

Exemple de déclaration :

```
struct complexe
{
double reel;
double imag;
};

struct complexe x, y;
```

Ceci définit un modèle de structure dont le nom est **complexe** et précise le nom et le type de chacun des membres (ou **champs**) constituant la structure (reel et imag de type doucle). **x** et **y** sont deux variables de type **struct complexe**.

Définir une structure consiste en fait à définir un nouveau type de variable, un type personnalisé.

Chaque champ d'une structure peut être d'un type absolument quelconque : pointeur, structure, tableau .... Il peut donc y avoir des imbrications de structures.

Autre exemple :

Un exemple type d'utilisation d'une structure est la gestion d'un répertoire téléphonique. Chaque fiche d'un répertoire contient, par exemple, le nom d'une personne, son prénom et son numéro de téléphone. Le regroupement de toutes ces informations dans une structure permet de manipuler facilement ces fiches.

```
struct personne
{
 char nom [20];        // le 1er et le 2ème champs sont des chaînes de caractères
 char prenom[20];
 unsigned long numero;
};
```

III - Utilisation de typedef

Pour simplifier la déclaration de types, on peut utiliser la déclaration **typedef** qui permet de définir ce que l'on nomme en langage C des "types synonymes", connus par la suite dans tout le programme.

Cette définition de type consiste à faire suivre le mot réservé **typedef** par une construction ayant la même syntaxe qu'une définition de variable. Ces nouveaux types sont ensuite utilisables comme un type de base.

A priori, elle s'applique à tous les types et pas seulement aux structures.

Exemples d'utilisation :

```
typedef          int entier;
entier n,p;      équivalent à  int n,p;  // redéfinition du type int, entier est un synonyme de int.

typedef   int *ptr;
ptr  p1,p2;      équivalent à  int *p1, *p2;
```

La directive typedef est très souvent utilisée avec les définitions de structures.

Exemple :

```
typedef struct
        {
         double reel;
         double imag;
        } complexe;

complexe x, y;
```

Maintenant, par rapport à la déclaration du paragraphe II, les variables x et y sont de type complexe et non de type struct complexe.

IV - Utilisation d'une structure

On peut utiliser une structure de 2 manières différentes :

- En travaillant individuellement sur chacun des champs.
- En travaillant de manière globale sur l'ensemble de la structure.

☞ utilisation globale d'une structure

Les compilateurs C considère une variable construite à partir d'un type structuré comme une variable simple par conséquent :

- Si les structures **x** et **y** ont été déclarées suivant le modèle **complexe** défini précédemment, on peut écrire **x = y**.

- Il est possible de construire une fonction qui accepte en argument ou qui retourne une structure.

- Il est possible de retourner une structure par une fonction.

☞ **utilisation des champs d'une structure**

Il est possible d'avoir accès à un champ d'une variable structurée de manière individuelle en utilisant l'opérateur ".". Cet accès aux champs se fait en utilisant le nom de la variable structurée suivi de l'opérateur "." et du nom du champ désiré :

nom_variable_structurée.nom_champ

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant.

Exemple :

```
x.reel = 15;            // affecte la valeur 15 au champ reel de la variable structurée x
printf("%ld", x.reel);  // affiche le champ reel de la variable x
```

V – Initialisation d'une structure

On retrouve pour les structures les règles d'initialisation qui sont en vigueur pour tous les types de variables, à savoir :

- En l'absence d'initialisation explicite, les structures "statiques" sont par défaut initialisées à zéro, les automatiques sont indéfinies.

- Initialisation possible lors de sa déclaration mais on ne peut utiliser que des constantes ou des expressions constantes.
  struct complexe x = {5, 6}; ou *struct personne pers = {"dupont","jean",0231747576};*

- Initialisation des champs par utilisation des fonctions scanf, getchar, gets.

```
scanf("%ld", &x.imag);      // saisit une valeur qui sera affectée au champ imag de la variable y
```

VI – Transmission d'une structure en argument de fonction

La norme ANSI a introduit la possibilité de transmettre une structure en argument d'une fonction, elle-même pouvant retourner un tel objet (ne pas passer les champs un à un).

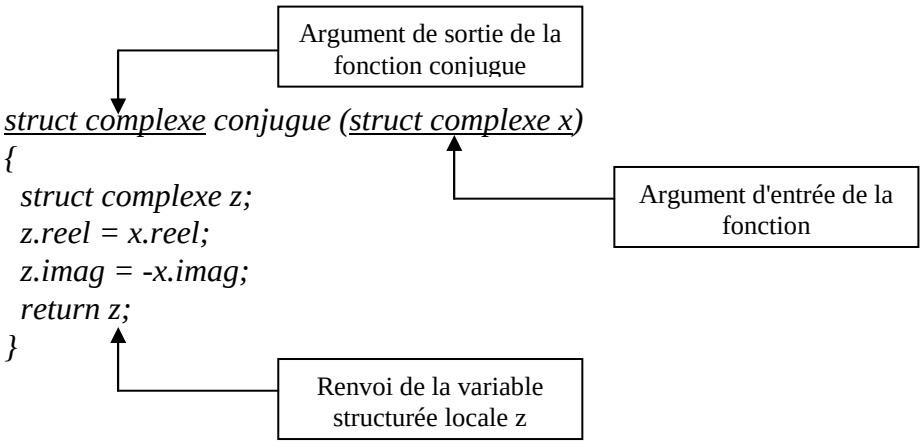
Exemple : le programme suivant utilise une fonction qui renvoie le complexe conjugué de la variable structurée "y1". Le résultat est "récupéré" par la variable structurée "y2".

```
struct complexe conjugue (struct complexe);        // prototype


main()
{
 struct complexe y1 = {2,3}, y2;      // déclaration de 2 variables structurées

 y2 = conjugue(y1);           // appel à la fonction : passage de y1 à la fonction -> résultat dans y2

 printf("Partie réelle = %lf      Partie imaginaire = %lf\n",y2.reel, y2.imag);
}
```



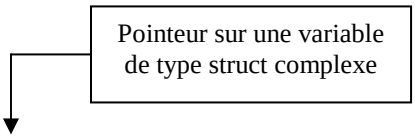

```
struct complexe conjugue (struct complexe x)
{
  struct complexe z;
  z.reel = x.reel;
  z.imag = -x.imag;
  return z;
}
```

VII – Pointeur de structure

Dans le cas où l'on souhaite modifier le champ d'une variable structurée déclarée dans le programme principal par l'intermédiaire d'une fonction, il est nécessaire de transmettre à la fonction l'adresse de la variable structurée que l'on récupère par un pointeur de même type. La déclaration ***struct complexe ** définit un pointeur qui pointera un espace mémoire contenant un objet de type struct complexe.

Pour accéder aux champs d'une structure à partir d'un **pointeur** associé à une variable structurée, il faut utiliser l'opérateur "->" : **nom_de_variable -> nom_du_cham**p.

Exemple : à partir de l'exemple précédent, on souhaite modifier le champ "**imag**" de la variable "y1" par l'intermédiaire d'une fonction ne fournissant aucune valeur spécifique (void). Le programme précédent devient :



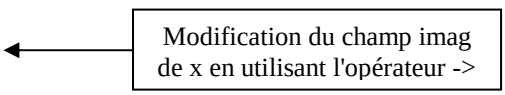

```
struct complexe
{
double reel;
double imag;
};

void conjugue (struct complexe *);   // prototype de la fonction conjugue

main()
{
  struct complexe y1 = {2,3};

  conjugue(&y1);        // passage de y1 à la fonction -> résultat dans y2
  printf("Partie réelle = %lf      Partie imaginaire = %lf\n",y1.reel, y1.imag);
}

void conjugue (struct complexe *x)
{
  x->imag = -x->imag;
}
```



Le mécanisme de passage des variables structurées dans une fonction est identique à celui d'une simple variable. Toute variable locale à une fonction est créée dans la pile.

Pour que la modification du champ "imag" de "y1" puisse se faire, il faut que la fonction **conjugue** reçoive l'adresse d'une structure et non plus sa valeur d'où la déclaration **conjugue(&y1)**. Cela signifie que son prototype est de la forme **void conjugue (struct complexe *)**.

L'opérateur "**->**" permet d'accéder à l'espace mémoire contenant le champ à modifier.

VIII – Tableaux de structures

Un modèle structure peut aussi apparaître comme type des éléments d'un tableau.

Exemple : soit la déclaration suivante :

```
struct point {
        char nom;
        int x;
        int y;
};

struct point courbe [50];
```

La structure **point** permet de représenter un point dans un plan, point qui est défini par son nom (caractère) et ses coordonnées (abscisse et ordonnée). **point** est un nom de modèle de struture et **courbe** représente un objet de type "tableau de 50 éléments du type point".

La déclaration *struct point courbe [50];* permet de définir une courbe de 50 points du type ainsi défini.

La notation **courbe[i].x** désigne la valeur du champ x de l'élément de rang i du tableau courbe.

Exemple d'initialisation de la variable courbe : *struct point courbe[50] = { {'A', 10, 25}, {'M', 12, 28} }*

IX – Portée d'une structure

La portée d'une structure peut être locale (à une fonction) ou globale (accessible par toutes les fonctions d'un même fichier source) selon l'endroit de sa déclaration. Elle ne peut pas en revanche être déclarée **extern** (accessible par d'autre fichier .c).

La déclaration **extern** s'applique à des noms de variables susceptibles d'être remplacés par une adresse au niveau de l'édition de liens. Un modèle de structure n'a de sens qu'au moment de la compilation du fichier source où il se trouve.

On peut toutefois placer les déclarations de modèles dans un fichier que l'on incorpore par des #include.

X – Les unions

Une déclaration **d'union** se présente comme une déclaration de structure. Mais, alors qu'un objet structuré est composé de la totalité de ses champs, un objet union est constitué d'un seul champ choisi parmi tous les champs définis.

Exemple de déclaration :

```
union
{
  float continu;
  int  discret;
} x;
```

```
typedef union
        {
          float continu;
          int   discret;
} essai;

essai x;
```

- A un instant donné, la variable x contiendra un entier ou un flottant, mais pas les deux ! C'est au programmeur de connaître à tout instant le type de l'information contenue dans x.
- L'accès à cette information se fait de façon analogue à l'accès à un champ de structure.
- Le programmeur utilisera x.continu s'il désire manipuler une valeur de type float, sinon il utilisera x.int pour manipuler une valeur de type int.
- Un objet de type union occupe une place égale à celle du plus grand de son champ.
- Tout comme les autres types composés rencontrés jusqu'ici, les unions peuvent être initialisées à la déclaration par exemple : *essai x = { 51.2 };*
- Comme pour une structure, on peut utiliser de manière globale par exemple :

```
union x1, x2;  // deux variables de type union essai
x1 = x2
```

VIII – LES FICHIERS

I – Introduction

Un fichier est un ensemble d'informations stockées sur une mémoire de masse (disque dur, disquette, bande magnétique, CD-ROM).

L'accès à des fichiers, pour lire ou modifier des fiches, peut se faire grâce à des fonctions standard que l'on rencontre avec tous les compilateurs. Ces fonctions se situent à différents niveaux :

- **au niveau bas** : les fonctions d'accès sont directement calquées sur les possibilités du système d'exploitation. Elles permettent un accès direct aux informations car elles ne sont pas bufferisées. Elles manipulent les informations sous forme binaire sans possibilités de formatage et le fichier est identifié par un numéro (de type entier) fixé lors de l'ouverture du fichier.

- **au niveau haut** où la notion de mémoire tampon est introduite (buffer). Les fonctions de niveau haut sont basées sur les fonctions de niveau bas mais elles effectuent des E/S bufferisées, permettent une manipulation binaire ou formatée des informations. Le fichier est identifié par un flux (FILE *) qui contient des informations élaborées relatives au fichier c'est-à-dire adresse du buffer, pointeur sur le buffer, etc.

On ne s'intéresse ici qu'au fichier de niveau haut. Au niveau haut, les caractères ne sont donc pas transmis directement dans le fichier, ils sont stockés temporairement dans un tampon et ce n'est que lorsqu'il est rempli que la transmission a lieu.

II – Fichier texte – fichier binaire

Quel que soit le système d'exploitation, les fichiers sont vus par le programmeur comme des flux de données c'est-à-dire des suites d'octets qui représentent les données. Il existe deux façons de coder les informations stockées dans un fichier :

- **En binaire** : si le fichier contient des données enregistrées sous une forme qui est la copie exacte de leur codage dans la mémoire de l'ordinateur, alors le fichier est dit **binaire**. Les opérations de lecture ou d'écriture sur de tels fichiers sont très rapides car elles ne requièrent pas un travail d'analyse ou de synthèse de l'expression écrite des données. En revanche, les fichiers binaires ne sont pas éditables ou imprimables.

- **En ASCII** : si le fichier contient des informations codées en ASCII,, alors le fichier est dit **texte**. Sa principale qualité est d'être exploitable par un logiciel, un système ou un équipement différents de celui qui les a produits. En particulier, ils peuvent être édités, imprimés. Le dernier octet de ces fichiers est EOF (caractère ASCII spécifique).

III – Déclaration d'un fichier

Au niveau haut, les fichiers sont désignés par un pointeur sur une structure baptisée **FILE** (majuscule obligatoire). Dans cette structure **FILE**, on trouve des informations telles que l'adresse du tampon, la position actuelle dans le tampon, le nombre de caractères qui y ont déjà été écrits ou qui restent à lire.

La déclaration s'effectue par : **FILE *fp**. **fp** est ici l'identificateur du pointeur sur la structure FILE.

IV – Ouverture d'un fichier

La fonction **fopen** ouvre un fichier, spécifié par un nom, soit en lecture, soit en écriture et de n'importe quel type. La structure de cette fonction s'écrit :

*File *fopen(const char *, const char *)*

La fonction **fopen** renvoie :

- un pointeur sur une structure de type **File** si l'ouverture est un succès. Ce pointeur est appelé "pointeur sur fichier" et il est utilisé dans la liste des paramètres des fonctions d'entrée/sortie standard ("stdio.h"). Ce pointeur sera utilisé par les opérations de manipulation du fichier ouvert (lecture, écriture ou déplacement).

- un pointeur **NULL** si le fichier n'a pas pu être ouvert (problèmes d'existence du fichier ou de droits d'accès) et un code d'erreur est stocké dans la variable entière globale **erno**.

Le 1er argument d'entrée de cette fonction **fopen** contient le nom du fichier sous forme d'une chaîne de caractères (il est possible de spécifier un chemin d'accès) et le 2ème, qui est aussi une chaîne de caractères, spécifie le mode d'ouverture. Les principaux modes disponibles sont :

- "**r**" : ouverture du fichier en mode lecture seulement.
- "**w**" : ouverture du fichier en mode écriture seulement. Si le fichier existe déjà, sa longueur est ramenée à 0 c'est à dire que toutes les données précédentes du fichier sont écrasées. Par contre s'il n'existe pas, il sera crée.
- "**a**" : ouverture du fichier en mode ajout seulement. Si le fichier existe déjà, les données supplémentaires seront ajoutées à la fin du fichier sans écraser les données précédentes. Si le fichier n'existe pas, la fonction le crée.

Exemple :
```
FILE *fp;
fp = fopen("c:\toto.txt", "r");
```

V – Fermeture d'un fichier

La fonction **fclose** réalise la fermeture d'un fichier (de n'importe quel type) préalablement ouvert par **fopen**. Elle a le prototype suivant :

*int fclose(FILE *)*

Cette fonction renvoie la valeur **0** si l'opération est un succès et dans le cas contraire renvoie la valeur **EOF (-1)**. Dans ce cas, le numéro de l'erreur a été stocké dans la variable **erno**.

VI – Gestion des erreurs

Les erreurs des fonctions d'entrées-sorties peuvent être récupérées par le programme. Des variables sont associées aux erreurs sur chaque flux d'entrée-sortie. Ces variables ne sont pas directement modifiables mais elles sont accessibles à travers un ensemble de fonctions qui permettent de les tester ou de les remettre à zéro.

De plus, pour donner plus d'informations sur les causes d'erreur, les fonctions d'entrées-sorties utilisent une variable globale de type entier appelé **errno**. Cette variable est aussi utilisée par les fonctions de bibliothèque servant à réaliser les appels système. La valeur de **errno** n'est significative que lorsqu'une opération a échoué et que l'appel de fonction correspondant a retourné une valeur spécifiant l'échec.

Les fonctions associées à la gestion des erreurs sont :

☞ **void clearerr (FILE *fp)** efface les indicateurs d'erreur et de fin de fichier concernant le flux pointé par ***fp**.

☞ **int feof(FILE *fp)** (End Of File en anglais) teste l'indicateur de fin de fichier. Cette fonction retourne 0 si la fin de fichier n'a pas été détectée et une valeur différente de 0 (valeur vraie) si la fin de fichier a été détectée. L'indicateur de fin de fichier ne peut être réinitialisé que par la fonction **clearerr**.

☞ **int ferror(FILE *)** retourne une valeur différente de zéro si une erreur s'est produite c'est-à-dire si la variable qui sert à mémoriser les erreurs sur le fichier ouvert a été affectée lors d'une opération précédente.

VII – Accès en lecture

Le langage C permet plusieurs types d'accès en lecture à un fichier via différentes fonctions. L'appel de ces fonctions provoque le déplacement du pointeur courant relatif au fichier ouvert.

☞ **lecture d'un fichier binaire - fread**

La fonction **unsigned int fread(void *p, unsigned int size, unsigned int n, FILE *fp)** effectue une opération de lecture de **n** objets ayant chacun une longueur de **size octets** sur un fichier binaire ouvert pointé par le pointeur **fp**.

***p** est un pointeur contenant l'adresse de la zone mémoire où l'on veut stocker les données. Il faut que l'espace mémoire pointée par le pointeur soit de taille suffisante pour supporter le transfert des données, c'est-à-dire d'une taille au moins égale à (**size*n**).

fread renvoie la valeur correspondant au nombre d'éléments lus si l'opération est un succès. Ce nombre peut être inférieur à celui initialement demandé. Dans ce cas, deux possibilités : on a atteint la fin de fichier (à tester avec **feof**) ou bien, une erreur est survenue.

☞ **lecture d'un fichier texte - fscanf**

La fonction **fscanf** permet de réaliser le même travail que **scanf** sur des fichiers ouverts en mode texte.

La fonction **int fscanf(FILE *fp, const char * format, void adresse)** lit des caractères depuis un flux et les fait correspondre au format voulu spécifié par le 2ème argument. Le 3ème argument correspond à l'adresse des variables à affecter à partir des données.

Cette fonction retourne le nombre d'objets lus, convertis et mémorisés correctement. la valeur **EOF** (-1) est retournée en cas d'erreur (fin de fichier atteinte avant la première conversion).

☞ autre lecture d'un fichier – fgetc - fgets

int fgetc(FILE *) réalise une lecture d'un caractère (unsigned char) dans le fichier associé. Cette fonction retourne la valeur du caractère lu dans un entier s'il n'y a pas d'erreur et si la fonction échoue (fin de fichier ou erreur), la fonction retourne EOF (-1).

char *fgets(char *, int n, FILE *) retourne l'adresse de la zone mémoire où se trouve la chaîne de caractères lue et de longueur (n-1) spécifié par le 2ème argument dans le fichier associé au 3ème argument.

S'il n'y a pas d'erreur, les caractères de la chaîne sont donc rangés dans la mémoire à partir de l'adresse donnée en 1er argument. Si le nombre de caractères lu est inférieur à la taille, le retour chariot est lu et rangé en mémoire. Il est suivi par un caractère nul '\0' de manière à ce que la ligne une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.

Cette fonction retourne le pointeur NULL si la fin de fichier est rencontrée et aucun caractère n'a été lu ou si une erreur de lecture a lieu (par exemple descripteur de fichier invalide).

VIII – Accès en écriture

Le langage C permet plusieurs types d'accès en écriture à un fichier via différentes fonctions. L'appel de ces fonctions provoque le déplacement du pointeur courant relatif au fichier ouvert.

☞ écriture d'un fichier binaire - fwrite

La fonction **unsigned int fwrite(void *p, unsigned int size, unsigned int n, FILE *fp)** effectue une opération d'écriture de **n** objets ayant chacun une longueur de **size octets** dans le fichier désigné par ***fp**. Les données à écrire sont présentes dans une zone mémoire repérée par ***p**. **fwrite** retourne le nombre de blocs écrits.

La valeur de retour indique le nombre d'objets réellement écrits. Si tous s'est bien passé, cette valeur est égale au paramètre **n**. Si le nombre rendu est différent de **n**, alors il y a eu une erreur (il faut tester **ferror** ou **errno**).

☞ écriture d'un fichier texte - fprintf

La fonction **fprintf** permet de réaliser le même travail que printf sur des fichiers ouverts en mode texte.

La fonction **int fprintf (FILE *fp, const char * format, void expression)** réalise une écriture formatée des éléments spécifiés par le 3ème argument sur un fichier préalablement ouvert par **fopen** en mode texte. Le format est spécifié par le 2ème argument.

Elle retourne le nombre d'octets écrits ou **EOF** s'il y a une erreur. Les "**\n**" sont transformés en **CR/LF** pour un fichier en mode texte (spécifique PC).

☞ autre écriture dans un fichier

int fputc(int , FILE *) écrit dans un fichier ouvert et associé au pointeur FILE* un caractère spécifié dans par le 1er argument. Ce caractère est converti en un unsigned char. Cette fonction retourne la valeur du caractère écrit dans un entier et en cas d'erreur, elle retourne **EOF**.

int fputs(const char *, FILE *) permet d'écrire une chaîne de caractères référencée par le 1er argument dans le fichier décrit par le 2ème argument. Le premier argument contient l'adresse de la zone mémoire qui contient les caractères à écrire. Cette zone doit être une chaîne de caractères (terminée par le caractère nul). Elle retourne valeur positive si l'écriture s'est correctement déroulée et en cas d'erreur d'écriture, la fonction retourne **EOF**.

IX – Action sur le pointeur de fichier

Les fonctions précédentes modifient de manière automatique le pointeur courant dans le fichier correspondant (adresse de l'octet dans le fichier à partir duquel se fait la prochaine opération d'entrée-sortie). Nous allons voir les fonctions qui permettent de connaître la valeur de cette position courante dans le fichier et de la modifier. Ces fonctions associées à la position dans le fichier sont :

int fseek(FILE *fp, long nb, int org) place le pointeur du flux indiqué (**fp**) à un endroit défini comme étant situé à **nb** octets de "l'origine" spécifiée par **org**.

- org = SEEK_SET (valeur = 0) correspond au début du fichier.
- org = SEEK_CUR (valeur = 1)à la position actuelle du pointeur.
- org = SEEK_END (valeur = 2)à la fin du fichier.

Elle retourne 0 en cas de succès et une valeur différente de zéro si le déplacement ne peut pas être réalisé.

long ftell(FILE *fp) retourne la valeur de la position courante du pointeur dans le fichier indiqué s'il n'y a pas d'erreur. Elle indique :

- le nombre d'octets entre la position courante et le début du fichier pour les fichiers binaires.
- une valeur permettant à **fseek** de repositionner le pointeur courant à l'endroit actuel pour les fichiers texte (**fseek** n'autorise que 0 ou la valeur retournée par **ftell** pour l'argument nb).

S'il y a erreur alors et la variable errno est modifiée.

X – Complément sur les unités standards d'entrée-sortie

En fait, l'exécution d'un programme commence systématiquement par l'ouverture trois pseudo-fichiers, ouverts par le système. Ils sont connectés aux organes d'entrée-sortie du PC. Ils ont pour nom (déclarés dans <stdio.h>) :

*FILE *stdin, *stdout, *stderr;*

stdin est l'unité standard d'entrée. Elle est habituellement affectée au clavier du poste de travail.
stdout est l'unité standard de sortie. Elle est habituellement affectée à l'écran du poste de travail.
stderr est l'unité standard d'affichage des erreurs. Elle est aussi affectée à l'écran du poste de travail.

À ces fichiers standards sont donc associées des fonctions prédéfinies qui permettent de réaliser les opérations suivantes :

- lecture et écriture caractère par caractère
- lecture et écriture de chaînes de caractères
- lecture et écriture formatées