Langage C

2008-2009

Bibliographie

Le langage C, B. W. Kernighan et D. M. Ritchie,
L'essentiel des structures de données en C, Horwitz, Sahni et Anderson-Freed
Le cours de Anne Canteaut, `http ://www-rocq.inria.fr/codes/Anne.Canteaut/COURS_C/`

Table des matières

Chapitre 1 : Notions De Base En Langage C

1. Généralités

Le langage C conçu en 1972 par D. Richie et K. Thompson pour développer le système d'exploitation : UNIX.
Un programme C est décrit par un fichier *texte*, appelé *fichier source*. Le programme est ensuite traduit en langage machine
grâce à un *compilateur.*

La compilation se décompose en trois phases successives :

- **Le traitement par préprocesseur :** transformations purement textuelles (remplacement de chaînes)
- **La compilation :** traduction en instructions compréhensibles par le microprocesseur (assembleur)
- **Assemblage et liaison :** assemblage des différents fichiers sources écrits et production d'un fichier dit *exécutable*

Le compilateur le plus utilisé est le compilateur gcc, disponible gratuitement pour Linux et Windows notamment

1.1. Un fichier source

Un programme C s'organise de la manière suivante:

```
[directives au préprocesseur]
[déclaration de variables globales]
[fonctions secondaires]

int main(void)
{
déclaration de variables locales
instructions
}
```

Pour simplifier, la déclaration des variables locales à une fonction se fait après le début du module, c'est à dire après l'accolade ouverte, celle **des variables globales** s'effectue à l'extérieur du corps de la fonction. Nous reviendrons plus en détail sur ces distinctions et sur la façon de les traduire en C.

Une instruction se termine par un **";"**, **les commentaires** sont encadrés par **"/*"** et par **"*/"**.

Le compilateur dissocie les minuscules des majuscules.

Les 32 mots clefs reconnus par le compilateur sont :

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

1.2. Déclarations

Les déclarations des objets manipulés par le programme permettent au compilateur de réserver un emplacement mémoire pour ces objets en leur affectant une adresse et une taille. Il y a deux genres d'objets: les objets constants et les variables. Le type des objets peut être prédéfini ou bien créé à l'aide du mot-clef **typedef**.

La déclaration se fait de la façon suivante : **type NomVar1;**

où **NomVar1** est une variable qui sera placée dans la zone de données.

Les types prédéfinis *:*

```
char
int
float          double
short          long          unsigned
```

- **Le type `char` :**

La plupart du temps une variable de type **char** est codée sur 1 octet (8 bits). Un caractère peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. Pour les langues d'Europe occidentale elle repose sur le codage ASCII (codage sur 7 bits des différents caractères). De plus en plus le codage UNICODE se répend, il repose sur un codage 16 bits et permet le codage des alphabets non Latin.

	déc.	hex.		déc.	hex.		déc.	hex.
	32	20	@	64	40	`	96	60
!	33	21	A	65	41	a	97	61
"	34	22	B	66	42	b	98	62
#	35	23	C	67	43	c	99	63
$	36	24	D	68	44	d	100	64
%	37	25	E	69	45	e	101	65
&	38	26	F	70	46	f	102	66
'	39	27	G	71	47	g	103	67
(	40	28	H	72	48	h	104	68
)	41	29	I	73	49	i	105	69
*	42	2a	J	74	4a	j	106	6a
+	43	2b	K	75	4b	k	107	6b
,	44	2c	L	76	4c	l	108	6c
-	45	2d	M	77	4d	m	109	6d
.	46	2e	N	78	4e	n	110	6e
/	47	2f	O	79	4f	o	111	6f
0	48	30	P	80	50	p	112	70
1	49	31	Q	81	51	q	113	71
2	50	32	R	82	52	r	114	72
3	51	33	S	83	53	s	115	73
4	52	34	T	84	54	t	116	74
5	53	35	U	85	55	u	117	75
6	54	36	V	86	56	v	118	76
7	55	37	W	87	57	w	119	77
8	56	38	X	88	58	x	120	78
9	57	39	Y	89	59	y	121	79
:	58	3a	Z	90	5a	z	122	7a
;	59	3b	[	91	5b	{	123	7b
<	60	3c	\	92	5c	|	124	7c
=	61	3d	]	93	5d	}	125	7d
>	62	3e	^	94	5e	~	126	7e
?	63	3f	_	95	5f	DEL	127	7f

Table 1 : Codes ASCII des caractères imprimables

```
main()
{
  char c = 'A';
  printf("%c", c );       // A s'affiche à l'écran
  printf("%d",c);         // 65 s'affiche à l'écran
  printf("%x",c);         // 41 s'affiche à l'écran
  c=c+1;
  printf("%c", c );       // B s'affiche à l'écran
  printf("%d",c);         // 66 s'affiche à l'écran
  printf("%x",c);         // 42 s'affiche à l'écran
}
```

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

- Les types entiers

Le mot-clef désignant le type entier est **int**. Un objet de type **int** est représenté par un mot "naturel" de la machine utilisée, 32 bits pour un DEC alpha ou un PC Intel. Un **char** peut également être vu comme un entier codé sur 8 bits.

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

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

Table 2 : Les types entiers

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

signed char	$[-2^7;2^7[$
unsigned char	$[0;2^8[$
short int	$[-2^{15}; 2^{15}[$

unsigned short int	$[0;2^{16}[$
int	$[-2^{31};2^{31}[$
unsigned int	$[0;2^{32}[$
long int (DEC alpha)	$[-2^{63};2^{63}[$

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

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

```
unsigned short x;

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

Dans les deux cas, **taille** vaudra 4.

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

- Les types flottants

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

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

Table 3: Les types flottants

Les flottants sont généralement stockés en mémoire sous la représentation de la virgule flottante normalisée.

- le type **float** est codé sur 4 octets et représente les nombres à valeur absolue dans l'intervalle 3.4 E-38 à 3.4 E+38,
- le type **double** est codé sur 8 octets et représente les nombres à valeur absolue dans l'intervalle 1.7 E-308 à 1.7 E+308,
- le type **long double** est codé sur 10 octets et représente les nombres à valeur absolue dans l'intervalle 3.4 E-4932 à 1.1 E+4932,

- **La déclaration des constantes :**
  Celle-ci peut se faire de deux façons :
  1- **#define NomConst valeurcte**
  (exemple : **#define a 5**)

la directive de compilation **#define** s'adresse au ***préprocesseur*** qui intervient juste avant le compilateur proprement dit. Le travail du préprocesseur consiste à traduire les lignes de programmation commençant par le caractère **#** de façon à préparer le travail du compilateur. Dans le cas de **#define**, le préprocesseur remplace dans

le fichier source toutes les occurrences de **NomConst** par sa valeur : **valeurcte**. ***La constante de nom* NomConst *n'a donc aucune existence physique***. Il existe d'autres directives de compilation dont certaines seront étudiées ultérieurement. Dans l'exemple toutes les occurrences de a seront remplacées par 5.

2- **const type NomConst = valeurcte;**
(ex: **const int b = 7;**)

le compilateur crée la variable constante **NomConst** en mémoire et lui affecte la valeur **valeurcte**. On pourra ainsi accéder à l'adresse de **NomConst** mais ***on ne pourra en aucun cas modifier sa valeur***. On précisera, le type de la constante. Dans l'exemple b est un entier qui vaut 7, il est accessible en lecture mais pas en écriture.

- ***Remarques :***

Il n'existe pas de type booléen en Langage C. Lors de l'utilisation de booléen (tests logiques...), il faudra se souvenir des conventions suivantes :
- un booléen est un entier,
- la valeur FAUX est un entier égal à zéro,
- la valeur VRAI est un entier égal à 1 dans le cas du résultat d'une évaluation, différent de zéro quand on cherche à évaluer.

Exemple : (3 > 4) a pour valeur 0 (FAUX) , (3 < 4) a pour valeur 1 (VRAI), et 3 est VRAI.

Exemple d'Initialisation lors de la déclaration

```
int i=2, j=-5; // i ppv 2 et j ppv –5
char c='A' ;   // c ppv 'A' c'est à dire 65 (code ASCII)
char d=65 ;     // d ppv 65 c'est à dire la lettre A
char e='\x41'; // e ppv 41 en Hexa, i.e. 65 en décimale soit la lettre A
```

2. Les Tableaux

2.1. Les tableaux à une dimension

La déclaration d'un tableau est la suivante

type NomTab [TAILMAX];

type est le type des éléments composant le vecteur, **NomTab** est le nom qui désigne le tableau, **TAILMAX** est la taille de déclaration c'est-à-dire le nombre d'éléments maximum utilisables.

Lors de la déclaration, le compilateur réserve la mémoire nécessaire pour le tableau c'est-à-dire qu'il réserve **TAILMAX*sizeof(type)** octets.
TAILMAX doit obligatoirement être une expression constante de type entier c'est-à-dire explicite ou définie grâce à une directive de compilation #define.

Par définition, en langage C, NomTab est l'adresse du premier élément du tableau, cette adresse est constante puisque c'est le compilateur qui lui a affecté une valeur lors de la déclaration du tableau. La valeur de cette adresse ne pourra en aucun cas être modifiée par le programme.

NomTab[i] désigne l'élément d'indice i du tableau (**ATTENTION les indices commencent à 0, par conséquent i pourra varier de 0 à TAILMAX-1**). Il désigne le contenu du tableau à la position **i+1** ; il s'agit donc d'une **valeur**.
&NomTab[i] caractérise l'adresse de l'élément **NomTab[i].**
Nous reviendrons plus en détail sur les notions d'adresses ultérieurement.

	Adresses	Valeurs
	&NomTab[TAILMAX-1]	NomTab[TAILMAX-1]
	&NomTab[TAILMAX-2]	NomTab[TAILMAX-2]
	...	
	&NomTab[2]	NomTab[2]
	&NomTab[1]	NomTab[1]
NomTab :	&NomTab[0]	NomTab[0]

L'initialisation d'un tableau peut se faire en même temps que sa déclaration.

Exemple :

```
float vect [ 4 ] = { -5.3 , 4.88 , 2.67 , -45.675 };
```
place -5.3 dans vect [0], 4.88 dans vect [1], 2.67 dans vect [2] et -45.675 dans vect [3] ;

```
float vect [ ] = { -5.3 , 4.88 , 2.67 , -45.675 };
```
place -5.3 dans vect [0], 4.88 dans vect [1], 2.67 dans vect [2] et -45.675 dans vect [3] et fixe sa dimension à 4.

2.2. Les tableaux à plusieurs dimensions

La déclaration d'un tableau à deux dimension est

```
type NomTab [ LIGMAX ] [ COLMAX ];
```

type est le type des éléments composant le tableau, **NomTab** est le nom qui désigne le tableau, **LIGMAX** est la taille de déclaration de la longueur maximale des lignes, **COLMAX** est la taille de déclaration de la longueur maximale des colonnes.

Lors de la déclaration, le compilateur réserve la mémoire nécessaire pour le tableau c'est-à-dire qu'il réserve **LIGMAX*COLMAX*sizeof(type)** octets. **LIGMAX et COLMAX** doivent obligatoirement être des expressions constantes de type entier c'est-à-dire explicites ou définies grâce à une directive de compilation **#define.**

Par définition, en langage C, **NomTab** est l'adresse du tableau, cette adresse est constante puisque c'est le compilateur qui lui a affecté une valeur lors de la déclaration du tableau. La valeur de cette adresse ne pourra en aucun cas être modifiée par le programme.

NomTab[i][j] désigne l'élément de la ligne i et de la colonne j (ATTENTION les indices commencent à 0, par conséquent i pourra varier de **0 à LIGMAX-1 et j** pourra varier de **0 à COLMAX-1**). Il s'agit donc d'une **valeur**.
&NomTab[i][j] désigne l'adresse de l'élément **NomTab[i][j]**.

Les éléments sont rangés en mémoire conformément à la figure ci dessous.

	Adresse éléménts	Valeurs
	&NomTab[LIGMAX-1][COLMAX-1]	NomTab[LIGMAX-1][COLMAX-1]
		...
	&NomTab[LIGMAX-1][0]	NomTab[LIGMAX-1][0]
NomTab[LIGMAX-1]	&NomTab[LIGMAX-1][0]	NomTab[LIGMAX-1][0]
		...
	&NomTab[1][COLMAX-1]	NomTab[1][COLMAX-1]

			...
		&NomTab[1][1]	NomTab[1][1]
	NomTab[1]	&NomTab[1][0]	NomTab[1][0]
		&NomTab[0][COLMAX-1]	NomTab[0][COLMAX-1]
			...
		&NomTab[0][1]	NomTab[0][1]
NomTab	NomTab[0]	&NomTab[0][0]	NomTab[0][0]

Exemple de déclaration et d'initialisation d'un tableau :

```
int Tab[2][2];
float Mat[2][4]= {{-5.3,4.88,2.67,-45.675} ,{3.72,-2.6,35.1,0.25} };
```

Les dimensions supérieures respectent le même principe.

```
int Tab[2][2][7][4]; // déclaration d'un tableau de dimension 4
```

3. Les Opérateurs

3.1. Opération d'affectation

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

variable = expression

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

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

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

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

3.2 Les opérateurs arithmétiques

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

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

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

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

  ```
  float x;
  x = 3 / 2;
  ```

  affecte à **x** la valeur 1. Par contre

  ```
  x = 4.4 / 2;
  ```

affecte à **x** la valeur **2.2**.

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

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

3.3. Les opérateurs relationnels

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

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type **int** (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.
Attention à ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =. Ainsi, le programme ci dessous imprime à l'écran **a et b sont egaux** !

```
main()
{
  int a = 0;
  int b = 1;
  if (a = b) // L'ERREUR EST ICI
    printf("\n a et b sont egaux \n");
  else
    printf("\n a et b sont differents \n");
}
```

3.4 Les opérateurs logiques booléens

&&	ET logique
||	OU logique
!	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un **int** qui vaut 1 si la condition est vraie et 0 sinon.
Dans une expression de type

expression-1 op-1 expression-2 op-2 ...expression-n

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i=12;
int p=3;

if ((i >= 0) && (i <= 9) && !(p== 0))
```

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

3.5 Les opérateurs logiques bit à bit

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

&	ET		**|**	OU inclusif
^	OU exclusif		**~**	complément à 1
<<	décalage à gauche		**>>**	décalage à droite

En pratique, les opérateurs **`&, |`** et **`^`** consistent à appliquer bit à bit les opérations suivantes :

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

L'opérateur unaire ~ change la valeur de chaque bit d'un entier. Le décalage à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît).

Considérons par exemple les entiers **`a=77`** et **`b=23`** de type **`unsigned char`** (*i.e.* 8 bits). En base 2 il s'écrivent respectivement 01001101 et 00010111.

	valeur		
expression	binaire	décimale	
`a`	**`01001101`**	**`77`**	
`b`	**`00010111`**	**`23`**	
`a & b`	**`00000101`**	**`5`**	
`a | b`	**`01011111`**	**`95`**	
`a ^ b`	**`01011010`**	**`90`**	

~a	**10110010**	**178**	
b << 2	**01011100**	**92**	multiplication par 4
b << 5	**11100000**	**112**	ce qui dépasse disparaît
b >> 1	**00001011**	**11**	division entière par 2

3.6 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

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

Pour tout opérateur *op*, l'expression

expression-1 op= expression-2

est équivalente à

expression-1 = expression-1 op expression-2

Toutefois, avec l'affection composée, **expression-1** n'est évaluée qu'une seule fois.

3.7 Les opérateurs d'incrémentation et de décrémentation

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

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

3.8 L'opérateur virgule

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

expression-1, expression-2, ... , expression-n

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

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

imprime `b = 5`.

3.9 L'opérateur conditionnel ternaire

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

condition **?** ***expression-1*** **:** ***expression-2***

Cette expression est égale à **expression-1** si **condition** est satisfaite, et à **expression-2** sinon. Par exemple, l'expression

x >= 0 ? x : -x

correspond à la valeur absolue d'un nombre. De même l'instruction

```
m = ((a > b) ? a : b);
```

affecte à `m` le maximum de `a` et de `b`.

3.10 L'opérateur de conversion de type

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

(*type*) ***objet***

Par exemple,

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

retourne la valeur 1.5.

3.11 L'opérateur adresse

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

&objet

3.12 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est définie par la flèche de la seconde colonne du tableau. On préférera toutefois mettre des parenthèses en cas de doute...

opérateurs	
`()` `[]` `->` `.`	→
`!` `~` `++` `--` -(unaire) (*type*) `*(indirection)` `&`(adresse) `sizeof`	←
`*` `/` `%`	→
`+` -(binaire)	→
`<<` `>>`	→
`<` `<=` `>` `>=`	→
`==` `!=`	→

& (et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	←
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

Table 1.4: Règles de priorité des opérateurs

Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions. Par exemple, il faut écrire **`if ((x ^ y) != 0)`**

4. Les Entrees-sorties Conversationnelles

Il s'agit des fonctions de la librairie standard **stdio.h** utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran.

4.1. Les sorties ou écritures a l'écran

La fonction printf :

Prototype : **int printf (const char *format, ...)**

La valeur retournée par **printf** est le nombre de caractères écrits ou une valeur négative en cas d'erreur. Cette valeur est rarement utilisée. Les paramètres de la fonction sont constitués des formats et des variables qui doivent être affichées. ***Le code format et le type des arguments doivent être en correspondance.***

Exemple :

soit **n** une variable entière contenant la valeur 12 et **moy** une variable réelle de valeur 13,67,
écrire 'la moyenne des ', **n**,' notes de l'étudiant est égale à ', **moy**
se traduit par

```
  int n=12;
  float moy=13.67;
  printf ( "la moyenne des %d notes de l'étudiant est égale à %f \n", n,
moy);
```

On verra alors sur l'écran : **la moyenne des 12 notes de l'étudiant est égale à 13.67**

format	conversion en	écriture
%d	**int**	décimale signée
%ld	**long int**	décimale signée
%u	**unsigned int**	décimale non signée
%lu	**unsigned long int**	décimale non signée
%o	**unsigned int**	octale non signée
%lo	**unsigned long int**	octale non signée
%x	**unsigned int**	hexadécimale non signée
%lx	**unsigned long int**	hexadécimale non signée
%f	**float**	décimale virgule fixe
%lf	**double**	décimale virgule fixe
%e	**float**	décimale notation exponentielle
%le	**long double**	décimale notation exponentielle
%g	**float**	décimale, représentation la plus courte parmi `%f` et `%e`
%lg	**long double**	décimale, représentation la plus courte parmi `%lf` et `%le`
%c	**unsigned char**	caractère
%s	**char***	chaîne de caractères

Table 5 :Caractères de conversion pour la fonction **`printf`**.

On peut avoir besoin d'inclure dans la chaîne de caractères ***format***, un certain nombre de ***caractères spéciaux*** dont la liste est donnée ci-dessous.

caractère en C	signification	abréviation
`\n`	fin de ligne	LF
`\t`	tabulation horizontale	HT
`\v`	tabulation verticale	VT
`\b`	retour arrière	BS
`\r`	retour chariot	CR
`\f`	saut de page	FF
`\a`	signal sonore	BEL
`\\`	barre oblique inverse	\
`\?`	point d'interrogation	?
`\'`	apostrophe	'
`\"`	guillemet	"

Table 6 :Caractères spéciaux

4.2. Les entrées ou lectures au clavier

4.2.1 La fonction (plus justement macro) `getchar` (dans `stdio.h`).

Prototype : **`int getchar ( void )`**

Elle renvoie un entier qui est le code ASCII du caractère frappé au clavier. Elle permet donc de lire au clavier un caractère et de le stocker dans une variable choisie.

Exemple :

```
char car ;            // déclaration variable car
car = getchar() ;     // lecture d'un caractère et mise en place dans car
```

Il faut être vigilant lorsque l'on utilise cette macro puisque les données frappées au clavier ne seront pas prises en compte tant que vous ne les aurez pas validées par "retour chariot".

4.2.2. La fonction **`scanf`** (dans **`stdio.h`**)

Prototype : **`int scanf ( const char *format, ... ).`**

La valeur retournée par **`scanf`** est le nombre d'objets convertis et affectés ou la valeur **`EOF`** en cas d'erreur. Cette valeur est rarement utilisée.

Les paramètres de la fonction sont constitués des formats et de l'adresse des variables où doivent être stockées les données saisies au clavier. ***Le code format et le type des arguments doivent être en correspondance.***

format	type d'objet pointé	représentation de la donnée saisie
`%d`	**`int`**	décimale signée
`%hd`	**`short int`**	décimale signée
`%ld`	**`long int`**	décimale signée
`%u`	**`unsigned int`**	décimale non signée
`%hu`	**`unsigned short int`**	décimale non signée
`%lu`	**`unsigned long int`**	décimale non signée
`%o`	**`int`**	octale
`%ho`	**`short int`**	octale
`%lo`	**`long int`**	octale
`%x`	**`int`**	hexadécimale

`%hx`	`short int`	hexadécimale
`%lx`	`long int`	hexadécimale
`%f`	`float`	flottante virgule fixe
`%lf`	`double`	flottante virgule fixe
`%Lf`	`long double`	flottante virgule fixe
`%e`	`float`	flottante notation exponentielle
`%le`	`double`	flottante notation exponentielle
`%Le`	`long double`	flottante notation exponentielle
`%g`	`float`	flottante virgule fixe ou notation exponentielle
`%lg`	`double`	flottante virgule fixe ou notation exponentielle
`%Lg`	`long double`	flottante virgule fixe ou notation exponentielle
`%c`	`char`	caractère
`%s`	`char*`	chaîne de caractères

Table 7: Formats de saisie pour la fonction **`scanf`**

Exemple : la saisie d'un entier s'écrit :

```
int n;
printf ( "Donnez le nombre de notes : " );
fflush(stdin);
scanf ( "%d", &n );
printf("La valeur entrée est : %d \n", n );
```

Conseil :

1: videz systématiquement le flux d'entrée avant lecture (utilisez `fflush(stdin);`)

2. n'utilisez pas `scanf` pour la saisie de chaîne de caractères !! En effet elle ne permet pas la saisie d'espace dans une chaîne de caractère. Ainsi `scanf(« %s »,...)` est à proscrire,

4.3. Saisie d'une chaîne de caractères

Une chaîne de caractères est un tableau de caractères, pour être interprétée correctement le nombre de caractères de la chaîne doit être inférieur à la place disponible dans le tableau moins un caractère, de plus le dernier caractère est le caractère nul '\0' (dont la valeur est 0).

Ainsi une chaîne déclarée comme suit :

```
char chaine[10]=""; // chaîne vide initialisée avec chaine[0]='\0'
```

ne pourra comporter que 9 caractères placés de chaine[0] à chaine[8].

4.3.1. La fonction **fgets** :

Prototype : **char *fgets(char *s, int n, FILE *stream);**

fgets lit des caractères depuis le flux d'entrée **stream** et les place dans la chaîne **s**. La fonction cesse la lecture soit lorsque **n-1** caractères ont été lus, soit suite à la lecture d'un caractère de saut de ligne (**\n**) (ce caractère est copié en fin de chaîne). La fin de la chaîne de caractères est marquée par l'ajout d'un caractère nul (**\0**).

Le flux d'entrée **stdin** correspond au clavier. Nous pouvons donc utiliser **fgets** pour la saisie clavier :

Par conséquent pour saisir une chaîne de caractères vous devrez saisir :

```
int main(void)
{
char chaine[15]="";
int longueur=0;
printf(" Entrez une chaine de caracteres (14 caractere max) \n");
fflush(stdin);             // on vide le tampon
fgets(chaine,15,stdin);  // on extrait au maximum 14 caractères du flux stdin
                         // on stocke ces caractères a l'adresse chaîne
longueur= strlen(chaine); // on récupère la longueur de la chaîne moins 1
if (chaine[longueur-1]=='\n')
                  // si le dernier caractère de la chaîne est retour chariot
'\n'
{longueur--;
chaine[longueur]='\0'; // alors on le remplace par le caractère de fin de
chaîne
}
printf("voici la chaine :%s",chaine); // affiche la chaîne
}
```

5. Les Instructions De Contrôle

5.1. Choix simple ou action conditionnelle `if else`

Forme générale

```
if ( condition )
{
  actions_alors;
}
else
{
  actions_sinon;
}
```

Exemple

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

5.2. Choix multiple `switch`

Forme générale

```
switch (choix )
{
  case val1 :  action1;
               break;
  case val2 :  action2;
               break;
    ...
  case valn :  actionn;
               break;
default : action_autre;
}
```

Exemple

```
switch (n )
{
  case 0 :      printf("nul\n");
               break;
 case 1 :
 case 3 :      printf("trois\n");
         break;
 default :     printf("erreur\n");
}
```

La valeur de choix ne peut être qu'un entier ou un caractère. ***Les expressions `val1` à `valn` doivent être des constantes entières ou caractères***. ***L'instruction `break` fait sortir de la structure de choix multiple***. Si elle est absente à la fin du bloc d'instructions de valeur ***`vali`***, l'exécution se poursuit avec les instructions de la valeur ***`vali+1`***.

5.3. Boucle `while`

La syntaxe de **`while`** est la suivante :

```
while (expression )
   instruction
```

Tant que **`expression`** est vérifiée (*i.e.*, non nulle), **`instruction`** est exécutée. Si ***`expression`*** est nulle au départ, ***`instruction`*** ne sera jamais exécutée. **`instruction`** peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
int main(void)
{
int i = 1;
while (i < 10)
  {
    printf("\n i = %d",i);
    i++;
  }
return 0;
  }
```

Ou encore pour le parcours d'une chaîne de caractères :

```
int main(void)
{
int i = 0;
char NomPhysicien[20]="Marie Curie"
// Affichage de la chaine NomPhysicien
while (NomPhysicien[i]!= '\0')
  {
    printf("%c",NomPhysicien[i]);
    i++;
  }
printf("\n");
// Affichage de la chaine NomPhysicien avec le format %s
printf("%s\n",NomPhysicien);
return 0;
  }
```

5.4. Boucle `do---while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle **`do---while`**. Sa syntaxe est

```
do
  instruction
while (expression );
```

Ici, ***`instruction`*** sera exécutée tant que **`expression`** est non nulle. Cela signifie donc que **`instruction`** est

toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int main(void)
{
int a;

do
  {
    printf("\n Entrez un entier entre 1 et 10 : ");
    scanf("%d",&a);
  }
 while ((a <= 0) || (a > 10));
return 0;
}
```

Ou encore pour contraindre une réponse :

```
int main(void)
{
char choix=0;

do
  {
    printf("\n Etes vous optimiste pour votre avenir (O/N) : ");
    fflush(stdin);
    choix=getchar();
    choix=toupper(choix);
    if(choix!='N' && choix!='O') printf("Repondre par (O ou N)");
  }
while(choix!='N' && choix!='O');

return 0;
}
```

5.5. Boucle `for`

La syntaxe est :

```
for (expr 1 ;expr 2 ;expr 3)
instruction
```

La première expression du **`for`** (**`expr 1`**) est évaluée avant d'entrer dans la boucle, la seconde (**`expr 2`**) conditionne la poursuite du **`for`**, la troisième (**`expr 3`**) est évaluée à chaque fin de parcours. Une version équivalente plus intuitive est:

```
expr 1;
while (expr 2 )
{instruction
expr 3;
}
```

5.6. Gestion de caisse

```
#include <stdio.h>                          /* pour les lectures écritures*/
#include <ctype.h>                          /* pour la fonction toupper*/

#define TVA 0.186

int main (void )
{
/* déclarations */

int nb_ventes=0,na=0;
float total_ventes=0, meilleur_vente=0;
float pu=0, prixHT=0, taxe=0, prixTTC=0;
char reponse;


/* boucle de saisie et de calcul */
do
{
     nb_ventes = nb_ventes + 1;
     /* traitement de la vente courante */
     /* demandes */
     printf ( "Quel est le prix unitaire HT de votre article ? " );
     fflush(stdin);
     scanf ( " %f ", &pu );
     printf ( "Combien avez-vous d'articles ? " );
     fflush(stdin);
     scanf( " %n ", &na );
     /*calculs */
     prixHT = pu * na;
     taxe = prixHT * TVA;
     prixTTC = prixHT + taxe ;
     /* affichage */
     printf ( "vos %d articles à %f euros piece coutent %f HT, et %f TTC \n", na, pu,
              prixHT, prixTTC );
     /* mise à jour des récapitulatifs */
     total_ventes += prixTTC;
     if ( prixTTC > meilleur_vente )
     {
        meileur_vente = prixTTC;
     }
     printf ("Voulez-vous continuer ? ");
     fflush(stdin);
     reponse = getchar();
}
while ( toupper (reponse) != NON );
/* affichage des résultats */
printf (" Vous avez effectué %d ventes \n ", nb_ventes);
printf ( " la meilleure de vos ventes est %f , et le total des ventes s'élève à %f \n ",
            meilleur_vente, total_ventes );}
return 0;
}
```

Chapitre 2 : Structures – Unions – Enumération

1. Définition De Types Composés avec `typedef`

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **`typedef`** :

`typedef` ***type synonyme;***

Par exemple:

```
typedef unsigned int UINT; // UINT est synonyme de unsigned int

int main(void)
{
  UINT z; // z sera un unsigned int
}
```

2. Les Structures

Une *structure* est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est ***`modele`*** suit la syntaxe suivante :

```
struct modele
{type_1 membre_1;
  type_2 membre_2;
    ...
  type_n membre_n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

`struct` ***`modele objet;`***

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele
{ type_1 membre_1;
  type_2 membre_2;
    ...
  type_n membre_n;
}objet;
```

Il est également possible de faire appel à **`typedef`** afin de systématiser la syntaxe :

```
// déclaration d'un type structure
typedef struct
{     type_1 membre_1;
      type_2 membre_2;

      type_n membre_n;
```

```
} TypeModele;
// déclaration d'un objet de ce type

TypeModele objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur *membre de structure*, noté **"."**. Le **i**-ème membre de **objet** est désigné par l'expression

objet.membre_i

On peut effectuer sur le **i**-ème membre de la structure toutes les opérations valides sur des données de type ***type_i***. Par exemple, le programme suivant définit la structure **complexe**, composée de deux champs de type **double** ; il calcule la norme d'un nombre complexe.

```
#include <math.h>
typedef struct
{
  double reelle;
  double imaginaire;
} TComplexe;

int main(void)
{
  Tcomplexe z;
  double norme;
    ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
  return 0;
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
Tcomplexe z = {2.1 , 2.1};
```

En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
...
int main(void)
{
      Tcomplexe z1, z2;
      z1.r=1;
      z1.i=2;...
      z2 = z1;
      return 0;
}
```

3. Les Champs De Bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (`int` ou `unsigned int`). Cela se fait en précisant le nombre de bits du champ avant le **;** qui suit sa déclaration. Par exemple, la structure suivante

```
struct registre
{
  unsigned int actif : 1;
  unsigned int valeur : 31;
```

```
};
```

possède deux membres, **actif** qui est codé sur un seul bit, et **valeur** qui est codé sur 31 bits. Tout objet de type **struct registre** est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation.
Le champ **actif** de la structure ne peut prendre que les valeurs 0 et 1. Aussi, si `r` est un objet de type **struct registre**, l'opération **r.actif += 2;** ne modifie pas la valeur du champ.
La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier.
Notons enfin qu'un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur **&.**

4. Les Unions

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

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

```
union jour
{
  char lettre;
  int numero;
};

int main(void)
{
  union jour hier, demain;
  hier.lettre = 'J';
  printf("hier = %c\n",hier.lettre);
  hier.numero = 4;
  demain.numero = (hier.numero + 2) % 7;
  printf("demain = %d\n",demain.numero);
  return 0;
}
```

Les *unions* peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type **unsigned int** d'une structure en les identifiant à un objet de type **unsigned long** (en supposant que la taille d'un entier **long** est deux fois celle d'un **int**).

```
typedef struct
{
  unsigned int x;
  unsigned int y;
} TCoordonnees;

union point
{
  Tcoordonnees coord;
  unsigned long mot;
};

int main(void)
{
```

```
  union point p1, p2, p3;
  p1.coord.x = 0xf;
  p1.coord.y = 0x1;
  p2.coord.x = 0x8;
  p2.coord.y = 0x8;
  p3.mot = p1.mot ^ p2.mot;
  printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
  return 0;
}
```

5. Les énumérations

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

enum *modele* {*constante_1*, *constante_2*,...,*constante_n*};

En réalité, les objets de type **enum** sont représentés comme des **int**. Les valeurs possibles ***constante_1, constante_2,...,constante_n*** sont codées par des entiers de **0** à **n-1**. Par exemple, le type **enum booleen** défini dans le programme suivant associe l'entier **0** à la valeur **faux** et l'entier **1** à la valeur **vrai**.

```
int main(void)
{
  enum booleen {faux=0, vrai=1};
  enum booleen b;
  b = vrai;
  printf("b = %d\n",b);
  return 0;
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

```
enum booleen {faux = 12, vrai = 23};
```

Chapitre 3 : Les Pointeurs

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

1. Adresse et valeur d'un objet

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

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

Dans l'exemple,

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable `i` à l'adresse 4831836000 en mémoire, et la variable `j` à l'adresse 4831836004, on a

objet	adresse	valeur
i	**&i= 4831836000**	3
j	**&j= 4831836004**	3

Deux variables différentes ont des adresses différentes. L'affectation **`i = j;`** n'opère que sur les valeurs des variables. Les variables **`i`** et **`j`** étant de type **`int`**, elles sont stockées sur 4 octets. Ainsi la valeur de `i` est stockée sur les octets d'adresse 4831836000 à 4831836003.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures.

L'opérateur **`&`** permet d'accéder à l'adresse d'une variable. Toutefois **`&i`** n'est pas une *Lvalue* mais une constante, en effet on ne peut pas faire figurer **`&i`** à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

2. Notion de pointeur

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

`type *`*`nom-du-pointeur;`

où ***`type`*** est le type de l'objet pointé. Cette déclaration déclare un identificateur, ***`nom-du-pointeur,`*** associé à un objet dont la valeur est l'adresse d'un autre objet de type ***`type`***. L'identificateur **`nom-du-pointeur`** est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle *Lvalue*, sa valeur est modifiable.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du

type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type **`char`**, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type **`int`**, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké.

L'*opérateur* ***unaire d'indirection*** **`*`** permet d'accéder directement à la valeur de l'objet pointé. Avant d'utiliser cet opérateur unaire il faut veiller à ce que le pointeur soit initialisé.

La constante symbolique NULL (0 en décimale) affectée à un pointeur signifie, par convention, qu'il ne pointe sur aucun objet.

Exemple de programme :

```
int main(void)
{
  int i = 3;
  int *p = &i;    // p pointe sur i
  printf("*p = %d i= %d\n",*p,i);

  *p=0;
  printf("*p = %d i=%d\n",*p,i);
  return 0;
}
```

imprime
`*p = 3 i=3`
`*p=0 i=0.`

Dans ce programme, les objets **`i`** et **`*p`** sont identiques. Nous sommes dans la configuration :

objet	adresse	valeur
`i`	**`&i=4831836000`**	**`3`**
`p`	**`&p=4831836004`**	**`4831836000`**

On peut donc dans un programme manipuler à la fois les objets **`p`** et **`*p`**. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

```
int main(void)
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
  return 0;
}
```

```
int main(void)
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
  return 0;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

objet	adresse	valeur
i	**&i=4831836000**	**3**
j	**&j=4831836004**	**6**
p1	**&p1=4831835984**	**4831836000**
p2	**&p2=4831835992**	**4831836004**

Après l'affectation **`*p1 = *p2;`** du premier programme, on a

objet	adresse	valeur
i	**&i=4831836000**	**6**
j	**&j=4831836004**	**6**
p1	**&p1=4831835984**	**4831836000**
p2	**&p2=4831835992**	**4831836004**

Par contre, l'affectation **`p1 = p2`** du second programme, conduit à la situation :

objet	adresse	valeur
i	**&i=4831836000**	**3**
j	**&j=4831836004**	**6**
p1	**&p1=4831835984**	**4831836004**
p2	**&p2=4831835992**	**4831836004**

3. Arithmétique des pointeurs

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

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier correspondant au nombre d'objets présents entre les deux adresses.

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

Si **`i`** est un entier et **`p`** est un pointeur sur un objet de type ***type***, l'expression **`p+i`** désigne un pointeur sur un objet de type ***type*** dont la valeur est égale à la valeur de **`p`** incrémentée de **`i*sizeof(`*type*`)`**. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation ++ et --. Par exemple, le programme

```
int main(void)
{
  int i = 3;
  int *p1=NULL, *p2=NULL;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
return 0;
}
```

affiche **p1 = 4831835984 p2 = 4831835988.** // pour des int codés sur 4 octets

Par contre, le même programme avec des pointeurs sur des objets de type **double** :

```
int main(void)
{
  double i = 3;
  double *p1=NULL, *p2=NULL;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
  return 0;
}
```

affiche **p1 = 4831835984 p2 = 4831835992**.

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

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux. Ainsi, le programme suivant imprime les éléments du tableau **tab** dans l'ordre croissant des indices.

```
#define N 5

int main(void)
{ int tab[N] = {1, 2, 6, 0, 7};
  int *p=NULL;
  int i=0;
  printf("\n ordre croissant:\n");
  for (p = &tab[0]; p <= &tab[N-1]; p++)    printf(" %d \n",*p);


// de manière équivalente

  p=&tab[0]; // <=> p=tab;
  printf("\n ordre croissant:\n");
  for(i=0;i<N;i++)  printf(" %d \n",p[i]);

  return 0;
}
```

En résumé, si **p=tab** (ou encore **p=&tab[0]**), il y a les équivalences suivantes :

p[i] <=> *(p+i) <=> tab[i]

&p[i] <=> (p+i) <=> &tab[i]

Si **p** et **q** sont deux pointeurs sur des objets de type ***type***, l'expression **p-q** désigne un entier dont la valeur est égale à **(p – q)/sizeof(*type*), c'est à dire le nombre d'objet de type *type* qui peuvent prendre palce entre les adresses p et q.**

4. Pointeurs et constantes

Le mot-clef **const** permet de protéger des zones en écriture, associé à un pointeur cela conduit à plusieurs cas de

figure.

```
int i=0,j=0;
1. const int  *pi=&i;   /* pi pointe sur i, c'est *pi qui est constant */
2. int *const pj=&j;    /* pj pointe sur j, c'est pj qui est constant */
3. const int *const pk=&k; /* pk pointe sur k, pk et *pk sont constants */
```

L'exemple ci dessous explicite les comportements :

```
#include <stdio.h>

void main(void)
{
  int i, j, k;
  const int *pi=&i;     /* pi pointe sur i, c'est *pi qui est constant */
  int *const pj=&j;     /* pj pointe sur j, c'est pj qui est constant */
  const int *const pk=&k; /* pk pointe sur k,  pk et *pk sont constants */

  pi = &j;          /* instruction légale */
  *pi = 5;          /* erreur de compilation, protection en écriture de *pi */
  i++;                  /* instruction légale */
  pj = &i;              /* erreur de compilation,protection en écriture de pj */
  *pj = i;              /* instruction légale */
  j++;                  /* instruction légale */
  pk = &i;              /* erreur de compilation */
  *pk = j;              /* erreur de compilation */
  k++;                  /* instruction légale */
}
```

5. Pointeurs et structures

Contrairement aux tableaux, les objets de type structure en C sont des *Lvalues*. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Si **p** est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression

(*p).*membre*

L'usage de parenthèses est ici indispensable car l'opérateur d'indirection ***** à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté **->**. L'expression précédente est strictement équivalente à

p->*membre*

Dans l'exemple ci dessous il est montré comment un pointeur permet de désigner un élément d'un tableau de structure.

```
#include <stdlib.h>
#include <stdio.h>
#define N 96

typedef struct
{
  int id;
  int poids;
} TPoulet;
```

```
int main(void)
{
  int n=0, i;
  TPoulet tab[N];
  Tpoulet * pSurPoulet=NULL; // pointeur pointant sur rien
do
{

  printf("nombre de Poulets (0-%d)?",N);
  fflush(stdin);
  scanf("%d",&n);
} while (n<0 && n>N);

  for (i =0 ; i < n; i++)
    {
      printf("\n saisie du Poulet numero %d\n",i);
      printf("identifiant du Poulet = ");
      fflush(stdin);
      scanf("%d",&tab[i].id);
      fflsuh(stdin);
      printf("\n poids en g = ");
      scanf("%d",&tab[i].poids);
    }

do
{

  printf("\n Entrez un numero  ");
  fflush(stdin);
  scanf("%d",&i);
} while (i<0 && i>n);

  pSurPoulet=&tab[i];

  printf("\n Poulet numero %d:",i);
  printf("\n Identifiant %d:",pSurPoulet->id);
  printf("\n poids = %d\n",pSurPoulet->poids);

  return 0;
}
```

Chapitre 4 : Les Chaînes De Caractères

1. Définition Et Déclaration Des Chaînes De Caractères En C

Comme dit précédemment ***une chaîne est vue comme un tableau de caractères***. C'est un tableau qui se termine par le caractère nul **'\0'** (dont la valeur est 0) de façon à pouvoir en détecter la fin. La place qu'il occupe en mémoire est donc supérieure d'une unité au nombre de caractères effectifs qu'il peut contenir.

Il y a plusieurs façons de déclarer une chaîne de caractères :

```
char nom[10] = "le chat";
```
ou
```
char nom[ ] = "le chat"; // taille ajustée automatiquement à 8
```

Remarques :

- Une chaîne de caractères constante est un texte encadré de **"** et de **"**, à ne pas confondre avec un caractère constant qui lui est encadré de **'** et de **'**. Ainsi 'A' est le caractère A alors que "A" est la chaîne A c'est-à-dire le caractère A suivi du caractère nul (**'\0'**).

- Une chaîne de caractères constante se voit affecter une adresse par le compilateur lors de son écriture, ainsi les lignes suivantes sont valides :

```
char *pNom;
pNom = "le chat";
```

l'instruction d'affectation réalise l'affectation de l'adresse de "le chat" au pointeur **pNom**. ***Ce n'est en aucun cas une copie de chaîne (ou de caractères).***

Le pointeur **pNom** peut alors être utilisé pour parcourir une chaîne de caractères :

```
int main(void)
{
int i=0;
char *pNom;
char Chaine[] = "le chat";
pNom=Chaine; // <=> pNom=&Chaine[0];
while (*pNom!='\0')
{
printf("%c",*pNom);
*pNom++;
}
printf("\n");
// ce qui est équivalent à
while (pNom[i]!='\0')
{
printf("%c",pNom[i]);
i++;
}
printf("\n");
// ou encore
printf("%s\n",pNom);
return 0;
}
```

2. Saisie d'une chaîne de caractères `fgets`

Prototype dans <stdio.h>
char * fgets (char * str, int num, FILE * stream);

Lit les caractères depuis le flux d'entrée **`stream`** et les place dans la chaîne **`str`** tant que **`num-1`** caractères n'ont pas été lus et que le caractère lu est différent du caractère saut de ligne (**`'\n'`**) (ce caractère est copié en fin de chaîne) et que l'on n'est pas en fin de flux. La fin de la chaîne de caractères est marquée par l'ajout d'un caractère nul (**`\0`**). Cette fonction retourne **`str`** en cas de succès, et un pointeur **`NULL`** en cas de fin de flux.

Le flux d'entrée **`stdin`** correspond au clavier. On rappelle donc ici les instructions préconisées pour assurer la saisie clavier d'une chaîne de caractères :

```
int main(void)
{
char chaine[15]="";
int longueur=0;
printf(" Entrez une chaine de caracteres (14 caractere max) \n");
fflush(stdin);             // on vide le tampon
fgets(chaine,15,stdin);  // on extrait au maximum 14 caractères du flux stdin
                         // on stocke ces caractères a l'adresse chaîne
longueur= strlen(chaine); // on récupère la longueur de la chaîne moins 1
if (chaine[longueur-1]=='\n')
                  // si le dernier caractère de la chaîne est retour chariot
'\n'
{longueur--;
chaine[longueur]='\0';  // alors on le remplace par le caractère de fin de
chaîne
}
printf("voici la chaine :%s",chaine); // affiche la chaîne
}
```

3. Recopie d'une chaîne de caractères `strcpy, strncpy`

Prototype dans <string.h>
char * strcpy (char * *destination*, const char * *source*);

Copie la chaîne pointée par ***`source`*** vers la chaîne pointée par ***`destination`***. Pour éviter des débordements de mémoire il faut s'assurer que la taille du tableau pointé par destination est suffisante pour contenir la chaîne ***`source`*** (caractère de fin de chaîne compris). Le pointeur ***`destination`*** est retourné.

Prototype dans <string.h>
char * strncpy (char * *destination*, const char * *source*, size_t *num*);

Copie les ***`num`*** premiers caractères de ***`source`*** vers ***`destination.`*** Si la fin de la chaîne ***`source`*** est rencontré avant que ***`num`*** caractères soient copiés ***`destination`*** est complétés de caractères ***`nul ('\0').`*** Le pointeur ***`destination`*** est retourné.

4. Concaténation de chaînes `strcat, strncat`

Prototype dans <string.h>

```
char * strcat ( char * destination, const char * source );
```

Ajoute une copie de la chaîne ***source*** à la chaîne ***destination***. La caractère de fin de chaîne de ***destination*** est remplacé par le premier caractère de ***source***, et un nouveau caractère de fin de chaîne est ajouté à la fin de la nouvelle chaîne résultante ***destination***. Le tableau de caractères pointé par ***destination*** doit être suffisamment grand pour accueillir la chaîne concaténée. Le pointeur ***destination*** est retourné.

```
Prototype dans <string.h>
char * strncat ( char * destination, char * source, size_t num );
```

Ajoute les ***num*** premiers caractères de ***source*** à ***destination***, plus un caractère fin de chaîne. Si la longueur de la chaîne ***source*** est inférieure à ***num***, seul les caractères précédents le caractère fin de chaîne sont recopiés. Le tableau de caractères pointé par ***destination*** doit être suffisamment grand pour accueillir la chaîne concaténée. Le pointeur ***destination*** est retourné.

5. Comparaison de chaînes `strcmp, strncmp`

```
Prototype dans <string.h>
int strcmp ( const char * str1, const char * str2 );
```

Compare les chaînes ***str1*** et ***str2***. Cette fonction débute par la comparaison du premier caractère de chaque chaîne. S'ils sont égaux, elle continue avec la paire de caractères suivants tant que les caractères sont égaux et qu'un caractère fin de chaîne n'est pas atteint. Cette fonction retourne un entier indiquant la relation d'ordre entre les deux chaînes :

Un zéro signifie que les deux chaînes sont égales
Une valeur positive indique que le premier caractère différent a une plus grande valeur dans ***str1*** que dans ***str2***
Une valeur négative indique l'inverse.

```
Prototype dans <string.h>
int strncmp ( const char * str1, const char * str2, size_t num );
```

Cette fonction débute par la comparaison du premier caractère de chaque chaîne. S'ils sont égaux, elle continue avec la paire de caractères suivantes tant que les caractères sont égaux et qu'un caractère fin de chaîne n'est pas atteint et que ***num*** caractères n'ont pas été comparés. Cette fonction retourne un entier indiquant la relation d'ordre entre les deux chaînes :

Un zéro signifie que les ***num*** premiers caractères des deux chaînes sont égaux.
Une valeur positive indique que le premier caractère différent a une plus grande valeur dans ***str1*** que dans ***str2***
Une valeur négative indique l'inverse.

6. Longueur d'une chaîne `strlen`

```
Prototype dans <string.h>
size_t strlen ( const char * str );
```

Retourne un entier égal à la longuer de la chaîne ***str***.
La longueur d'une chaîne correspond au nombre de caractère entre le début de la chaîne et le caractère fin de chaîne `'\0'`. A ne pas confondre avec la taille du tableau, par exemple :

```
char MaChaine[100]="chaine test";
```

definit un tableau de **100** caractères mais la chaîne **MaChaine** a une longuer de **11** caractères. Par conséquent **sizeof(MaChaine)** retourne **100**, et **strlen(MaChaine)** retourne **11**.

7. Recherche d'une chaîne dans une autre `strstr`

```
Prototype dans <string.h>
char * strstr ( const char * str1, const char * str2);
```

Retourne un pointeur sur la première occurrence de la chaîne ***str2*** dans la chaine ***str1***, ou un pointeur NULL si ***str2*** n'est pas une partie de ***str1***. La comparaison ne porte pas sur le caractère fin de chaîne.

8. Test de caractères `isaplha,isalnum, islower, isupper`

prototype dans <ctype.h>

int isalpha (int c); retourne 0 si **c** n'est pas un caractère alphabétique
int isalnum (int c); retourne 0 si **c** n'est pas un caractère alphanumérique
int islower (int c); retourne 0 si **c** n'est pas un caractère minuscule
int isupper (int c); retourne 0 si **c** n'est pas un caractère majuscule

9. Recherche de caractères dans une chaîne `strchr,strbrk`

```
Prototype dans <string.h>
char * strchr(char * str, int character);
```

Retourne un pointeur sur la première occurrence de *character* dans la chaîne *str*.
Le caractère fin de chaîne est considéré, par conséquent cette fonction peut être utilisée pour localiser la fin d'une chaîne.

```
Prototype dans <string.h>
char * strbrk(char * str, int character);
```

Retourne un pointeur sur la première occurrence dans *str1* d'un des caractères de *str2*, ou un pointeur NULL si aucun caractère ne s'y trouve. La recherche ne comprend pas le caractère de fin de chaîne.

Exemple :

```
/* strpbrk exemple d'utilisation, extraction des voyelles */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "This is a sample string";
  char key[] = "aeiou";
  char * pch;
  printf ("Vowels in '%s': ",str);
  pch = strpbrk (str, key);
  while (pch != NULL)
  {
    printf ("%c " , *pch);
    pch = strpbrk (pch+1,key);
  }
  printf ("\n");
  return 0;
}
```

fournit l'affichage suivant :

```
Vowels in 'This is a sample string': i i a a e i
```

10. Extraction de chaîne `strtok`

```
Prototype dans <string.h>
char * strtok(char * str, const char * delimiters);
```

Une séquence d'appel de cette fonction sépare la chaîne ***str*** en plusieurs sous chaînes.

Lors du premier appel la fonction attend en paramètre une chaîne de caractères comme argument ***str***, dont le premier caractère est utilisé pour débuter la recherche des jetons présents dans la chaîne ***delimiters***.

Lors des appels suivants, la fonction attend un pointeur **NULL** et utilise la position à droite de la dernière sous chaîne comme nouveau point de départ pour la recherche.

Pour déterminer le début et la fin d'une sous chaîne la fonction recherche du point de départ la première occurrence d'un caractère qui n'est pas contenu dans la chaîne ***delimiters***. Puis démarrant de ce début de chaîne elle recherche la prochaine occurrence dans ***str*** d'un caractère de la chaîne ***delimiters***, ce caractère devient la fin de la sous chaîne.

Cette fin de sous chaîne est automatiquement remplacée par un caractère fin de chaîne et le début de la sous chaîne suivante est retourné.

Lorsque le caractère fin de chaîne de ***str*** a été trouvé lors d'un appel à ***strtok***, tous les appels successifs à cette fonction avec un pointeur **NULL** comme premier argument retourne un pointeur **NULL**.

```
/* strtok extraction de sous chaînes */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Separation de chaine \"%s\" en sous chaine:\n",str);
  pch = strtok (str," ,.-");
            //les délimiteurs sont espace, virgule, point et tiret
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

Affichage écran :

```
Separation de chaine "- This, a sample string." en sous
chaine:
This
a
sample
string
```

11. Duplication de chaîne `strdup`

```
Prototype dans <string.h>
char *strdup(const char *s1)
```

Duplique la chaîne ***s1***, la mémoire allouée doit être libérée par l'utilisateur (fonction **free**).

Le pointeur de retour spécifie l'emplacement réservé pour la nouvelle chaîne, il vaut **NULL** en cas d'échec.
Fonction non standard ANSI. La chaîne doit être libérée par l'utilisateur ensuite.

Exemple :

```
int main(void)
{ char s1[]= "La phrase initiale";
  char * s2=NULL;

s2=strdup(s1);
strcpy(s1, "nouveau",);
printf("%s",s1);
printf("%s",s2);
free(s2); // libération de la mémoire allouée par strdup
}
```

Affichage écran:

```
nouveau
La phrase initiale
```

Chapitre 5 : Les Fonctions

1. Définition d'une Fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme :

```
type nom-fonction (type_1 arg_1,...,type_n arg_n)
{déclarations de variables locales
  liste d'instructions
}
```

La première ligne de cette définition est l'*en-tête* de la fonction. Dans cet en-tête, ***type*** désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef **`void`**.
Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction.
Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef **`void`**.
Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, **`return`**, dont la syntaxe est

`return(`*expression***`);`**

La valeur de **`expression`** est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type **`void`**), sa définition s'achève par

`return;`

Plusieurs instructions **`return`** peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier **`return`** rencontré lors de l'exécution.

1.1. Déclaration des paramètres formels d'une fonction

Trois types de paramètres formels se dégagent :

Les paramètres d'entrée,
on appelle ***un paramètre d'entrée*** une variable qui est fournie à la fonction par le module appelant. La fonction n'accède qu'en lecture à la variable.

Les paramètres de sortie,
on appelle ***un paramètre de sortie*** une variable qui est fournie par la fonction au module appelant.

Les paramètres d'entrée-sortie,
on appelle ***un paramètre d'entrée/sortie*** une variable qui est fournie à la fonction par le module appelant. Cette variable est accessible en lecture/écriture par la fonction.

1.2. Pour les paramètres d'entrée :

1.2.1. On passe la valeur du paramètre (passage d'une variable float, int, char, double, long,...), il s'agit d'une recopie de la variable

OU

1.2.2. On passe un pointeur sur constante qui contient l'adresse de la variable (passage de l'adresse d'une variable ,...), on accède à la variable en lecture uniquement.

1.3. Pour les paramètres de sortie

On retourne une variable, utilisation du return. La variable est recopiée dans le paramètre effectif de retour.

1.4. Pour les paramètres d'entrée/sortie :

On passe un pointeur qui contient la valeur de la variable (passage de l'adresse d'une variable,...), la variable est accessible en lecture et en écriture depuis la fonction.

2. Exemple d'utilisation de fonctions

2.1. Fonction ayant 3 paramètres d'entrée et un paramètre de sortie

La fonction **`equation1`** retourne une variable de type **`float`** contenant la somme de trois variables de type **`float`** passées en entrée.

```
/**** déclaration d'une fonction <=> prototype des fonctions ***********/
/* fonction ayant 3 paramètres d'entrée et un paramètre de sortie

float equation1(float , float , float );

/* Code du module appelant */

void main(void)
{float alpha=2, beta=5;
float x=3,w;
double y,z;
// Appel de la fonction equation 1 qui assure le calcul de y=a*x+b
// les valeurs de x, alpha et beta sont recopiés dans a,x et b
y=equation1(x,alpha,beta);

w=equation1(2,5,8); // autant d'appel que voulu
}

float equation1(float a, float x, float b)
{
return(a*x+b);
}
```

2.2. Fonction ayant 1 paramètre d'entrée/ sortie

La fonction **`compte`** incrémente une variable de type **`int`** passée en entrée, cette variable est modifiée par la fonction.

```
/**** declaration d'un fonction <=> prototype des fonctions ***********/
void compte(int * );
/* la fonction compte reçoit le pointeur sur une variable de type entier, cette
varaible sera accessible en lecture et en écriture */

/* Code du module appelant */
void main(void)
{ int i=0;
compte(&i); // appel de la fonction, notons l'opérateur & devant la varibale i
printf("i=%d",i);
compte(&i);
printf(" i=%d, ",i);
}
/* définition de la fonction compte */
void compte(int * pj)
{
*pj=*pj+1; // notons l'opérateur * qui permet l'acces à la variable de type int
           //située à l'adresse pj
}
```

Ce programme affiche : **i=1, i=2,**

2.3 Fonction ayant 2 paramètres d'entrée et 2 paramètres de sortie

La fonction **equation2** calcule et fournit le module et l'argument de deux **float** passés en entrée.

```
#define pi 3.14157
/**** declaration d'un fonction <=> prototype des fonctions ***********/

void equation2(float , float , double *, double *);

/* la fonction equation2 ne renvoie pas de paramètres (void devant).
Elle reçoit 2 float et 2 adresses de variables de type double
ces 2 adresses vont permettre d'accéder en lecture et en écriture aux variables
pointées */

/* Code du module appelant */

void main(void)
{
float w,x;
double z,y;
w=2;
x=2;
equation2(x,w,&z,&y); // Notons l'opérateur & devant z et y
}

/* définition de la fonction equation2 */

void equation2(float x, float y, double *pModule, double *pArgument)
{
*pMmodule=sqrt(x*x+y*y);
// la variable se trouvant à l'adresse pMmodule ppv sqrt(x*x+y*y)
// la variable se trouvant à l'adresse pArgument ppv l'argument
if (x>0) *pArgument=atan(y/x);
if (x<0) *pArgument=atan(y/x)+pi; /* ajout de pi */
if (x==0) { if (y>0) *pArgument=pi/2;
```

```
if (y<0) *pArgument=-pi/2;
if (y==0) *pArgument=0;
}
}
```

2.4. Fonction ayant un tableau comme paramètre d'entrée et un paramètre de retour

La fonction **somme** calcule la somme des éléments d'un tableau de **float** fournit en entrée. Notons qu'un tableau sera toujours fournit avec le nombre d'éléments auquel la fonction doit accéder.

```
/**** declaration d'un fonction <=> prototype des fonctions ***********/

float somme(int ,const float *);

/* fonction somme retourne un float (paramètre de retour) et reçoit la valeur
d'un int (paramètre d'entrée) et un tableau en entrée (pointeur sur une
constante). On passe juste l'adresse de base du tableau, on ne recopie pas tout
le tableau, par contre on précise que les éléments du tableau ne devront pas
être modifiés dans la fonction.*/

/* Code du module appelant */

#define Taille 5

void main(void)
{
float tab[Taille];
int NbElements;
float w;
NbElements=3;
tab[0]=5;
tab[1]=8;
tab[2]=6
w=somme(NbElements,tab); //rappel tab est l'adresse du 1er element, (&tab[0])
}

/* définition de la fonction somme qui assure la somme des nb elements du
tableau*/

float somme(int nb,const float *ptab)
// le mot-clef const interdit l'écriture dans le tableau
{
// declaration des variables locales à la fonction, accessibles depuis la
// fonction uniquement

float val=0;
int i;
for (i=0;i<nb;i++) val=val+ptab[i]; /* <=> val=val+*(ptab+i) */
return(val);

}
```

2.5. Fonction ayant un tableau comme paramètre d'entrée/sortie

La fonction **initialisation** assure la saisie d'un tableau de **float.** L'adresse de base du tableau et le nombre

d'élément à saisir sont fournit en argument. Les éléments du tableau seront accessibles en écriture.

```
/* prototype de la fonction */

void initialisation(int , float *)

/* Code du module appelant */

#define taille 5

void main(void)
{ float tab[taille];
int nbelements;
float w;
nbelements=3;
initialisation(nbelements,tab); // la fonction va modifier 3 elements
}

/* Code de la fonction */

void initialisation(int n, float *ptab)

/* Reçoit l'adresse du tableau passe en argument, n precise le nombre
d'éléments du tableau*/

{ int i;
for (i=0;i<n;i++)
{ printf("\n Entrez la valeur de tab[%d] :",i);
fflush(stdin);
scanf("%f",&ptab[i]);   /* ou alors scanf("%f",ptab+i);
                           le pointeur ptab permet l'accès en écriture */
}
}
```

2.6. Fonction ayant une variable structurée comme paramètre d'entrée/sortie

La fonction **saisie_individu** assure la saisie d'une variable de type **Tindividu** (type strucutré).

```
#define longchaine 26

/* défintion d'un type structuré */

typedef struct {
char nom[longchaine];
int age;
} Tindividu;

/* prototype de la fonction */

/*fonction qui remplit une variable structurée, paramètre d'entrée/sortie */
void saisie_individu(Tindividu *);
// la structure sera modifiée on passe l'adresse
```

```
/* Code du module appelant */

void main(void)
{
Tindividu MonBonhomme={"",0}; // initialisé avec 1 chaîne vide et age à 0
saisie_individu(&MonBonhomme); // on passe l'adresse
}

/* Code de la fonction */

void saisie_individu(t_individu * pUnBonhomme)
{int longueur=0;
printf(" Son nom ? : ");
/* saisie du champ nom */
fflush(stdin);
fgets(pUnBonhomme->nom,longchaine,stdin); // acces au champ avec ->
      longueur= strlen(pUnBonhomme->nom);
      if (pUnBonhomme->nom[longueur-1]=='\n')
      {     longueur--;
            pUnBonhomme->nom[longueur]='\0';
      }

printf("\n Son age ? : ");
fflush(stdin);
scanf("%d",&(pUnBonhomme->age)); // l'adresse du champ age

}
```

2.7. Fonction ayant une variable structurée comme paramètre d'entrée

Dans ce cas il es possible de passer la valeur de la structure (recopie de la structure dans un paramètre formel de la fonction) mais cette recopie peut être gourmande, on préfère ici passer un pointeur sur constante qui permet uniquement l'accès en lecture. La fonction **affiche_individu** assure l'affichage à l'écran d'une variable de type **Tindividu** (type structuré).

```
#define longchaine 26

/* défintion d'un type structuré */

typedef struct {
char nom[longchaine];
int age;
} Tindividu;

/* prototype de la fonction */

void affiche_individu(const Tindividu * );
/*fonction qui affiche une variable structurée, paramètre d'entrée, passage
par adresse le mot clef const prohibe l'accès en écriture
*/

/* Code du module appelant */

void main(void)
{
Tindividu MonBonhomme={"Guevara",42};
```

```
affiche_individu(&MonBonhomme); // on passe l'adresse
}

/* Code de la fonction */

void affiche_individu(const Tindividu * pUnBonhomme)
{
printf(" Son nom : %s \n",pUnBonhomme->nom);
printf(" Son age : %d \n",pUnBonhomme->prenom);
}
```

3. Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même *durée de vie*. On distingue deux catégories de variables.

3.1. Les variables permanentes (ou statiques)

Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef **`static`**.

3.2. Les variables temporaires

Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, **`auto`**, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type **`register`**. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur, il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

La durée de vie des variables est liée à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

3.3. Variables globales

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, n'est une variable globale :

```
int n;
void fonction();

void fonction()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i <3; i++)
    fonction();
}
La variable n est initialisée à zéro par le compilateur et il s'agit d'une
variable permanente. En effet, le programme affiche
appel numero 1
appel numero 2
appel numero 3
```

3.4. Variables locales

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant

```
int n = 10;
void fonction();

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}


main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

affiche

```
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef **static** :

static *type nom-de-variable ;*

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, **n** est une variable locale à la fonction secondaire **fonction,** mais de classe statique.

```
int n = 10;
void fonction();

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i <3; i++)
    fonction();
}
```

Ce programme affiche

```
appel numero 1
appel numero 2
appel numero 3
```

On voit que la variable locale **n** est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

4. Les paramètres de la fonction main

Le prototype de la fonction main peut être sous la forme :

```
int main ( void); // pas de paramètres formels
int main ( int argc, char **argv); // avec paramètres formels
```

Dans les deux cas le paramètre de retour est un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur.

De manière optionnel la fonction peut accueillir des paramètres formels :

nb_arg :	correspond au nombre d'arguments reçus,
arg[0] :	chaîne de caractères qui comprend le chemin et le nom du logiciel,
arg[i] :	chaîne de caractères comprenant les paramètres.

Le programme ci-dessous présente un exemple d'exécution :

```
#include <stdio.h>


int main(int nb_arg, char **arg)
{

  int i;

   /* affichage des arguments du module main*/

   for (i=0;i<nb_arg;i++)   printf(" arg[%d] = %s \n", i,arg[i]);

 return 0;  /* ce programme renvoie un 0, au systeme appelant */
}
```

Execution :
Depuis une ligne de commande on entre :
>exemp20 Guevara Castro Allende

et on obtient :

```
arg[0]=d:\cours_c\exemp20.exe
arg[1]= Guevara
arg[2]= Castro
arg[3]= Allende
```

la première chaîne correspond au nom de l'exécutable et à sa localisation sur le disque dur (ici **exemp20.exe** qui se trouve sur le disque **d:** dans le répertoire **d:\cours_c**).

5. Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages (voir la fonction **qsort()**). Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction ayant pour prototype

type fonction(*type_1*, . . . ,*type_n*);

est de type

***type* (*)(*type_1*, . . . ,*type_n*);**

Dans l'exemple ci-dessous on montre comment utiliser cette fonctionnalité :

```
#include <stdio.h>


/* declaration d'un type fonction recevant 2 float retournant 1 float */

typedef float Tfonction(float,float);


/*** Prototype des fonctions    ********************************************/

float somme(float , float );
```

```
float produit(float ,float );
float division(float , float );

float operation(float , float , Tfonction );   // le dernier paramètre formel
                                               // est de type fonction

void main(void)
{
 float a,b,c;
 int choix=0;
 fonction *f;     /* f pointeur sur une fonction */

do{
printf("* Type d'operation choisi : 1. somme a+b \n");
printf("*                           2. produit a*b \n");
printf("*                           3. division a/b \n");
printf("*                           0. fin du programme \n");
printf("*                           Entrez votre choix : ");
fflush(stdin);
scanf("%d",&choix);

printf("Saisie de a : ");
fflush(stdin);
scanf(" %f",&a);
printf("Saisie de b : ");
fflush(stdin);
scanf(" %f",&b);

switch(choix)
{ case 1: f=&somme;  /* f ppv l'adresse de la fonction somme */
        break;
 case 2: f=&produit; /* f ppv l'adresse de la fonction produit */
       break;
 case 3: f=division;
            /* le nom d'une fonction est son adresse division <=>&division*/
       break;
 default : break;
}
if (0<choix&&choix<4)
{      c=f(a,b);
       printf(" resultat = %2.2f , appuyer sur Entree pour continuer...",c);
       fflush(stdin);
       getchar();
}
}  while (choix!=0);



/* on peut faire passer une fonction en paramètre */

c=operation(a,b,division);  /* division est passee en parametre */
printf(" resultat de la division = %2.2f \n",c);

f=&produit;
c=operation(a,b,f);  /* f est passee en paramètre */
printf(" resultat de la multiplication = %2.2f \n",c);
}

float somme(float x, float y)
{return(x+y);}
```

```
float produit(float x,float y)
{return(x*y);}

float division(float x, float y)
{ if (y==0) { if (x!=0) {printf("division par zero impossible\n");
                         exit(1);
                        }
           else return(1);
         }
 else return(x/y);
}



float operation(float x, float y, fonction NomLocalFonction)

{ return(NomLocalFonction(x,y)); }

/* NomLocalFonction est un pointeur, puisque le nom de la fonction est passé en
paramètre et que le nom d'une fonction est sont adresse !!*****/
```

6. Fonctions avec un nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre variable de paramètres. En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème : toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions **printf** et **scanf**.

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation **...** (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype

```
int f(int a, char c, ...);
```

prend comme paramètre un entier, un caractère et un nombre quelconque d'autres paramètres. De même le prototype de la fonction **printf** est

```
int printf(char *format, ...);
```

puisque **printf** a pour argument une chaîne de caractères spécifiant le format des données à imprimer, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête **stdarg.h** de la librairie standard. Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel ; cette variable a pour type **va_list**. Par exemple,

```
va_list liste_parametres;
```

Cette variable est tout d'abord initialisée à l'aide de la macro **va_star**t, dont la syntaxe est

```
va_start(liste_parametres, dernier_parametre);
```

Où **dernier_parametre** désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la **va_end** :

```
va_end(liste_parametres);
```

On accède aux différents paramètres de liste par la macro **va_arg** qui retourne le paramètre suivant de la liste:

va_arg(liste_parametres, type)

où **type** est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme suivant, où la fonction **affichage** effectue l'affichage des chaînes passées en argument en nombre quelconque.

```
#include <stdio.h>
#include <stdarg.h>          /* pour la gestion des parametres */
#define longueur 9

/* Prototype d'une fonction recevant des paramètres en nombre inconnu,.. */
/* nb correspond a ce nombre de paramètres */

void affichage(int nb,...);


void main(void)
{
      affichage(3,"Che","Fidel","Salvador");
      affichage(2,"Guy","Emile");
}


void affichage(int nb,...)
/* fonction affichant les chaines passees en parametre */
{   int status,i,j;
      char *ch;         /* pointeur sur une suite de caracteres */
      char nom[longueur];
      va_list pointe;   /* pointeur sur une suite de caracteres */

/*****   va_list type predefini dans stdarg.h :typedef char *va_list *****/

      va_start(pointe,nb);
      /**** va_start renvoie dans pointe (de type va_liste)
       l'adresse de l'element suivant nb passe en argument*/

   for (i=0;i<nb;i++)
      {
      ch=va_arg(pointe,char *);
/* ch ppv l'adresse de la chaine de caracteres a l'adresse pointe et pointe ppv
l'adresse du parametre suivant*/

/* on recopie la chaine debutant en ch dans nom, on ne connait pas a priori
 la taille de ch on ne peut donc utiliser strcpy() */
      j=0;
      do
      { nom[j]=ch[j];
        j++;
      }
      while(ch[j]!='\0' && j<(longueur-1));  // nom[] est de longueur 9

       nom[j]='\0'; // on ajoute le caractere de fin de chaine
```

```
    printf(" affichage des parametres : %s \n",nom);
    }  /* pour le for */

  va_end(pointe); /* rend la memoire eventuellement reservee par va_start */

  }  /* pour la fonction */
```

Affichage écran résultant :

```
affichage des parametres : Che
affichage des parametres : Fidel
affichage des parametres : Salvador
affichage des parametres : Guy
affichage des parametres : Emile
```

Chapitre 6 : Les Fichiers

Il est possible de lire et d'écrire des données dans un fichier. Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (***buffer***), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture) ...

Ces informations sont rassemblées dans une structure dont le type, **`FILE *`**, est défini dans **`stdio.h`**. Un objet de type **`FILE *`** est appelé *flot de données* (en anglais, stream).

Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande **`fopen`**. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction **`fclose.`**

1. La fonction `fopen`

```
Prototype dans <stdio.h>
FILE * fopen ( const char * filename, const char * mode );
```

Cette fonction, de type **`FILE*`** ouvre un fichier et lui associe un flot de données.

La valeur retournée par **`fopen`** est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur **`NULL`**. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction **`fopen`** est égale à **`NULL`** afin de détecter les erreurs (lecture d'un fichier inexistant...).

`filename` est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères.

`mode`, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les modes d'accès diffèrent suivant le type de fichier considéré. On distingue

- les *fichiers textes*, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture ;
- les *fichiers binaires*, pour lesquels les caractères de contrôle se sont pas interprétés.

Les différents modes d'accès sont les suivants :

`"r"`	ouverture d'un fichier texte en lecture
`"w"`	ouverture d'un fichier texte en écriture
`"a"`	ouverture d'un fichier texte en écriture à la fin
`"rb"`	ouverture d'un fichier binaire en lecture
`"wb"`	ouverture d'un fichier binaire en écriture
`"ab"`	ouverture d'un fichier binaire en écriture à la fin
`"r+"`	ouverture d'un fichier texte en lecture/écriture
`"w+"`	ouverture d'un fichier texte en lecture/écriture
`"a+"`	ouverture d'un fichier texte en lecture/écriture à la fin
`"r+b"`	ouverture d'un fichier binaire en lecture/écriture
`"w+b"`	ouverture d'un fichier binaire en lecture/écriture
`"a+b"`	ouverture d'un fichier binaire en lecture/écriture à la fin

Ces modes d'accès ont pour particularités :

- Si le mode contient la lettre **r**, le fichier doit exister.
- Si le mode contient la lettre **w**, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre **a**, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- **stdin** (standard input) : unité d'entrée (par défaut, le clavier) ;
- **stdout** (standard output) : unité de sortie (par défaut, l'écran) ;
- **stderr** (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

2. La fonction `fclose`

```
Prototype dans <stdio.h>
int fclose ( FILE * stream );
```

stream est le flot de type **FILE*** retourné par la fonction **fopen** correspondant.

La fonction **fclose** retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

3. La fonction `fread`

```
Prototype dans <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Lit un tableau de ***n*** éléments , ayant chacun une taille de ***size*** octets, depuis bytes le flux ***stream*** et stocke ces éléments à l'adresse spécifiée par *ptr*. Le pointeur de fichier ***stream*** est ensuite avancé du nombre d'octest lus qui en cas de succès est égal à (***size * n***).
Retourne le nombre d'éléments lus, si cette valeur est différente de n, soit une erreur de lecture est survenue soit le fin du fichier est atteinte.

4. La fonction `fwrite`

```
Prototype dans <stdio.h>
size_t fwrite(const void * ptr, size_t size, size_t n, FILE *
stream);
```

Ecrit un tableau de ***n*** éléments, chacun d'une taille de ***size*** octets, depuis l'adresse pointée par ***ptr*** vers la position courante du flux ***stream***. Le flux est avancé du nombre total d'octets écrits qui est égal à (***size * count***).
Retourne le nombre d'éléments effectivement écrits, s'il diffère de ***n***, cela indique une erreur d'écriture.

5. La fonction `feof`

```
Prototype dans <stdio.h>
int feof( FILE * stream);
```

Vérifie si l'indicateur de fin de fichier associé au flux ***stream*** est actif. Elle retourne une valeur différent de 0 le cas

échéant.

6. La fonction `fseek`

```
Prototype dans <stdio.h>
int fseek ( FILE * stream, long int offset, int origin);
```

Place le pointeur de fichier ***stream*** à une nouvelle position définie en ajoutant ***offset*** à une position de référence spécifiée par ***origin***. Elle retourne 0 en cas de succès et différent de 0 en cas d'échec.

Pour les flux ouverts en lecture/écriture l'appel à la fonction **fseek** permet de passer d'un mode à l'autre.

origin peut prendre les valeurs suivantes :

SEEK_SET	Début du fichier
SEEK_CUR	Position courante du pointeur de fichier
SEEK_END	Fin du fichier

7. La fonction `ftell`

```
Prototype dans <stdio.h>
long int ftell ( FILE * stream);
```

Retourne la valeur courante du pointeur de fichier ***stream***.
Pour les fichiers binaires la valeur retournée correspond au nombre d'octets depuis le début du fichier.
Pour les fichiers texte, la valeur n'est pas forcément l'exact nombre d'octets depuis le début du fichier, mais cette valeur peut être utilisée pour indiquer une position à la fonction ***fseek***.
En cas de succès la position courante est retournée.

8. La fonction `rewind`

```
Prototype dans <stdio.h>
void rewind(FILE * stream);
```

Place le pointeur en début de fichier.
Un appel à **rewind** est équivalent à **fseek(stream,0,SEEK_SET);**
Pour les flux ouverts en lecture/écriture l'appel à la fonction **rewind** permet de passer d'un mode à l'autre.

9. La fonction `fprintf`

```
Prototype dans <stdio.h>
int fprintf ( FILE * stream, const char * format, ... );
```

Ecrit dans ***stream*** des données formatées tel que spécifié dans le paramètre ***format***. Retourne le nombre de caractères écrits en cas de succès, et -1 en cas d'erreur. Les formats admis correspondent à ceux de la fonction **printf**.

10. Exemple chargement d'un fichier en mémoire

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

```
int main (void) {
  FILE * pFile;
  long lSize;
  char * buffer;
  size_t result;

  pFile = fopen ( "lenomdufichier.bin" , "rb" );
  if (pFile==NULL) {printf("Erreur lors de l'ouverture.\n"); exit (1);}

  // Détermination de la taille du fichier
  fseek (pFile , 0 , SEEK_END); // on se place à la fin
  lSize = ftell (pFile); // on récupère la taille
  rewind (pFile); // on revient au début

  // allocation mémoire pour stocker l'ensemble du fichier
  buffer = (char*) malloc (sizeof(char)*lSize);
  if (buffer == NULL) {printf("Erreur d'allocation memoire"); exit (2);}

  // copie du fichier dans le buffer
  result = fread (buffer,1,lSize,pFile);
  if (result!=lSize) {printf("Erreur de lecture"); exit (3);}

  /* l'ensemble du fichier est maintenant chargé en mémoire. */

  // on ferme le fichier
  fclose (pFile);
  free (buffer); // on rend la memoire allouee
  return 0;
}
```

11. Exemple Lecture et extraction de lignes d'un fichier texte

```
/* Ouverture d'un fichier texte chargement en mémoire
 puis extraction ligne par ligne à l'aide de strtok
 */

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



int main(void)
{
long taille=0;
char * pChaine=NULL;
FILE * pFichier;
char * p=NULL;

// Chargement d'un fichier texte dans un tableau de caractères


      pFichier=fopen("lenom.txt","rb");
     if (pFichier==NULL)
     {
     printf("Erreur lors de l'ouverture du fichier \n");
```

```
      exit(1);
      }

      fseek(pFichier,0,SEEK_END);
      taille=ftell(pFichier); // on récupère la taille du fichier
      rewind(pFichier); // on revient au début

      pChaine=(char *) realloc(NULL,taille*sizeof(char));
      if (pChaine==NULL)
      {
      printf("Erreur Allocation tableau pChaine \n");
      exit(2);
      }

      fread(pChaine,taille,1,pFichier);
      fclose(pFichier);

      // extraction des lignes une à une
       p=strtok(pChaine,"\n"); // p pointe sur le début de ligne.


       while(p!=NULL)
       {
             printf("%s\n",p); // affichage de la ligne
             p=strtok(NULL,"\n"); // on passe à la suivante
       }
   free(pChaine);

}
```

Chapitre 7 : Allocation Dynamique

La connaissance a priori de la taille des tableaux et des données nécessaires à l'exécution d'un programme n'est pas toujours possible. Pour permettre la création de données en fonction du besoin il existe des instructions qui permettent d'allouer de la mémoire au programme.

1. Allocation simple `malloc`

```
Prototype stdlib.h
void * malloc ( size_t size );
```

Alloue un bloc de ***size*** octets de mémoire, et retourne un pointeur de type **`void *`** indiquant l'adresse de début du bloc. Le contenu du bloc mémoire est indéterminé. En cas d'echec le pointeur retourné vaut **`NULL`**.

Exemple allocation d'un tableau de `float`

```
int main(void)
{     float * tab=NULL;
      int taille=0;
printf(" taille du tableau désiré ?")
fflush(stdin);
scanf("%d",&taille);
tab=(float ) malloc(sizeof(float)*taille);
if (tab==NULL)
{     printf(" Echec allocation");
      exit(1);
}

for (i=0;i<taille);i++)
{
tab[i]=i*10;
}
for (i=0;i<taille);i++) printf("%f \n",tab[i]);
free(tab);
}
```

2. Allocation multiple `realloc`

```
Prototype stdlib.h
```

```
void * realloc ( void * ptr, size_t size);
```

Réalloue de la mémoire. La taille du bloc mémoire pointé par ***ptr*** est redimensionné à ***size*** octets, augmentant ou réduisant la bloc existant. Si besoin la fonction déplace le bloc vers un nouvel emplacement. Le paramètre de retour correspond à l'adresse de base du nouveau bloc (pointeur de type ***void ***). Le contenu du bloc mémoire est maintenu. Si la nouvelle taille ***size*** est supérieure à l'ancienne, la valeur du nouveau bloc est indéterminée.

Si ***ptr*** vaut **NULL**, la fonction se comporte comme la fonction **malloc**. Si ***size*** vaut 0, la mémoire précédemment allouée est rendue, la fonction se comporte alors comme **free**. En cas d'échec le pointeur **NULL** est retourné.

Exemple utilisation de `realloc` :

```
#include <stdio.h>              /* pour scanf, printf */
#include <stdlib.h>    /* pour exit, calloc, malloc, realloc, free */

void main(void)
{

float *vect=NULL;   /* pointeur sur une suite de réels, initialisé à NULL  */


int i,j,k,nb;

k=3;

// reservation de k éléments

vect=(float *) realloc(vect,k*sizeof(float));
if (vect==NULL)
{
printf(" problème allocation memoire \n");
exit(1);
}

// on remplit le tableau

for (j=0;j<k;j++)  vect[j]=j;

// on l'affiche
printf("1. adresse du tableau = %p et adresse du pointeur vect %p \n"
,vect,&vect);
for (j=0;j<k;j++)
printf("vect[%d]=%f et son adresse %p \n", j, vect[j], &vect[j]);


// reservation de 2 elements supplémentaires

i=2;
nb=k+i;
vect=(float *) realloc(vect,nb*sizeof(float));

if (vect==NULL)
{
printf(" problème allocation memoire \n");
exit(1);
}

// on remplit le tableau

for (j=k;j<nb;j++)  vect[j]=j;
```

```
// on l'affiche

printf("2. adresse du tableau = %p et adresse du pointeur vect %p \n"
,vect,&vect);
for (j=0;j<nb;j++)
printf("vect[%d]=%f et son adresse %p \n", j, vect[j], &vect[j]);

// on libère l'espace mémoire

free(vect);
```
}

AFFCIHAGE ECRAN

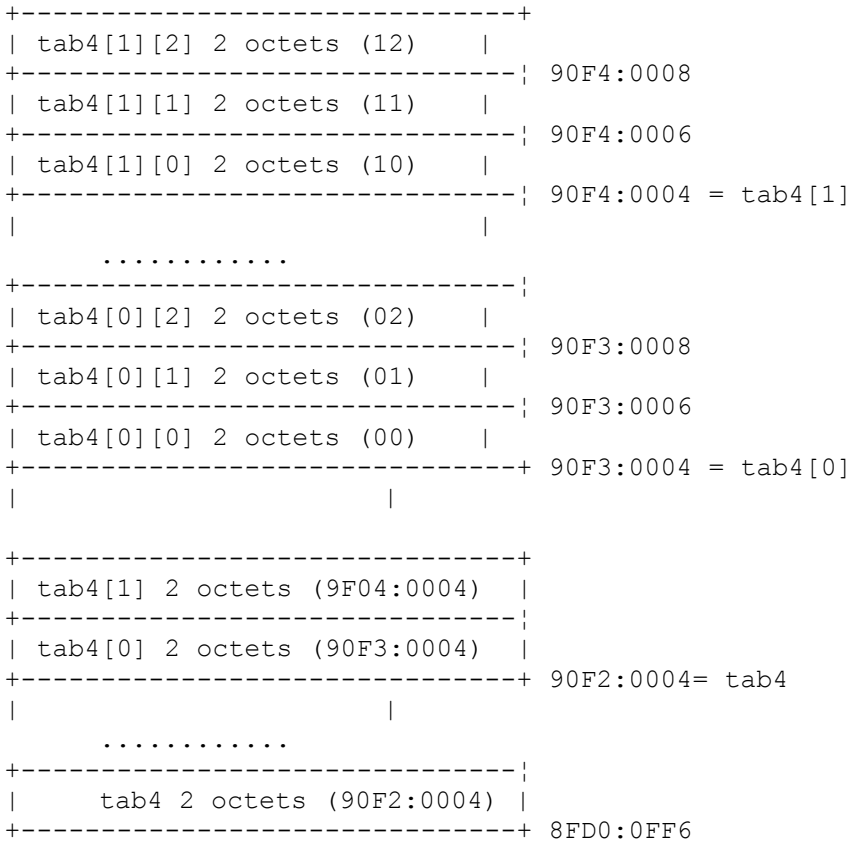
```
1. adresse du tableau = 90F2:0004 et adresse du pointeur vect 8FD0:0FF8
vect[0]=0.000000 et son adresse 90F2:0004
vect[1]=1.000000 et son adresse 90F2:0008
vect[2]=2.000000 et son adresse 90F2:000C
2. adresse du tableau = 90F3:0004 et adresse du pointeur vect 8FD0:0FF8
vect[0]=0.000000 et son adresse 90F3:0004
vect[1]=1.000000 et son adresse 90F3:0008
vect[2]=2.000000 et son adresse 90F3:000C
vect[3]=3.000000 et son adresse 90F3:0010
vect[4]=4.000000 et son adresse 90F3:0014
```

Exemple allocation dynamique de tableau à plusieurs dimension :

```
#include <stdio.h>                  // pour scanf, puts, printf
#include <stdlib.h>                 // pour l'allocation dynamique

void main(void)
{

int **tab4=NULL;
unsigned int i,j,k,n;

// tableau dont on ignore le nombre de lignes ET le nombre de colonnes a
// priori

printf(" Donnez le nombre de lignes du tableau : ");
fflush(stdin);
scanf("%u",&k);
printf(" Donnez le nombre de colonnes du tableau:");
fflush(stdin);
scanf("%u",&n);

// allocation dynamique d'un tableau de k pointeurs pointant sur des pointeurs
// pointant sur des entiers

 tab4=(int **)  realloc(tab4,k*sizeof(int *));

if (tab4==NULL) {printf("Memoire insuffisante...");
      exit(1);}

// allocation dynamique d'autant de tableau de n entiers qu'il y a de
//ligne, l'adresse est mise dans tab4[i]

for (i=0;i<k;i++)
```

```
{
tab4[i]=(int *) realloc(NULL,n*sizeof(int));
if (tab4[i]==NULL) {printf("Memoire insuffisante...");
               exit(2);}
}

// on remplit tab4
for (i=0;i<k;i++)
for (j=0;j<n;j++) tab4[i][j]=i*10+j;


// on rend l'espace mémoire devenu inutile
// les blocs d'entiers

for (i=0;i<k;i++) free(tab4[i]);

// le tableau de pointeurs

free(tab4);

}
```

Schéma mémoire du tableau à taille a priori inconnue
le pointeur tab4 est dans la pile, il pointe sur un tableau de pointeurs
les éléments tab4[i] qui sont dans le TAS contiennent
l'adresse des différentes lignes du tableau, qui sont des blocs d'entiers rangés de manière contigue dans le TAS



Chapitre 8 : Listes Chaînées