COURS LANGAGE C

MICHEL LANDSCHOOT

CHAPITRE 1 : ELEMENTS DE BASE

CHAPITRE 2 : INSTRUCTIONS

CHAPITRE 3 : OPERATEURS

CHAPITRE 4 : POINTEURS

CHAPITRE 5 : TYPES DÉRIVÉS

CHAPITRE 6 : FONCTIONS

CHAPITRE 13 : LES FICHIERS

CHAPITRE 14 : OBTENTION D' UN EXECUTABLE

CHAPITRE 15 : LA LIBRAIRIE STANDARD

CHAPITRE 16 : LA PROGRAMMATION MODULAIRE

ANNEXES

1 CODAGE BINAIRE

2 OPERATEURS BIT A BIT

3 BIBLIOGRAPHIE

CHAPITRE I : ELEMENTS DE BASE

1 LES COMMENTAIRES

/* commentaires */	/* commentaires */	/* * commentaires * commentaires */

Un commentaire peut s'étendre sur plusieurs lignes.

Le premier */ rencontré ferme tous les /* précédents. Les commentaires imbriqués ne sont donc pas autorisés, sauf option de compilation activée. Mais dans ce cas la portabilité des sources n'est pas garantie.

2 LES IDENTIFICATEURS

Constantes symboliques *en majuscules par convention*	**Variables typées**	**Déclarations protypes de fonctions**
#define **SIZE** 100	char **c**; int **i**; int **mois**;	extern int **f**(); extern void **g**(); extern int **h**(int); extern int **k**(int n);

règles lexicales :

- suite de caractères alphanumériques ou caractère surligné _
- premier caractère alphabétique ou caractère surligné _
- longueur des identificateurs dépendante du compilateur
- différenciation des majuscules et minuscules (case sensitive)

3 LES MOTS RESERVES (MOTS CLÉS)

Typologie des mots-clés :

Types	Classes d'allocation	Instructions de contrôle	Opérateurs	Etiquettes prédéfinies
char	auto	if	= + - / %	case
short	static	else	<< >>	default
int	extern	while	&& ||	
long	register	do	& |	
float		for	<<=	
double		switch	>>=	
signed		break	== +=	
unsigned		continue		
enum		goto	etc...	
struct		return		
union				
typedef				
void			sizeof	
const				
volatile				

volatile : programmation système

auto : obsolète, pour les variables locales à une fonction

register : obsolète, pour demander au compilateur de ranger une variable dans l'un des registres du processeur, si cela est possible. Les compilateurs modernes le décident seuls dorénavant.

4 LES TYPES SIMPLES

4.1 TYPE CARACTÈRE

Type	Taille	Valeurs	
char	**1 octet**	**signed char** : [-128,+127]	**unsigned char** : [0-255]

4.2 TYPES ENTIERS

types char, short, int, long, long long

signed (defaut) ou unsigned

les tailles sont "machine dépendante" et "système dépendant"

garantie : sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

Type	Taille	Valeurs	
short /short int	**>=2 octets**	**signed** [-32768, 32767]	**unsigned** [0, 65535]
int	**>= 2 octets**	taille d'un mot machine: le plus souvent 32 bits **signed** [-27147483648, 27147483647]	
long / long int	**>= 4 octets**		
long long	**>= 8 octets (2 mots machine)** 16 octets sur des architectures 64 bits		

4.3 TYPES REELS

Type	Taille	Valeurs	Chiffres significatifs
float	4 octets	+/- 3.4028210 +/-38	6
double	8 octets	+/-1.79769 10 +/-308	15
long double	8 ou 16 octets	+/- 1.18973 10 +/-4932	18

5 LES CONSTANTES

Une constante possède une valeur fixe non modifiable et un type particulier.

En C, une constante n'a pas d'identificateur, c'est une valeur littérale.

5.1 CONSTANTES ENTIERES

décimale : base 10

Une constante décimale est un **int** à condition qu'elle tienne dans un **int**, sinon un **long**.

octale : base 8 et commençant par un 0

hexadécimale: base 16 et commençant par 0x

Les lettres A, B, C, D, E, F ou a, b, c, d, e, f sont utilisées pour représenter successivement les nombres 10, 11, 12, 13, 14 et 15.

Les suffixes u ou U et l ou L spécifient explicitement des constantes respectivement non signées et de type long : 251293L, 45lu, 12u, 4565665L

Les suffixes ll ou LL pour long long.

5.2 CONSTANTES VIRGULE FLOTTANTE

Une constante en virgule flottante est de **type double par défaut**.

syntaxe : [+-][entier].[Ee][+-][entier][fFlL]

exemples : 1.23 .16 0.16 16 E – 2 2.564e6 1.6 e -1

suffixe **f ou F** pour float : 12.35f

suffixe **l ou L** pour long double

5.3 CONSTANTES CARACTERES

Une constante caractère est un entier particulier de **type char** et dépendant de la machine cible.

notation entre quotes : 'a' = 48 en ASCII 'a' = 240 en EBCDIC

Des caractères standards sont possibles :

'\b'	backspace
'\f'	saut de page
'\n'	passage à la ligne
'\r'	retour chariot
'\t'	tabulation horizontale
'\v'	tabulation verticale
'\\'	backslash
'\"	apostrophe
'\"'	guillemet
'\0'	entier 0 (les 8 bits à 0)
'\ooo'	nombre octal
'\xhhh'	nombre hexadécimal

5.4 CHAÎNES DE CARACTERES

Les chaînes de caractères sont des tableaux de caractères terminés par **le caractère nul \0**.

Pour poursuivre sur plusieurs lignes une chaîne, il faut mettre le **caractère** \ en fin de ligne.

```
Lands\
choot
```

attention : sizeof("Landschoot") == strlen("Landschoot") + 1

11 octets = 10 octets + 1 octet (le '\0')

5.5 LES CONSTANTES NOMMEES

Le mot-clé **const** rend un objet constant.

Exemples:

```
/* définition d'une constante x */
const int  x = 55;
x = 93;          /* erreur : modification du CONTENU */

/* définition d' un pointeur pc sur une chaîne de caractères constante */
const char * pc = "Landschoot";
pc[3] = 'N';             /* erreur : modification du CONTENU */
pc = "Michel";           /* ok : modification du CONTENANT */

/* définition d' un pointeur constant pc */
char * const pc = "Langage C";
pc[10] = pc[11] = 'P';          /* ok : modification du CONTENU */
pc = "Langage C++";             /* erreur : modification du CONTENANT */

/* définition d' un pointeur constant pc sur une chaîne de caractères constante */
const char * const pc = "Landschoot";
pc[3] = 'N';             /* erreur : modification du CONTENU */
pc = "Michel";           /* erreur : modification du CONTENANT */

/* déclaration prototype de la fonction de comparaison de chaînes de caractères */
int strcmp (const char * p1, const char * p2);
/* les données pointées par les pointeurs p1 et p2 restent inchangées : LE CONTENU */
/* les valeurs de p1 et p2 peuvent changées : LE CONTENANT */
```

6 ENUMERATIONS

Une énumération est un ensemble de constantes entières rangées par ordre croissant, à partir de 0 par défaut.

exemple :

enum motcle { ASM = 0, AUTO = 1, BREAK = 2};

ou avec des valeurs par défaut, celles de la suite des entiers naturels

enum motcle { ASM, AUTO, BREAK };

ce qui revient à écrire :

const int ASM = 0;

const int AUTO = 1;

const int BREAK = 2;

D'autres valeurs que celles par défaut peuvent être données.

exemple :

enum nombre { zero, un, deux, assez = 10, assez_un, beaucoup = 20};

ce qui revient à :

const int zero = 0;

const int un = 1;

const int deux = 2;

const int assez = 10;

const int assez_un = 11;

const int beaucoup = 20;

7 GAIN D'ESPACE MEMOIRE

7.1 CHAMPS DANS UNE STRUCTURE

Ils permettent d'optimiser la mémoire et de descendre au niveau du bit (programmations hardware et système).

exemple :

```
struct registre_etat {
        unsigned int autorise : 1;          /* 1 bit : 0 ou 1 */
        unsigned int numero_page : 3;       /* 3 bits : 0  à 7 */
        unsigned int acces : 1;             /* 1 bit : 0 ou 1 */
        unsigned : 1;                       /* inutilisé */
        unsigned int non_resident : 1;      /* 1 bit : 0 ou 1 */
};
```

Les champs sont obligatoirement de type entier, il n'est pas possible de prendre l'adresse d'un champ.

Les champs non nommés sont permis (pour caler sur le début d'un mot machine).

Attention : raccourci l'espace des données mais allonge le code pour manipuler de telles variables.

7.2 UNIONS

Une union regroupe des objets de type différents à une même adresse mémoire. La taille de l'union est celle du plus grand objet.

Les unions sont rarement utilisées.

exemple :

```
union adresse_stagiaire {
        struct hotel hot;
        struct particulier part;
};
```

où:

```
struct hotel {
        int categorie;
        short numero-chambre;
        char adresse[256];
} ;
```

```
struct particulier {
        char nom[64];
        char adresse[256];
};
```

CHAPITRE 2 : INSTRUCTIONS

Une partie de la grammaire du langage C sous forme B.N.F. (Backus Naur Form):

instruction :

déclaration

{ liste-instructions opt }

expression opt;

if (expression) instruction

if (expression) instruction **else** instruction

switch (expression) instruction

while (expression) instruction

do instruction **while** (expression) ;

for (instruction-for; expression opt; expression opt)

instruction

case expression-constante : instruction

default : instruction

break;

continue;

return expression opt;

goto identificateur;

identificateur : instruction

liste-instructions :

instruction

liste-instructions instruction

instruction-for :

déclaration

expression opt;

exemples d'instructions :

un bloc

une expression suivie d'un ;

if (expression) instruction;

case expression-constante : instruction

default : instruction

Le caractère **;** signifie dans certains contextes "instruction vide".

exemple : *while (getchar () != 'Q')*

;

Un ensemble (bloc) d'instructions délimité par { et } est une instruction.

1 INSTRUCTIONS CONDITIONNELLES

1.1 TEST SIMPLE

if (expression) instruction; **if** (expression) instruction1 **else** instruction2; La structuration en "else if" imbriqués : **if** (expression1) instruction1; **else if** (expression2) instruction2; **else if ...** **else** instructionN; /* le cas par défaut */	**if** (expression) instruction; **if** (expression) instruction1 **else** instruction2; La structuration en "else if" imbriqués : **if** (expression1) instruction1; **else if** (expression2) instruction2; **else if** ... **else** instructionN; /* le cas par défaut */

remarque :

if (expression) <==> if (expression != 0)

exemples :

if (a >100) *b = a % 100;*	*if ((x <y) || (y <z))* *{* *x = x * y;* *z = z / y;* *}*

attention : Une mauvaise présentation des sources peut conduire à des résultats inattendus dans le cadre des "if else imbriqués". Un enjoliveur (formateur) de sources peut alors s'avérer être très utile.

Exemple :

mauvaise présentation *if (var1 > 0)* *if (x > y)* *z = x;* *else* *z = y;*	Dans l'exemple ci-contre, le "else" est associé avec le "if" le plus interne. Une présentation correcte est : *if (var1 > 0)* *if (x > y)* *z = x;* *else* *z = y;*
Aussi pour associer le "else" au premier "if", il convient d'utiliser des accolades : *if (var1 > 0)* *{* *if (x > y)* *z = x;* *}* *else* *z = y;*	

1.2 TEST MULTIPLE, AIGUILLAGE

Cette instruction permet de comparer une expression de type entier à différentes valeurs constantes. Ces constantes, nécessairement toutes différentes, doivent être de type entier, sinon le recours à la construction des "else if" s'impose.

```
switch  (expression)
{
        case CTE1 :   instruction1;
        case CTE2 :   instruction2;
                ...
        case CTEN :   instructionN;
        default : instructionD;          /* facultatif */
}
```

Ordre d évaluation :

expression est comparée successivement à CTE1, CTE2 ... Si elle est égale à une des constantes CTEi, les instructions qui suivent l'étiquette CTEi sont exécutées jusqu'à la rencontre d'une directive de débranchement inconditionnelle.

En général, le mot-clé **break** déclenche le débranchement à la première instruction située après la } du switch.
Le traditionnel et si décrié **goto** peut aussi être utilisé, ou encore un **return** pour sortir de la fonction.

Si expression n'est égale à aucune des constantes :
alors si l'étiquette default existe (non nécessairement à la fin du switch), l'instruction qui la suit est exécutée.
sinon le switch est sans action.

exemple :

```
int c;
c = getchar ();
switch ( c )
{
        case    'a' :    case    'A' :
        case    'e' :    case    'E' :
        case    'i ' :   case    'I' :
        case    'o' :    case    'O' :
        case    'u' :    case    'U' :
        case    'y' :    case    'Y' : printf ("voyelle \n"); break;
        case    'b' :    case    'B' :
        case    'c' :    case    'C' :
        ...
        case    'z' :    case    'Z' : printf ("consonne \n"); break;
        case    '0' :
        case    '1' :
        ...
        case    '9' : printf ("chiffre \n");break;
        default : printf ("vous ne jouerez décidemment pas aux chiffres et \
                                aux lettres \n"); break;
}
```

Remarquez le "break" à la dernière ligne. Ceci est une bonne pratique à adopter dans l'optique où d'autres "case" pourraient être ajoutés lors d'une phase ultérieure du codage. L'absence de ce "break" dans le cas "default" conduirait à l'exécution de l'instruction du cas suivant.

2 INSTRUCTIONS DE BOUCLE

2.1 BOUCLE WHILE

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

L'expression est évaluée, et si une valeur non nulle est rencontrée l'instruction est exécutée. Le processus continue de manière cyclique jusqu'à ce que la valeur de l'expression soit nulle; auquel cas le programme reprend son exécution après l'instruction.

exemples :

Le plus classique

```
int notes[20], i;

i = 0;
while (i < 20)
{
        notes[i] = 0;
        i++;
}
```

Code plus concis

```
int notes[20], i = 0;

while ( i < 20 )
        notes[i++] = 0;
```

2.2 BOUCLE DO ... WHILE

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

Même évaluation que précédemment, mais l'instruction est évaluée au moins une fois.

2.3 BOUCLE FOR

Boucle for	Boucle while équivalente

for (expression1; expression2; expression3)
instruction;
	expression1;
	while (expression2)
	{
	instruction;
	expression3;
	}

expression1	suite d'instructions séparées par le délimiteur ',' et exécutées une et une seule fois à l'entrée dans la boucle
expression2	**tant que** expression2 est vraie, le corps de la boucle instruction est exécuté
expression3	suite d'instructions séparées par le délimiteur ',' et exécutées après le corps de la boucle, suite à quoi expression2 est de nouveau évaluée ...

Chacune de ces 3 parties est optionnelle, ainsi que le corps de la boucle.

Communément les expressions 1 et 3 sont des affectations, tandis que l'expression 2 comprend un opérateur relationnel.

Exemples:

<table>
<tr><td>Le plus classique
<pre>
int notes[20], i;
for (i = 0; i < 20; i++)
        notes[i] = 0;
</pre></td><td>expression3 vide : moins lisible
<pre>
int notes[20], i;
for (i = 0; i < 20;)
        notes[i++] = 0;
</pre></td></tr>
<tr><td>expression1 et expression3 vides : peu lisible
<pre>
int notes[20], i = 0;
for (; i < 20;)
        notes[i++] = 0;
</pre></td><td>expression3 affectation et instruction vide: peu lisible et dangereux
<pre>
int notes[20], i;
for (i = 0; i < 20; notes[i++] = 0)
        ;
</pre></td></tr>
<tr><td colspan="2">Mise à 0 des éléments de la seconde diagonale de la matrice
<pre>
int i, j, matrice[20][20];
for (i = 0, j = 20 - 1; (i < 20) && (j >= 0); i++, j--)
        matrice[i][j] = 0;
</pre></td></tr>
</table>

Attention aux boucles infinies :

<table>
<tr><td>Pas de test d'arrêt
<pre>
for (; ;)
{
        ...
}
</pre></td><td>Test d'arrêt toujours vrai
<pre>
for (i = 0; i >= 0; i++)
{
        ...
}
</pre></td></tr>
</table>

3 INSTRUCTIONS DE CONTROLE

3.1 SORTIE DE BOUCLE OU D'AIGUILLAGE : BREAK

```
while ( ... )
{
        ...
        .
        if (expression)
                break;
        ...
}
```

```
for ( ...; ...; ... )
{
        …
        if (expression)
                break;
        …
}
```

```
while ( ... )
{
        ...
        .
        while (…)
        {
                ...
                if (expression)
                        break;
                ...
        }
        …
}
```

```
switch (expression)
{
        case CTE1 : ...; break;
        ...
        case CTEN : ...; break;
        default : ...; break;
}
```

3.2 SAUT À L'ITERATION SUIVANTE : CONTINUE

Retour au contrôle de la boucle : l'itération de la boucle

```
while ( ... )
{
        ...
        if (expression)
                continue;
        ...
}
```

```
for ( ...; ...; ... )
{
        …
        if (expression)
                continue;
        …
}
```

remarque : **continue** ne peut être utilisé dans un **switch**

3.3 SAUT A UNE ETIQUETTE : GOTO

Le goto si décrié (programmation spaghetti) existe. Il est vivement conseillé d'en limiter l'usage. Ainsi on pourra l'utiliser pour sortir de plusieurs boucles imbriquées.

```
        if ( ... )
                goto label;
        ...
label :
        ...
        while ( ... ) {
                while ( ... ) {
                        ...
                        if (erreur)
                                goto erreur_label;
                        ...
                }
                ...
        }
        ...
erreur_label :
        ...
```

Remarques : Les étiquettes ont une visibilité limitée à un bloc de fonction.

Pas de goto entre 2 fonctions.

3.4 SORTIE D'UNE FONCTION : RETURN

Suite à une instruction "**return expression"** dans le corps d'une fonction, le contrôle du flot des instructions est rendu à l'appelant.

Le type de l'expression de retour doit être en adéquation avec le type de retour de la fonction.

Exemple :

```
/* déclaration prototype de la fonction */
extern int valeur_absolue ( int x );

int main()
{
        int a = 6;
        /* appel de la fonction */
        printf("%d \n", valeur_absolue(a));
        return 0;
}
```

```
/* définition de la fonction */
int valeur_absolue ( int x )
{
        if (x > 0)
                return x;
        return (-x);
}
```

3.5 SORTIE D'UN PROGRAMME : EXIT()

La **primitive système exit()** provoque la sortie propre de l'exécution d'un programme.
Les fichiers ouverts sont fermés après les éventuelles écritures sur disque.
La fonction exit() a un argument de type entier qui est un code de retour à interprétrer par l'appelant.
Exemples :

```
exit(0);        /* sortie sans erreur */
exit(1);        /* sortie avec un code erreur égal à 1 : erreur de syntaxe */
exit(2);        /* sortie avec un code erreur égal à 2 : erreur sur les fichiers */
...
```

CHAPITRE 3 : OPERATEURS

1 OPERATEURS ARITHMETIQUES

Opérateurs binaires :	+	addition
	-	soustraction
	*	multiplication
	/	division entière
	%n	modulo (reste de la division euclidienne)
Opérateur unaire :	-	opposé d'un nombre

- La division entière tronque toute partie décimale.
- Le modulo ne s'applique ni à des flottants ni à des doubles.
- Une expression arithmétique est évaluée de la gauche vers la droite en tenant compte de la priorité des opérateurs. Les opérateurs *, /, % ont la même priorité et une priorité supérieure à celle des opérateurs + et -.
- Les opérateurs associatifs et commutatifs tels * et + n'ont pas leur ordre d'évaluation spécifié par le langage. Le compilateur est libre de réorganiser l'arbre syntaxique d'une expression parenthésée comprenant de tels opérateurs.

2 OPERATEURS RELATIONNELS

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

- Le résultat d'une comparaison est soit 1 (vrai) soit 0 (faux).
- Dans une expression, ces opérateurs sont évalués de gauche à droite.
- Les opérateurs >, >=, <, <= ont une priorité plus forte que celle des opérateurs == et !=.
- Ils ont tous une priorité plus faible que celle des opérateurs arithmétiques.

  Ainsi : **var1 < var2 - 1;** est évaluée comme **var1 < (var2 - 1);**

Attention :

x <= y <= z ne signifie pas que y est dans l'intervalle [x,z].

x <= y <= z <==> (x <= y) <= z

Interprétation de (x <= y) <= z

si (x <= y) est faux alors son évaluation est 0 et z est comparé à 0 : 0 <= z

si (x <= y) est vrai alors son évaluation est une valeur différente de 0 et z est comparé à cette valeur

Pour exprimer la sémantique "y appartient à l'intervalle [x,z]", on écrit :

(x <= y) && (y <= z)

Interprétation de x <= y == z <= u

x <= y == z <= u <==> (x <= y) == (z <= u)

donc 4 cas possibles :

vrai == vrai donc valeur différente de 0 car vrai

vrai == faux donc 0 car faux

faux == vrai donc 0 car faux

faux == faux donc valeur différente de 0 car vrai

3 OPERATEURS LOGIQUES

Le langage C ne supporte pas le type booléen.

Une expression nulle est considérée comme fausse.

Toute expression non nulle est considérée comme vraie.

Opérateurs binaires :	**&&**	et logique
	||	ou logique

Opérateur unaire :	**!**	négation logique

if (expression) **<==>** **if (expression != 0)**

if (!expression) **<==>** **if (expression == 0)**

- && et ||sont évalués de la gauche vers la droite.
- Leur évaluation cesse dès que faux pour && et vrai pour || sont atteints.
- On dit qu'il y a **rupture d'évaluation.**

exemples :

if ((x <= borne_inf) && (x <= borne_sup)) { ... }

Si l'expression (x <= borne_inf) est fausse

alors l'expression (x <= borne_sup) n'est pas évaluée ainsi que le corps du if.

while (((c = getchar()) != '\0') && (i++ <= MAX)) { ... }

Si l'expression *((c = getchar()) != '\0') est fausse*

alors l'expression (i++ <= MAX) n'est pas évaluée ainsi que le corps du if.

Les effets de bord sont à proscrire dans de tels contextes.

exemple :

```
int x, y = 3, i;
scanf(" %d", &i);
if ((x = ++i) && (y =++i))
{
        ... /* pas de modification de x, y et i
}
```

Que valent x, y et i ?

Si la valeur saisie de i est -1 alors l' expression x = ++i est évaluée à 0, il y a rupture d'évaluation du && et par suite l'expression y = ++i n'est pas évaluée ainsi que le corps du if. Les variables x, y et i valent respectivement 0, toujours 3 et 0.

Si la valeur saisie de i est différente de -1 alors l'expression x = ++i n'est pas évaluée à 0, il n'y a pas rupture d'évaluation du && et par suite l'expression y = ++i est évaluée ainsi que le corps du if.

Exemple :

Si la valeur saisie de i est 4, alors les variables x, y et i valent respectivement 5, 6 et 6.

4 CONVERSIONS ARITHMETIQUES

Beaucoup d'opérateurs provoquent les conversions et produisent les types résultants d'une façon similaire. Ce mécanisme est appelé la "**conversion arithmétique usuelle**".

Si un des opérandes est du type **long double**, l'autre est **converti en long double.**

Sinon, si un des opérandes est du type **double**, l'autre est **converti en double.**

Sinon, si un des opérandes est du type **float**, l'autre est **converti en float.**

Sinon, les **promotions d'entiers** sont appliqués aux deux opérandes.

Ensuite, si un des opérandes est du type **unsigned long**, l'autre est **converti en unsigned long**.

Sinon, si un opérande est du type **long int** et l'autre du type **unsigned int**, *puis si un long int peut représenter toutes les valeurs d'un unsigned int*, le **unsigned int est converti en long int** ; sinon les deux opérandes sont **convertis en unsigned long int**.
Sinon, si un des opérandes est du type **long int**, l'autre est **converti en long int.**
Sinon, si un des opérandes est du type **unsigned int**, l'autre est **converti en unsigned int.**
Sinon, les deux opérandes sont des **int**.

Rappel : La **promotion d'entiers** est la conversion d'un type char, short, d'une énumération, d'un champ de bits int (signé ou non) en un int ou un unsigned int si l'int n'est pas suffisant.

Les flottants ne sont plus convertis en double (cf K&R 78).

5 OPERATEURS D'INCREMENT ET DE DECREMENT

- L'opérateur d'incrément ++ est utilisé pour ajouter 1 à son opérande.
- L'opérateur de décrément -- est utilisé pour retrancher 1 à son opérande.
- Les deux opérateurs peuvent être utilisés soit en notation préfixée ou postfixée.
- L'expression ++x incrémente x avant l'utilisation de sa valeur.
- L'expression x++ incrémente x après l'utilisation de sa valeur.
- L'expression --x décrémente x avant l'utilisation de sa valeur.
- L'expression x-- décrémente x après l'utilisation de sa valeur.

Exemple :

```
int x = 2;
int y = 0;

y = ++x;      /* x vaut 3, y vaut 3 */
y = x++;      /* x vaut 3, y vaut 3, puis x vaut 4*/
y = x--;      /* x vaut 4, y vaut 4, puis x vaut 3 */
```

6 OPERATEURS AU NIVEAU DU BIT

Le langage C est souvent qualifié de langage de moyen niveau dans la mesure où il offre la capacité d'opérer sur les bits.

Ils sont applicables aux **types entiers** (char, short, int, long) et aux **types entiers non signés** (unsigned). Ils n'opèrent ni sur les flottants ni sur les doubles.

Le bit de poids fort d'un entier est son signe : 0 nombre positif, 1 nombre négatif

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.

Exemple :

Pour une variable 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**.

Exemple :

Pour une variable de type char (8 bits) :

-1 sera représenté par 1 1 1 1 1 1 1 1

-2 sera représenté par 1 1 1 1 1 1 1 0

-12 sera représenté par 1 1 1 1 0 1 0 0

preuve

type char (8 bits)	-1	-2	-12
Valeur absolue	0 0 0 0 0 0 0 1	0 0 0 0 0 0 1 0	0 0 0 0 1 1 0 0
Complément à 1	1 1 1 1 1 1 1 0	1 1 1 1 1 1 0 1	1 1 1 1 0 0 1 1
Ajout de 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 0	1 1 1 1 0 1 0 0

Opérateurs binaires :

&	et binaire	et logique bit à bit des 2 opérandes
|	ou binaire inclusif	ou inclusif bit à bit des 2 opérandes
^	ou binaire exclusif	ou exclusif bit à bit des 2 opérandes
<< n	décalage à gauche	décalage non circulaire de n bits à gauche
>> n	décalage à droite	décalage non circulaire de n bits à droite

Remarque : Bit à bit se traduit par **bitwise** en anglais

Opérateur unaire :

~	complément à 1	les bits à 0 passent à 1 et réciproquement

Attention : Ne pas confondre les opérateurs & et | avec les opérateurs logiques && et ||.

Lors d'un **décalage à gauche**, les bits les plus à gauche sont perdus.

Les positions binaires rendues vacantes sont **remplies par des 0**.

Lors d'un **décalage à droite**, les bits les plus à droite sont perdus.

Si l'entier à décaler est **non signé** alors les positions binaires rendues vacantes sont **remplies par des 0**, s'il est **signé** le remplissage s'effectue à l'aide du **bit de signe**.

exemple : *char x = 11, y = 55;*

x	0	0	0	0	1	0	1	1
y	0	0	1	1	0	1	1	1
x & y	0	0	0	0	0	0	1	1
x | y	0	0	1	1	1	1	1	1
x ^ y	0	0	1	1	1	1	0	0
x << 3	0	1	0	1	1	0	0	0
y >> 3	0	0	0	0	0	1	1	0
~x	1	1	1	1	0	1	0	0

short x = 5;	x	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
short y = 10;	y	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
x = x | y;	x | y	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1.

short x = 5;	x	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
short y = **-10**;	y	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0
x = x | y;	x | y	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1

char x = 117	0 1 1 1 0 1 0 1	char x = **-10**	1 1 1 1 0 1 1 0
x < < 3	1 0 1 0 1 **0 0 0**	x < < 3	1 0 1 1 0 **0 0 0**
x > > 3	**0 0 0** 0 1 1 1 0	x > > 3	**1 1 1** 1 1 1 1 0

L'opérateur & est utilisé pour mettre à 0 des bits.

exemple :

*#define MASK **0x0F***

*int x = **0x34**;*

*x = **x & MASK**;*

x	**0**	**0**	**1**	**1**	0	1	0	0
MASK	**0**	**0**	**0**	**0**	1	1	1	1
x = x & MASK	**0**	**0**	**0**	**0**	0	1	0	0

L'opérateur | est utilisé pour mettre à 1 des bits.

exemple :

*#define MASK **0x0F***

*int x = **0x34;***

*x = **x | MASK**;*

x	0	0	1	1	**0**	**1**	**0**	**0**
MASK	0	0	0	0	**1**	**1**	**1**	**1**
x = x | MASK	0	0	1	1	**1**	1	**1**	**1**

7 OPERATEURS D'AFFECTATION

Le langage C est un langage basée sur les expressions, aussi une affectation peut être utilisée en lieu et place d'une expression. Dans une expression, les affectations sont évaluées de droite à gauche. Les opérateurs d'affectation ont une priorité inférieure à celle de la plupart des opérateurs. Lors d'une affectation, le membre gauche est converti dans le type de celui du type droit. L'opérateur d'affectation de base est =.

Les opérateurs composés à partir des opérateurs arithmétiques : +=, -=, *=, /=, %=
Les opérateurs composés à partir des opérateurs sur les bits : &=, |=, ^=, >>=, <<=

Leur signification est :

x op= y **<==>** **x = x op y**

mais avec l'avantage d'**une seule évaluation de x**.

exemples :

*x *= y;* <==> *x = x * y;*
x >>= y; <==> *x = x >> y;*
*x *= y + 6;* <==> *x = x * (y + 6);*

Les affectations étant des expressions, sont possibles :

leur chaînage : *a = b = c = d = 6;* <==> *a = (b = (c = (d = 6)));*
leur imbrication dans une expression : *if ((x = y - z) <= u) { ... }*

exeemple:

int x = 5, y =12, z = 6;
*x *= y /= z - 3;*
Quelles sont les valeurs de x, y, z?

1) y /= z – 3 <==> y = y / (z -3), soit y vaut 4
2) x *= y <==> x = x * y, soit x vaut 20

8 OPERATEUR TERNAIRE

? :	produit une valeur de manière conditionnelle

syntaxe : expression1 ? expression2 : expression3;

sémantique : si expression1 est vraie
alors évaluation de expression2
sinon évaluation de expression3

Les expressions expression2 et expression3 doivent être de types scalaires (entier, caractère, flottant, double, pointeur).
Le plus souvent cet opérateur est utilisé avec une affectation.

x = (expression1 ? expression2 : expression3);	signifie :
	if (expression1)
	x = expression2;
	else
	x = expression3;

exemples :

```
valeur_absolue_x = ((x >=0) ? x : -x);
max = (( x >= y ) ? x : y);
```

avec une **macro-définition (déconseillé en raison d'effets de bord potentiels):**

```
#define ABS(x)      (((x) >0 ) ? (x) : -(x))
#define MAX(x,y)    (((x) >= (y) ) ? (x) : (y))
```

conseil : Utiliser plutôt des fonctions.

9 OPERATEUR SIZEOF

Il donne la taille en octets de l'objet ou du type donné dans les parenthèses.

Il est évalué à la compilation et ne pénalise donc pas le temps d'exécution.

exemples :

```
int x;              /* déclaration d'un entier x */
float y;            /* déclaration d'un flottant y */
x = sizeof(int);    /* x vaut la taille du type entier */
x = sizeof(y);      /* x vaut la taille du flottant y */
```

Cet opérateur assure la portabilité des programmes en n'étant pas machine dépendante.

10 OPERATEUR DE SEQUENCE

,	Sépare une séquence d'expressions

syntaxe : expression1, expression2, ...

L'évaluation a lieu de la gauche vers la droite.

L'expression expression1,expression2 a pour valeur celle de expression2.

exemple :

```
int a = 5, b = 7;
int c;
c = ( a++, a + b + 8);
```

Que valent a, b et c?

a vaut 6, b vaut 7 et c vaut a + b + 8, soit 21.

Les opérateurs suivants seront étudiés de manière plus approfondie dans la suite du cours.

11 OPERATEUR ()

Il a une sémantique dépendante du contexte.

Soit il modifie l'ordre d'évaluation d'une expression.

```
int a = 3, b = 5;
int c;
c = (a + b) * 10;
```

Soit il indique un appel de fonction : *somme*(*3, 4, 8*)

Soit la liste des paramètres d'une fonction lors de sa déclaration prototype ou de sa définition : *int somme*(*int a, int b, int c*)

12 OPERATEURS SUR LES TABLEAUX

[] élément d'un tableau et déclaration/définition d'un tableau

exemple :

```
int tab[7];
tab[2] = 8;
```

13 OPERATEURS SUR LES POINTEURS

& adresse d'une variable

***** contenu de l'objet pointé par un pointeur

14 OPERATEURS SUR LES STRUCTURES

. accès à un membre de structure par une variable

→ accès à un membre de structure par un pointeur

15 OPERATEUR DE CONVERSION EXPLICITE (CAST)

(type) force le type d'un objet ou d'une expression

exemple : *double x = 7.678;*

double y;

*y = **(int)** x + 22/7; /* y vaut 7 + 3 = 10 et x vaut toujours 7.678 */*

16 SOMMAIRE DES OPÉRATEURS

Priorité	Opérateur	Signification	Exemple
1	→	sélection de membre	pointeur → membre
	.	sélection de membre	objet . membre
	[]	indiçage	pointeur [expr]
	()	appel de fonction	fonction(liste_expr)
	sizeof	taille d'un objet	sizeof (expr)
	sizeof	taille d'un type	sizeof (type)
2	++	post-incrémentation	lvalue ++
	++	pré-incrémentation	++ lvalue
	--	post-décrémentation	lvalue --
	--	pré-décrémentation	-- lvalue
	~	complément	~ expr
	!	non	! expr
	-	moins unaire	- expr
	+	plus unaire	+ expr
	&	adresse de	& lvalue
	*	déréférence	*expr
	()	cast (conversion de type)	(type) expr

3	* / %	multiplication division modulo (reste)	expr1 * expr2 expr1 / expr2 expr1 % expr2
4	+ -	addition (plus) soustraction (moins)	expr1 + expr2 expr1 - expr2
5	<< >>	décalage à gauche décalage à droite	expr1 << expr2 expr1 >>expr2
6	< <= > >=	inférieur à inférieur ou égal à supérieur supérieur ou égal à	expr1 < expr2 expr1 <= expr2 expr1 > expr2 expr1 >= expr2
7	== ! =	égal différent de	expr1 == expr2 expr1 ! = expr2
8	&	ET bit-à-bit	expr1 & expr2
9	^	OU exclusif bit-à-bit	expr1 ^ expr2
10	|	OU inclusif bit-à-bit	expr1 | expr2
11	&&	ET logique	expr1 && expr2
12	||	OU logique inclusif	expr1 || expr2
13	? :	Expression conditionnelle	expr1 ? expr2 : expr3

14	= *= /= %= += -= <<= >>= & = |= ^=	affectation simple multiplication et affectation division et affectation modulo et affectation addition et affectation soustraction et affectation décalage à gauche et affectation décalage à droite et affectation ET et affectation OU inclusif et affectation OU exclusif et affectation	lvalue = expr lvalue *= expr lvalue /= expr lvalue %= expr lvalue += expr lvalue -= expr lvalue <<= expr lvalue >>= expr lvalue & = expr lvalue |=expr lvalue ^=expr
15	,	virgule (séquence)	expr1, expr2

Tous les opérateurs sont associatifs à gauche sauf les unaires et les affectations qui sont associatifs à droite.

CHAPITRE 4 : POINTEURS

1 POINTEURS ET ADRESSES

1.1 DEFINITION

UN POINTEUR EST UNE VARIABLE QUI CONTIENT L'ADRESSE D'UNE AUTRE VARIABLE.

On dit qu'un pointeur pointe sur une variable.

Le langage C encourage l'utilisation judicieuse des pointeurs.

Le libre usage des pointeurs nécessite une programmation soigneuse.

Les pointeurs sont utilisés pour la génération compacte et efficiente de code par le compilateur.

Un pointeur se déclare de la même manière que toute autre variable : ***int * p;***

Le caractère ' ***** ' avant l'identificateur p indique que p est un pointeur.

int spécifie le type de l'objet pointé par p;

On dit que p et un pointeur d'entier.

le type de p est int *

Un pointeur étant une variable il souscrit à toutes les règles concernant la portée, la durée de vie, la classe d'allocation ...

1.2 LES OPERATEURS UNAIRES & ET *

& pour l'adresse d'une variable

***** pour le contenu de l'adresse pointée, on parle alors de la valeur déréférencée d'un pointeur.

exemple :

```
int a, * p;     /* on définit une variable entière a et un pointeur p sur un entier */
p = &a;         /* initialisation du pointeur :p contient alors l'adresse de a,
                   il pointe sur a*/
*p = 4;         /* on donne la valeur 4 à la valeur déréférencée du pointeur p;
                   la variable a vaut  donc 4 */
```

Ainsi, pour un pointeur p, il ne faut pas confondre les 3 notions essentielles :

&p	adresse du pointeur p
p	valeur du pointeur, soit l'adresse d'une autre variable
***p**	contenu de l'adresse pointée par p valeur déréférencée du pointeur p.

Il convient de noter que l'opérateur **&** ne s'applique que sur des objets correspondant à une location mémoire (notion de **lvalue**) :

- les variables de type :
  - caractère, entier, flottant, double
  - pointeur sur un objet de type quelconque
  - tableau, structure, union
- les fonctions (le compilateur "voit" une fonction en tant que son adresse mémoire dans le **segment text**)
- les éléments d'un tableau, les champs d'une structure

Ainsi l'opérateur **&** ne peut être appliqué aux expressions, aux constantes, aux variables de classe de stockage **register**.

Il convient de noter que pour une variable x de type quelconque on a :

***&x < == > * (&x) < == > x**

***** et **&** sont deux opérateurs **unaires** de **même priorité** avec liaison de l'opérande de la **droite vers la gauche**.

Si un pointeur **p** pointe sur une variable **a** alors***p** peut être utilisé partout où **a** pourrait l'être.

exemple :

voici un bloc d'instructions coupé de son contexte :

```
{
        int a, b,  * p;
        a = 10;             /* initialisation de a */
        p = &a;             /* initialisation de p */
        b = *p + 2;         /* soit b = 10 + 2 */
        *p = *p + 4;        /* soit a = a + 4 */
        *p + = 11;          /* soit a + = 11 */
        *p = 0;             /* soit a = 0 */
}
```

1.3 LES OPERATEURS ++ ET --

1.3.1 INCREMENTATION DES VALEURS DEREFERENCEES

Rappel : les deux opérateurs unaires * et ++ ont même priorité, la liaison de l'opérande étant de la droite vers la gauche.

Les équivalences suivantes ont lieu :

++*p	**< == > ++(*p)**	pré_incrémentation de la valeur déréférencée
***++p**	**< == > *(++p)**	
	< == > ++p;	pré_incrémentation du pointeur (adresse "suivante")
	***p;**	valeur déréférencée
***p++**	**< == > * (p++)**	
	< == > * p;	valeur déréférencée
	p++;	post_incrémentation du pointeur (adresse "suivante")

Ainsi pour post_incrémenter la valeur déréférencée d'un pointeur faut-il écrire : (*p)++

exemple :

```
{
        int x, y, * p;
        x = 2;
        p = &x;
        ++*p;           /* soit ++x donc x vaut 3 */
        (*p)++;         /* soit x++ donc x vaudra 4 à la prochaine utilisation */
        y = ++*p;       /* y vaut 5 */
        y = (*p) ++;    /* y vaut 5 et x vaut 6 */
        y = *p;         /* y vaut 6 */
}
```

1.3.2 DECREMENTATION DES VALEURS DEREFERENCEES

Les problèmes rencontrés sont les mêmes que ceux liés à l'incrémentation.

1.4 L'OPERATEUR D'AFFECTATION

Les pointeurs étant des variables, l'affectation entre deux pointeurs est possible :

p1 = p2;

Cette instruction copie la valeur de p2 dans p1. Autrement dit p1 pointe vers le même objet que p2.

Attention : Il faut que les **pointeurs soient de même type**. Sinon un warning est indiqué à la compilation quant à un problème potentiel d'alignement en mémoire. Il convient alors de recourir à des conversions explicites de pointeurs.

```
int * p;
char * q;
...
p = (int *) q;  /* conversion explicite (cast) */
```

2 POINTEURS ET ARGUMENTS DE FONCTIONS

Une utilisation fréquente des pointeurs est de les passer en arguments des appels de fonction.

Rappel :

En C le **passage des paramètres se fait par valeur** (call by value). Ainsi si une fonction modifie les arguments, cette modification n'est valable que dans le corps de la fonction. Au retour de l'appel, les arguments ont conservé leurs valeurs. Les valeurs des arguments sont copiées dans les paramètres formels sur la pile.

<u>Exemple :</u> Un exemple classique est celui de la fonction swap() :

```
extern void swap (int , int );
int main()
{
        int a = 1, b = 2;
        swap(a, b);
        /* a et b valent toujours 1 et 2. */
        printf ("a = %d b = %d \n", a, b);
        return 0;
}
void swap (int x, int y)
{
        int tmp;
        tmp = x;
        x = y;
        y = tmp;
}
```

Pour conserver les modifications au retour de l'appel, **les arguments doivent être des adresses**. **Les paramètres doivent être** alors des variables pouvant contenir des adresses, autrement dit **des pointeurs**.

```
extern void swap (int * x,  int * y);
int main()
{
        int a = 1, b = 2;
        swap(&a, &b);
        /* a et b valent maintenant 2 et 1. */
        printf ("a = %d b = %d \n", a, b);
        return 0;
}
```

```
void swap (int * x,  int *y)
{
        int tmp;
        tmp = *x;
        *x = *y;
        *y = tmp;
}
```

Ce mode de passage des paramètres s'appelle aussi l'**appel par valeur ou par copie** (ce sont les valeurs des pointeurs qui sont copiées sur la pile).

3 POINTEURS ET TABLEAUX

En C une étroite relation existe entre les tableaux et les pointeurs.

Toute opération mettant en oeuvre un sous-tableau ou un élément d'un tableau peut être réalisée à l'aide de pointeurs.

Le code généré à partir des pointeurs sera en général plus rapide.

L'**opérateur primaire []** permet l'accès à un élément d'un tableau.

a [n-1] permet l'accès au n^{ième} élément d'un tableau a.

Les indices des tableaux commencent à 0.

Considérons les instructions suivantes :

```
int x;
int a [10];          /* définition d'un tableau de 10 entiers */
int *  pa;           /* définition d'un pointeur d'entier */
a[0] = 4;            /* affecte la valeur 4 au 1er élément */
a[3] = 2;            /* affecte la valeur 2 au 4ème élément */
pa = &a[0];          /* pa pointe sur le 1er élément du tableau.
```

*pa contient l'adresse du 1er element */*

*x = *pa;* */* copie la valeur du 1er élément du tableau*

*dans la variable x. x vaut 4. */*

L'opérateur primaire **[] à une priorité plus élevée** que celle de l'opérateur unaire **&.**

La liaison pour **[]** se fait de la **gauche vers la droite**.

Ainsi : **&a[0] < == > &(a[0])**

Si pa pointe sur le n^{ième} élément d'un tableau, la notation pa + i permet de pointer sur le (n + i)^{ième} élément.

Exemples :

Si pa = &a[0]	Si pa = &a[3]
alors	alors
pa + 1 < == > &a[0] + 1	pa + 4 < == > &a[3] + 4
< == > &a[0 + 1]	< == > &a[3 + 4]
< == > &a[1]	< == > &a[7]

plus généralement :

Si pa = &a[n] alors pa + i < == > &a[n] + i

< == > &a[n + i]

La notation * (pa + i) est la valeur déréférencée du pointeur pa + i .

Si pa = &a[n] alors *(pa + i) <==> a[n + i]

Exemples :

Si pa = &a[0] alors :

*(pa + 1) < == > *(&a[0] + 1)

< == > * (&a[0 + 1])

< == > * (&a[1])

< == > a [1]

Si pa = &a[3] alors :

*(pa + 4) < == > *(&a[3] + 4)

< == > * (&a[3 + 4])

< == > * (&a[7])

< == > a [7]

plus généralement :

Si pa = &a[n] alors :

***(pa + i) < == > *(&a[n] + i)**

< == > * (&a[n + i])

< == > a [n + i]

Les parenthèses dans l'expression *(pa + i) sont nécessaires pour l'accès au (n + i)ième élément, si pa = &a[n].

En effet l'**opérateur unaire * a une priorité plus élévée** que celle de l'**opérateur binaire** +.

Attention :

L'expression *pa + i ramène la valeur déréférencée du pointeur pa augmentée de i.

Reprenons et complétons les expressions vues précédemment :

```
int x, y;
int a [10];          /* définition d'un tableau de 10 entiers */
int * pa;            /* définition d'un pointeur d'entier */
int * pb;            /* définition d'un pointeur d'entier */
a[0] = 4;            /* affecte la valeur 4 au 1er élément */

a[3] = 2;            /* affecte la valeur 2 au 4ème élément */
a[7] = 11;           /* affecte la valeur 11 au 8ème élément */
pa = &a[0];          /* pa pointe sur le 1er élément du tableau.
                        pa contient l'adresse du 1er élément */
x = *pa;             /* copie la valeur du 1er élément du tableau dans la
                        variable x. x vaut 4. */
pb = pa + 3;         /*      pb = &a[3];    */
x = *pb;             /* x vaut 2 */
y = *(pa + 3);       /* y vaut 2 */
x = *(pb + 4);       /* x vaut 11 */
x = *pb + 4;         /* x vaut 2 + 4 */
```

En C le nom d'un tableau est synonyme de la location mémoire de son premier élément :

&a[0] < == > a

d'où : **pa = &a[0] < == > pa = a**

Plus généralement :

&a[i] < == > &a[0 + i] < == > &a[0] + i **< == > a + i**

d'où : **pa = &a[i] < == > pa = a + i**

Par suite : ***(a + i)** < == > *(&a[i])

< == > a[i]

4 POINTEURS GENERIQUES

Information : Un pointeur de n'importe quel type peut être affecté à un pointeur générique (universel) de type **void *** sans utiliser de conversion:

int x = 3;
int * pi;
**void *** p;
pi = &x;
p = pi; // particulier pi vers général p
pi = (int *) p; // général p vers particulier pi
	avec un pointeur p de type void *	

	INTERDIT	**AUTORISE**
	p++	((int *) p)++
	*p	* (int *) p

Les pointeurs génériques sont souvent utilisés dans des structures de données génériques et en paramètres de fonction et comme type de retour d'une fonction.

RESUME

Adresse du ième élément du tableau a	**&a[i]**	**a + i**	
Affectation du pointeur pa sur le ième élément du tableau a	**pa = &a[i]**	**pa = a + i**	
Accès au ième élément du tableau a	**a[i]**	***(a + i)**	***pa**

Niveaux	**Déclaration**	**Dans ce contexte**
Entier **0**	**int** x	x associé à **0** (pas d'indirection; pas de montée)
Tableau (a pointeur constant) **+1**	**int** a**[**10**]**	**[]** associé à **+1** (une indirection, une montée)
Pointeur (pa pointeur variable) **+1**	**int** * pa	* associé à **+1** (une indirection, une montée)
Opérateur d'adressse & **+1**	pa = **&** x **+1** = **+1** 0	**&** associé à **+1** (une indirection, une montée)

Niveaux	**Dans ce contexte**
a [0] = 4; ***+1 -1 = 0***	**[]** associé à **-1** (une déférenciation, une descente)
pa = & a [0]; ***+1 = +1 +1 -1***	CA MARCHE!
*x = * pa;* ***0 = -1 +1***	* associé à **-1** (une déférenciation, une descente)
pb = pa + 3; ***+1 = +1 0***	CA MARCHE!
*y = * (pa + 3);* ***0 = -1 (+1 + 0)***	CA MARCHE!
x = * pb + 4; ***0 = -1 +1 + 0***	CA MARCHE!

CHAPITRE 5 : TYPES DÉRIVÉS

1 TABLEAU À UNE DIMENSION

Un tableau est un ensemble dimensionné d'objets de même type.

1.1 DÉCLARATION

Exemples : *int table [50];* *tableau de 50 entiers*

char buffer [20]; *tableau de 20 caractères*

Les indices de tableaux commencent à zéro.

Ainsi, l'indice d'un tableau de dimension N va de zéro à N-1.

Attention :

Le compilateur ne fait aucune vérification du dépassement des bornes d'un tableau.

L'**identificateur d'un tableau** est en fait une **constante** égale à l'adresse du premier élément du tableau. Cette particularité sera étudiée dans le chapitre sur les "pointeurs et tableaux".

Il est possible de déclarer des tableaux d'objets dérivés :

tableau de pointeurs, tableau de structures

tableau de tableaux (multidimensionnel)

L'omission de la dimension d'un tableau n'est possible que dans 3 cas :

si la déclaration comporte une initialisation

lors d'une déclaration de variable externe déjà déclarée dans un autre source (allouée ailleurs)

lorsque l'on déclare un paramètre de fonction qui est l'adresse d'un tableau.

La déclaration d'un tableau se traduit par une allocation mémoire de taille égale à la dimension par la taille du type de chaque élément.

1.2 INITIALISATION

1.2.1 INITIALISATION À LA DÉCLARATION

Pour initialiser un tableau à sa déclaration, il doit être :

- soit une variable globale
- soit une variable locale avec une classe d'allocation automatique ou statique.

Dans les deux cas, le tableau sera alloué à la compilation.

Les valeurs d'initialisation doivent obligatoirement être des constantes évaluées à la compilation.

Exemples

Variables globales *int tabi [5] = {1, 2, 3, 4 , 5};*

char tabc [4] = {'a', 'b', 'c', 'd'};

Variables locales (dans une fonction) :

automatique: *float tf1[4] = {1.3, 1.2, 5.4, 3.14};*

statique*:* *static float tf2[4] = {1.3, 1.2, 5.4, 3.14};*

Il est possible de ne pas initialiser complètement un tableau à sa déclaration.

Dans le cas d'un tableau global ou statique, le compilateur remplit les éléments non initialisés par des zéros pour les nombres ou par des caractères nuls '\o' pour les caractères.

Dans le cas d'un tableau local, les valeurs d'initialisation sont aléatoires.

Exemples :

int table [6] = {1, 2, 3};

donne :

1	2	3	0	0	0

char buf[6] = {'a', 'b'};

donne :

a	b	\0	\0	\0	\0

Il est également possible d'omettre la dimension du tableau. Le compilateur allouera la place mémoire nécessaire pour stocker les éléments de la liste d'initialisation.

Exemples :

int table [] = {1, 2, 3};

donne :

1	2	3

char buf [] = {'a', 'b', 'c', 'd'};

donne :

a	b	c	d

1.2.2 INITIALISATION PAR UNE INSTRUCTION

Exemples :

élément par élément	***par une boucle***
char buf [3]; *int table [10], i = 0;* *buf [0] = 'a';* *buf [1] = 'b';* *buf [2] = 'c';*	*while (i <10)* *{* *table [i] = 0;* *i++;* *}*

2 TABLEAUX A PLUSIEURS DIMENSIONS

2.1 DÉCLARATION

Exemple :

int matrice [2] [5]; tableau de 2 lignes de 5 entiers

char buffer [5] [80]; tableau de 5 lignes de 80 caractères

2.2 INITIALISATION

2.2.1 INITIALISATION À LA DÉCLARATION

Exemple : *int t [4]] [2] = {1,2,3,4,5};*

donne :

1	2
3	4
5	0
0	0

L'utilisation des accolades imbriquées dans la liste d'initialisation permet de forcer la disposition.

Exemple : *int t [2] [4] = {{1,2}, {3,4,5}};*

donne :

1	2	0	0
3	4	5	0

Seule dimension la plus externe peut être omise.

Exemple : *int t [] [3] = {1,2,3,4,5,6};*

donne :

1	2	3
4	5	6

3 POINTEURS ET TABLEAUX

Nous avons vu que l'identificateur d'un tableau est une constante égale à l'adresse du premier élément.

Exemple : *short tab[3] = {1,2,3};*

Il est possible de déclarer un pointeur et de lui affecter cette constante :

*short * pt;*

pt = tab;

Suite à cette instruction, tab et pt contiennent tous les deux l'adresse du premier élément du tableau. La différence tient au fait que pt est modifiable alors que tab ne l'est pas

On peut alors se déplacer dans le tableau et accéder aux éléments en utilisant pt :

pt++; fait pointer sur l'élément suivant

*pt vaut 2

*(pt + 1) vaut 3

L'**opérateur d'indexation []** est aussi utilisable sur un pointeur :

pt [0] vaut 2

pt [1] vaut 3

De même, il est possible d'appliquer certains opérateurs à un nom de tableau :

==, !=, <, >, <=, >=, -, +, *, []

Exemple : ***tab** vaut tab [0]

ATTENTION :

En revanche, les opérateurs suivants sont interdits, puisqu'un nom de tableau est une constante : =, +=, -=, ++, --, & (opérateur adresse)

4 LES CHAÎNES DE CARACTÈRES

Il n'existe pas en C de type chaîne de caractères. Toutefois, la librairie standard offre un certain nombre de fonctions de traitements des chaînes de caractères. Pour ces fonctions une chaîne de caractères, c'est une suite d'octets dont le dernier contient **le caractère nul '\0'.**

4.1 DÉCLARATION

char ch [] = "constante chaîne";

*char * s = "constante chaîne";*

char buffer [20];

buffer [0] = 'm';

...

buffer [15] = 'b';

Ici, les 3 identificateurs ch, s et buffer sont utilisables comme des chaînes de caractères.

Dans le 1er cas, ch est un nom de tableau de la taille de la constante chaîne.

Dans le 2ème cas, s pointe sur l'adresse de la constante chaîne, s est un pointeur modifiable.

Dans le 3ème cas, buffer est un tableau de caractères dont les éléments sont affectés un par un pour construire une chaîne de caractères à l'éxécution.

4.2 FONCTIONS SUR LES CHAÎNES

fonctions d'entrée-sortie sur les chaînes :

printf	possibilité d'afficher une chaine (%s)
scanf	possibilité de lire une chaine (%s)
gets	lecture d'une chaine
puts	écriture d'une chaine

fonctions de traitements :

strlen	longueur d'une chaine
strcpy	copie de chaine
strcat	concaténation de chaines
strcmp	comparaison de chaines

Pour plus de précisions, se reporter au chapitre sur la librairie standard.

Exemple :

```
static char message [] = "bonjour";
int  length = 0;
printf ("%s \n", message);
length = strlen (message);
printf ("longueur chaine : %d \n", length);
```

4.3 COPIE DE TABLEAUX DE CARACTERES

3 versions

tableau

pointeur + indice

pointeur + increment

Clarifier le contrat

Copier le contenu d'un tableau dans un autre

Le tableau destination doit être assez grand

Le tableau source doit avoir une marque de fin ('\0')

Le tableau source ne doit pas être modifié.

La marque de fin sera ajoutée au tableau destination

2 paramètres :

dst = argument de sortie

src = argument d'entrée

Vision tableau
On ne voit pas de pointeur …

```
void copy(char dst[], const char src[])
{
        int i;
        for (i=0;src[i] != '\0'; i++)
        {
                dst[i] = src[i];
        }
        dst[i] = '\0';
}
```

Vision mixte …
Utilisation indice 0, incrément des pointeurs …
src et dst sont jetables

```
void copy(char dst[], const char src[])
{
        for ( ;src[0] != '\0'; src++, dst++)
        {
                dst[0] = src[0];
        }
        dst[0] = '\0';
}
```

Vision pointeurs …
Utilisation indirection, incrément des pointeurs …
src et dst sont jetables

```
void copy(char * dst, const char * src)
{
        for ( ;*src != '\0'; src++, dst++)
        {
                *dst = *src;
        }
        *dst = '\0';
}
```

Ecritures compactes

```
void copy(char * dst, const char * src)
{
        while (*src != '\0')
        {
                *dst++ = *src++;
        }
        *dst = '\0';
}


void copy(char * dst, const char * src)
{
        while ((*dst++ = *src++) != '\0')
        {
                ;
        }
}

/* THE MUST! */
void copy(char * dst, const char * src)
{
        while (*dst++ = *src++)
        {
                ;
        }
}


void copy(char * dst, const char * src)
{
        while (*dst++ = *src++);
}


/* K&R 78 */

void copy(char * dst, const char * src)
{
        while (*dst++ = *src++);
}
```

5 LES STRUCTURES

5.1 DÉFINITION

Une structure C est un objet défini par l'utilisateur, et qui réunit sous forme d'un type unique une collection d'objets de types différents.

5.2 DÉCLARATION

Exemple :

```
struct date
{
        int jour, mois, année;
};
```

Cette déclaration fait de **struct date** un type C, souscrivant au règles classiques de visibilité.

Les identificateurs utilisés dans la définition des membres de la structure serviront à manipuler ces membres dans les instructions du programme.

Il n'y a pas de conflit entre un identificateur interne à une structure et un objet externe :

```
long l;
struct  machin
{
        int * l;
        ...
};
```

Un nom de membre ne peut être utilisé 2 fois dans une structure.

Déclaration de variables :

Exemple :

*struct date d, dt [7], * p, ** pp;*

d	variable de type **struct date**
dt	tableau de type **struct date [7]** de 7 structures de type struct date
p	pointeur de type **struct date *** sur un objet de type struct date
pp	pointeur de type **struct date **** sur un pointeur de type struct date *

5.3 ALLOCATION MÉMOIRE

La déclaration du type *struct machin* ne réserve aucune place mémoire.

```
struct machin {
        char b [3];          /* 3 octets */
        int x;               /* 2 octets */
        char c;              /* 1 octet */
        long l;              /* 4 octets */
};
```

En revanche, la déclaration de la variable m de type *struct machin* réserve de la place mémoire.

```
struct machin m;
```

Dans un objet de type structure , le compilateur alloue une zone mémoire dans laquelle chaque membre est stocké dans son ordre d'apparition, et sur une longueur déterminée par son type. Cependant , sauf option de compilation spécifique (structure "packée"), chaque membre est aligné sur une frontière de mot (sauf les caractères) . Une structure peut donc occuper plus de place qu'il n'y paraît.

```
#include <stdio.h>

struct machin {
        char b [3];      /* 3 octets */
        int x;           /* 4 octets */
        char c;          /* 1 octet */
        long l;          /* 4 octets */
};
```

```
int main()
{
   struct machin m;

   printf("taille de struct machin = %d \n", sizeof(struct machin));
   printf("taille de m = %d \n", sizeof(m));
  return 0;
}
```

Sur un machine 32 bits, mots de 4 octets, la taille de la structure est de 16 octets :

(3 + **1**) + 4 + (1 + **3**) + 4 (**1** et **3** octets pour l'**alignement** sur les frontières de mots).

Sur un machine 32 bits, une structure occupe toujours un nombre d'octets multiple de 4: des octets d'extrémité peuvent donc également être "ajoutés" après le dernier membre. Ainsi, si dans la structure s on ajoute le membre 'char a', elle occupera tout de même 20 octets et non 17.

(3 + **1**) + 4 + (1 + **3**) + 4 + (1 + **3**)

Ce mécanisme est transparent à la programmation si le programmeur s'astreint à manipuler ses structures à l'aide des opérateurs prévus à cet effet dans le langage.

Par souci d'optimisation de place, il est conseillé de ranger les champs par taille croissante :

```
struct s {
        char b [3]; /* 3 octets */
        int x;   /* 4 octets */
        char c;/* 1 octet */
        long l; /* 4 octets */
}; 16 octets
```

```
struct s {
        char c; /* 1 octet */
        char b [3]; /* 3 octets */
        int x;   /* 4 octets */
        long l; /* 4 octets */
}; Optimisation : 12 octets
```

struct s { *char b [3]; /* 3 octets */* *int x; /* 4 octets */* *char c;/* 1 octet */* *long l; /* 4 octets */* *char a;* *};* ***20 octets***	*struct s {* *char a; /* 1 octet */* *char c; /* 1 octet */* *char b [3]; /* 3 octets */* *int x; /* 4 octets */* *long l; /* 4 octets */* *};* ***Optimisation : 16 octets***

<u>Remarque : écritures déconseillées</u>

1) la définition et la déclaration peuvent être cumulées :

```
struct date {
        int jour, mois, annee :
} d, table [7], * p;

struct date ** pp;      /* type date connu */
```

2) le nom du type est facultatif :

```
struct {
        int jour, mois, annee;
} d, table [7];
```

mais *struct date ** pp;* /* NE COMPILE PAS : type date inconnu */

5.4 INITIALISATION DES STRUCTURES

L'initialisation est possible pour les structures et les tableaux de structures globaux et locaux automatiques et statiques.

La syntaxe est identique aux tableaux, les membres de la structure étant remplis dans leur ordre d'apparition. Les éléments restant sont mis à 0. Cette initialisation peut être cumulée avec la déclaration.

Exemples :

```
struct date d= {01, 05, 1994};
struct date table [2] = { {01, 05, 1994} , {24, 12, 1994}};
struct date2 {
        char * nom ;
        struct date dt;
}  table2 [] = { {"Dimanche", 01, 05, 1994},  {"Samedi", 24, 12, 1994}} ;

struct  salarie {
        char nom[32];
        short age;
        flot salaire;
}  table [4] = {{"Dupont", 31, 9900}, {"Martin", 33, 12700}};
```

remarque : table [2] et table [3] sont à 0.

5.5 UTILISATION D'UNE VARIABLE DE TYPE STRUCTURE

Accès au membre d'une structure depuis une variable : **opérateur '.'**

Exemples :

d.jour	est égal à 01
d.mois	est égal à 05
d.annee	est égal à 1994
table2[0].nom	pointe sur "Dimanche"

table2[1].dt.mois	est égal à 12
table2[0].dt.annee	est égal à 1994

Accès au membre d'une structure depuis un pointeur : **opérateur →**

Exemples :

*struct salarie * p = table; /* soit &table[0] */*

p →age	est égal à 31
p →nom	est l'adresse de début du tableau contenant "dupont"
p++;	p est égale à table + 1 soit &table[1]
p →salaire	est égal à 12700

La notation	**p→membre**	équivaut à	**(*p).membre**

Pour changer les valeurs d'une structure :

struct salarie x = {"Dupont", 31, 9900};

x.age = 39;

Attention:

Pour les chaînes ne pas faire

x.nom = "Durand"; ==========> FAUX

x.nom est soit un tableau soit un **pointeur constant**

Message compilateur : **lvalue required**

ou LHS (left hand side)

ou non modifiable.

Utiliser la fonction **strcpy** de la librairie standard :

strcpy(x.nom, "Durand");

5.6 AUTRES OPÉRATEURS

L'opérateur & retournant l'adresse d'une variable est autorisé :

```
struct salarie  x = {"dupont", 31, 9900};
struct salarie * p;
p = &x;
```

L'affectation d'une structure par une structure de même type est autorisée :

```
struct salarie  x = {"dupont", 31, 9900};
struct salarie y;
y = x;            /* y vaut  {"dupont", 31, 9900} */
```

L'affectation est réalisée par une **recopie superficielle** membre à membre des champs de la structure.

Attention :

```
struct date2  d1 = {"Dimanche", 01, 05, 1994};
struct date2  d2;
d2 = d1;                  /* d2 vaut {"Dimanche", 01, 05, 1994};
```

Mais le champ nom de la variable d2 est un pointeur : c'est sa valeur qui est recopiée. Par suite, les 2 pointeurs nom des variables d1 et d2 pointent sur la même chaîne de caractères. On dit qu'il y a partage de données.

Si le nom de la date d1 change alors ce changement sera répercuté sur le nom de la date d2.

Le remède est d'écrire une fonction de recopie profonde.

La nouvelle zone pointée, ici le nom de la date d2, doit être allouée dynamiquement, puis le nom de la date d1 y est recopié.

Ainsi chaque date aura son propre nom et il n'y aura pas partage de données.

Une structure ou une adresse de structure peut être passée en argument d'une fonction. Une fonction peut retourner une structure ou une adresse de structure (allouée dynamiquement dans la fonction, mais surtout pas allouée statiquement sur la pile, problème de la référence pendante).

Les tests de comparaison == et != ne sont pas autorisés.

Il faut écrire des **fonctions de comparaison membre à membre** des structures .

Exemple complet : structure personne

```
/* définition du type struct personne */
struct personne
{
        char nom[30];
        char prenom[40];
        int age;
        char sexe;
};
```

```
/* déclarations prototypes des fonctions */
extern void saisie(struct personne *);
extern void affiche(const struct personne *);
```

```
int main()
{
        struct personne toto;
        struct personne * p;


        toto.age = 13;

        printf("nom =  "); scanf("%s", toto.nom);

        printf("prenom =  "); scanf("%s", toto.prenom);

        printf("sexe =  "); scanf(" %c";  &(toto.sexe));

        affiche(&toto);

        p = &toto;    /* *p vaut toto */

        p->age = 14; /* (*p).age = 13; */

        printf("nom =  "); scanf("%s", p->nom);

        affiche(p);
        saisie(p);
        affiche(p);
        return 0;
}
```

```
void affiche(const struct personne * p)
{
	/* p->age = 150; NE COMPILE PAS! */
	printf("%s \t %s \t %d \t %c \n", p->nom, p->prenom, p->age, p->sexe);
}


void saisie(struct personne * p)
{
	printf("age = "); scanf(" %d", &(p->age));

	printf("nom = "); scanf("%s", &(p->nom[0]));

	printf("prenom = "); scanf("%s", p->prenom);

	printf("sexe = "); scanf(" %c", &(p->sexe));
}
```

STRUCTURE IMBRIQUEE

struct date
{
int jour;
int mois;
int annee;
};
	struct personne
	{
	char nom[30];
	char prenom[40];
	int age;
	char sexe;
	struct date dateNaissance;
	};

```
int  main ()
{
	struct personne toto;
	struct personne * p;
	p = &toto;

	toto.dateNaissance.jour = 10;
	toto.dateNaissance.mois = 09;
	toto.dateNaissance.annee = 2010;
	(p->dateNaissance).jour = 11;
	saisie(&toto);
	affiche(p);
	return 0;
}
```

```
void affiche(const  struct  personne * p)
{
	/* p->age = 150; INTERDIT*/
	printf("%s \t %s \t %d \t %c \n", p->nom, p->prenom,
	                                  p->age, p->sexe);
	printf("%d \t %d \t %d \n", p->dateNaissance.jour,
	                            p->dateNaissance.mois,
	                            p->dateNaissance.annee);
}


void saisie(struct  personne * p)
{
	printf("age = "); scanf(" %d", &(p->age));
	printf("nom = "); scanf("%s", &(p->nom[0]));
	printf("prenom = "); scanf("%s", p->prenom);
	printf("sexe = "); scanf(" %c", &(p->sexe));
	printf("jour = "); scanf(" %d", &(p->dateNaissance.jour));
	printf("mois = "); scanf(" %d", &(p->dateNaissance.mois));
	printf("annee = "); scanf("%d", &(p->dateNaissance.annee));
}
```

TABLEAU DE STRUCTURES

```c
#include <stdio.h>

struct personne
{
	char nom[30];
	char prenom[40];
	int age;
	char sexe;
};

extern void saisie(struct personne *);
extern void affiche(const struct personne *);
extern void saisie_tableau(struct personne * t,int taille);
extern void affiche_tableau(const struct personne * t,int taille);


int main()
{
	struct personne toto, titi;

	struct personne tab[] = {toto, titi}; /* copie des variables toto et titi */

	saisie_tableau(tab,sizeof(tab)/sizeof(tab[0]));

	affiche_tableau(tab,sizeof(tab)/sizeof(tab[0]));

	return 0;
}

void affiche(const struct personne * p)
{
	printf(" %s \t %s \t %d \t %c \n", p->nom, p->prenom, p->age, p->sexe);
}

void saisie(struct personne * p)
{
	printf("age = ");scanf(" %d", &(p->age));

	printf("nom = "); scanf("%s", &(p->nom[0]));

	printf("prenom = "); scanf("%s", p->prenom);

	printf("sexe = "); scanf(" %c", &(p->sexe));
}

void affiche_tableau(const struct personne * t, int taille)
{
	struct personne * p;

	for (p = t; p < t + taille; p++)
		affiche(p);
}
```

```
void saisie_tableau(struct personne * t,int taille)
{
	struct personne * p;

	for (p = t; p < t + taille; p++)
		saisie(p);

}
```

TABLEAU DE STRUCTURES IMBRIQUEES

```
#include <stdio.h>


struct date
{
	int jour;
	int mois;
	int annee;
};


struct personne
{
	char nom[30];
	char prenom[40];
	int age;
	char sexe;
	struct date dateNaissance;
};

extern void saisie(struct personne *);
extern void affiche(const struct personne *);
extern void saisie_tableau(struct personne * t,int taille);
extern void affiche_tableau(const struct personne * t,int taille);

int main()
{
	struct personne toto, titi;

	struct personne tab[] = {toto, titi};  /* copie des variables toto et titi */

	saisie_tableau(tab,sizeof(tab)/sizeof(tab[0]));

	affiche_tableau(tab,sizeof(tab)/sizeof(tab[0]));

	return 0;
}
```

```c
void saisie (struct personne * p)
{
	printf("age = "); scanf(" %d", &(p->age));

	printf("nom = "); scanf("%s", &(p->nom[0));

	printf("prenom = "); scanf("%s", p->prenom);

	printf("sexe = "); scanf(" %c", &(p->sexe));

	printf("jour = "); scanf(" %d", &(p->dateNaissance.jour));

	printf("mois = "); scanf(" %d", &(p->dateNaissance.mois));

	printf("annee = "); scanf(" %d", &(p->dateNaissance.annee));
}


void affiche (const struct personne * p)
{
	/* p->age = 150; NE COMPILE PAS */

	printf("%s \t %s \t %d \t %c \n", p->nom, p->prenom, p->age, p->sexe);

	printf("%d \t %d \t %d \n", p->dateNaissance.jour, p->dateNaissance.mois,
                                                        p->dateNaissance.annee);
}


void affiche_tableau(const struct personne * t, int taille)
{
	struct personne * p;

	for (p = t; p < t + taille; p++)
		affiche(p);
}

void saisie_tableau(struct personne * t, int taille)
{
	struct personne * p;

	for (p = t; p < t + taille; p++)
		saisie(p);

}
```

5.7 LES STRUCTURES RECURSIVES

Une structure peut contenir des membres qui sont des "pointeurs sur elle-même".
On peut ainsi représenter des listes chaînées (pointeurs sur la structure suivante et la précédente), des arborescences (arbres, arbres binaires, graphes ...) ...

Exemple :

```
struct fiche {
        char nom [20];
        int age;
        double salaire;
        struct fiche * precedente;
        struct fiche * suivante;
 };
struct fiche a = {"Andre", 28, 2000.80};
struct fiche  j = {"Jean", 27, 3000.25};
struct fiche  p = {"Pierre", 32, 4000};
```

Adresse	Nom	Age	Salaire	Precedente	Suivante
A20	Andre	28	2000.80	0	A60
A60	Jean	27	3000.25	A20	A100
A100	Pierre	32	4000	A60	0

Le pointeur dont la valeur est 0, encore notée NULL, ne correspond à aucune valeur licite. Déférencer un pointeur nul conduit à une erreur d'exécution (segmentation fault).

Instructions réalisant le chaînage logique ci-dessus :

a.precedente = NULL;

a.suivante = &j;

j.precedente = &a;

j.suivante = &p;

p.precedente = &j;

p.suivante = NULL;

6 CHAMP DE BITS DANS LES STUCTURES

Un objet de type char, int ou unsigned int peut être vu comme une structure dont les membres sont des suites de bits consécutives.

Cela est intéressant pour les logiciels de type 'drivers' dans lesquels on a à lire des parties de mots d'états qui occupent quelques bits.

Exemple :

```
struct
{
        unsigned  bit_faible : 1,
                  car_ascii : 7,
                  bit_centre : 1,
                           : 3,     /* padding */
                  demi_octet : 4;
} r = {0, 'a', 1, 0x0f};       /* 'a' == 0x61 */
```

```
struct
{
   unsigned bit_faible : 1;
   unsigned car_ascii : 7;
   unsigned bit_centre : 1;
   unsigned : 3;              /* padding */
   unsigned demi_octet : 4;
} r = {0, 'a', 1, 0x0f};                  /* 'a' == 0x61 */
```

7 LES UNIONS

Une union est une structure dans laquelle tous les membres sont alloués à partir de la même adresse, qui est celle de l'union elle-même. Celle-ci a donc la taille de son membre le plus long (arrondi au pair).

Exemple :

Interprétation d'une zone mémoire de 8 octets selon différents formats

```
struct {
        int type;
        union udef {
                struct {int a; int b;} entiers;
                long l;
                double f;
                char * s;
        } u;
} su;
```

La syntaxe est analogue aux structures : *su.type, su.u.entiers, su.u.entiers.a, su.u.l*

8 LES ÉNUMÉRATIONS

8.1 DÉCLARATION

Cette déclaration permet de définir une liste de constantes par énumération et définit un nouveau type.

Chaque constante correspond en fait à une valeur entière. Par défaut, la première vaut 0, la deuxième 1, etc...

Exemple :

```
enum semaine
{
        dimanche, lundi, mardi, mercredi, jeudi, vendredi, samedi
};
```

enum semaine est un nouveau type.

8.2 INITIALISATION

On peut changer la valeur des constantes en les initialisant à partir d'une certaine valeur.

Exemple : *enum semaine {*

dimanche = 7, lundi = 1, mardi, mercredi, jeudi, vendredi, samedi};

la valeur 7 est associée à dimanche, 1 à lundi, 2 à mardi, ..., et 6 à samedi.

8.3 DÉCLARATION DE VARIABLE

Exemple : *enum semaine jour;*

enum semaine jourJ = jeudi;

Utilisation :

- affectation **:** *jour = lundi;* ou *jour = 0;*
- test : if (jour == samedi) ou *if (jour == 6)*

Les notations entières ou sous forme d'une constante par énumération sont donc équivalentes.

Attention :

Le compilateur ne vérifie pas que la valeur affectée fait partie de l'ensemble des valeurs de l'énumération.

jour = 124;

génére l'erreur : warning : énumération type clash, operator = mais fonctionne bien à l'éxécution.

CHAPITRE 6 : FONCTIONS

1 PASSAGE DE PARAMÈTRES À UNE FONCTION

En C, le **passage** de paramètres se fait toujours "**par valeur** " :

les **paramètres effectifs** (***arguments***) sont recopiés dans les **paramètres formels** (***paramètres***) et ce dans une zone de travail locale à la fonction et allouée sur la **pile interne** du langage C propre à chaque utilisateur.

La fonction ne modifie que cette zone locale et donc en aucune façon les variables (arguments) de la fonction appelante.

Cette zone locale contenant les paramètres est allouée automatiquement sur la pile à l'entrée dans la fonction et est désallouée automatiquement à la sortie de la fonction.

Les paramètres et les variables locales d'une fonction sont de classe de stockage **auto** (***gestion automatique***).

Pour pouvoir modifier une variable de la fonction appelante indirectement, il faut passer son **adresse en paramètre**.

1 pas de modification	**2 modification**
void f ***(int x)*** *{* `        `***x*** *= 10;* *}* *int main ()* *{* `        `*int a = 8;* `        `*f(**a**);* `        `*return 0;* *}*	*void f* ***(int * p)*** *{* `        `****p*** *= 10;* *}* *int main ()* *{* `        `*int a = 8;* `        `*f(**&a**);* `        `*return 0;* *}*
a vaut 8	**a vaut 10**

Un autre exemple, celui classique de la fonction **swap**, a été développé dans le chapitre **POINTEURS**.

2 DÉCLARATION PROTOTYPE

Afin que le compilateur vérifie le typage, tout appel de fonction doit être précédé d'une déclaration prototype de la fonction.

Plus précisément, le compilateur vérifie l'adéquation entre le type et le nombre des arguments de l'appel de fonction et le type et le nombre des paramètres de la déclaration prototype.

Vocabulaire :

Appel : argument ou paramètre effectif

Déclaration prototype : paramètre ou paramètre formel

La déclaration protype peut être :

1) **globale** : elle figure alors en tête d'un fichier .c ou dans un fichier d'entête .h à inclure dans le fichier .c, où l'appel est fait
2) **locale** à une fonction.

Le **mot-clé extern** est facultatif, mais conseillé par la plupart des experts.

Exemple :

```
extern double cos (double) ;

void f ()
{
        double x;
 /* déclaration prototype locale, usage peu fréquent */
        extern double calcul (int);
        x = calcul (5);
        y = cos(x);
        x = calcul();       /* NE COMPILE PAS */
        y = cos(x, 10);     /* NE COMPILE PAS */
        ...
}
```

```
/* cos : déclaration prototype dans math.h */
#include <math.h>
/* calcul : déclaration prototype dans calcul.h */
#include "calcul.h"

void f ()
{
        double x;
        x = calcul (5);
        y = cos(x);
        x = calcul();       /* NE COMPILE PAS */
        y = cos(x, 10);     /* NE COMPILE PAS */
        ...
}
```

CHAPITRE 7 :
CONVERSIONS DE TYPES
TYPES COMPLEXES

1 CONVERSION DE TYPES

1.1 CONVERSION IMPLICITE

Le langage C effectue de lui-même des conversions de types implicites.

La conversion implicite s'effectue :

- lorsqu'une valeur est affectée à une variable de type différent
- en présence d'opérateurs qui convertissent leurs opérandes avant d'effectuer l'opération
- lors du passage de valeurs comme paramètres à une fonction.

D'une manière générale, il est conseillé de ne pas trop mélanger des variables de types différents dans une même expression.

1.2 CONVERSION EXPLICITE

Une conversion explicite permet de convertir le type d'une expression ou d'une variable.

exemple :

```
float f ;
int i = 8 , j = 7;
f = (float) i / (float) j;
```

La conversion explicite est aussi utilisée pour convertir les pointeurs. En effet, le compilateur n'accepte pas l'affectation d'un pointeur dans un pointeur de type différent.

exemple :

```
int * p;
p = (int *) malloc(5 * sizeof(int)); /* tableau dynamique de 5 entiers */
```

2 LES TYPES COMPLEXES

Le mot clé **typedef** permet de définir un identificateur synonyme d'un type déjà existant.

Cet identificateur pourra ensuite être utilisé comme un nom de type lors de déclarations.

Il est parfois d'usage de mettre les types synonymes en majuscules.

2.1 DÉFINITION DE TYPES SYNONYMES

typedef char BOOLEEN;	BOOLEEN	est synonyme de	char
typedef enum booleen BOOLEAN;	BOOLEAN	est synonyme de	enum booleen
	avec : enum booleen { FALSE = 0, TRUE = 1 };		
typedef unsigned short USHORT;	USHORT	est synonyme de	unsigned short
typedef short NOTES[20];	NOTES	est synonyme de	short [20]

Utilisation dans des définitions de variables :

BOOLEEN status ;	au lieu de	char status;
BOOLEAN bool;	au lieu de	enum booleen ;
USHORT n ;	au lieu de	unsigned short n;
NOTES table ;	au lieu de	short table [20];

Pour les structures :

struct personne
{
char nom[30];
char prenom[40];
int age;
char sexe;
};
typedef struct personne personne;
	OU
	typedef struct personne
	{
	char nom[30];
	char prenom[40];
	int age;
	char sexe;
	}**personne**;

`extern void saisie(struct personne *);` `extern void affiche(const struct personne *);`	`OU extern void saisie(personne *);` `OU extern void affiche(const personne *);`

Le nouveau type personne est un synonyme du type struct personne.

2.2 CONSTRUCTION DE TYPES COMPLEXES

Il est possible de définir des variables de types très complexes en C.

La méthode est la suivante pour pouvoir comprendre une déclaration comportant un type complexe :

Il faut partir de l'intérieur (parenthèse la plus imbriquée) vers l'extérieur en allant alternativement à droite puis à gauche.

exemple :

char * (* (*var) ( )) [10] ;

char * (* (* var) ( )) [10] ;

6 4 2 1 3 5

1. L'identificateur var est 2. un pointeur de 3. fonction retournant
4. un pointeur vers 5. un tableau de 10 6. pointeurs de caractères

Si vous savez lire cette déclaration, vous êtes maintenant capable de la définir.

Dans certains cas, les parenthèses sont nécessaires pour modifier l'ordre d'évaluation.

int * var [10] ; tableau de 10 pointeurs d'entiers,

int (* var) [ ] ; pointeur sur un tableau d'entiers.

int * var ( ) ; une fonction retournant un pointeur d' entier

int * (*var) ( ) ; un pointeur sur une fonction, cette fonction retourne un pointeur d'entier.

int (* var [ ]) ( ) ; un tableau de pointeurs vers des fonctions retournant un entier.

int * (* var [ ]) ( ) ; un tableau de pointeurs vers des fonctions retournant un pointeur vers un entier.

int ** var [ ] ( ) ;

signifierait que var est un tableau de fonctions retournant un pointeur vers un pointeur vers un entier ; ceci est impossible, une fonction n'étant pas un objet de taille connue, elle ne peut donc pas être combinée au sein d'un objet dérivé de type tableau.

CHAPITRE 8 :
CLASSES D'ALLOCATION DES VARIABLES

La classe d'allocation d'une variable détermine :

- l'endroit où va être allouée cette variable,
- le moment où elle va être allouée (à la compilation, à l'exécution).
- son mode d'initialisation

1 ENVIRONNEMENT D'UN PROCESSUS

Pour comprendre la notion de classe, il faut connaître les différentes zones mémoire d'un processus, image mémoire d'unc programme C en cours d'exécution :

DATA : zone de données allouées à la compilation
(variables globales et variables locales static)

TEXT : zone des instructions

PILE : zone des données dynamiques
(paramètres et variables locales des fonctions)

TAS : zone d'allocation dynamique (malloc)

2 CLASSE REGISTER

```
register char c ;          register k ;      / * sous entendu int * /
register char * s ;
```

Types permis : char, short, int et leur version "unsigned" et les pointeurs.

Le mot clé **register** suggère au compilateur de stocker une variable locale dans un registre. Il n'est pas possible de savoir :

- si cette suggestion a été ou non prise en compte,
- et si oui, dans quel registre se trouve la variable.

L'opérateur & est interdit sur une variable register. Quelques registers biens choisis peuvent sensiblement améliorer les temps d'éxécution.

3 CLASSE AUTOMATIQUE

Classe par défaut des variables locales (mot-clé obsolète **auto**)

Les variables automatiques sont allouées sur la pile.

Cette allocation est dynamique, et rien ne garantit que, d'un appel sur l'autre à une même fonction, une variable automatique sera allouée au même endroit.

Par défaut, ces variables ne sont pas initialisées.

Leur durée de vie est égale au temps d'exécution du bloc de la fonction dans laquelle elles sont déclarées.

4 CLASSE STATIQUE (LOCALE)

static int compteur ;

Les variables locales statiques ne sont visibles que de la fonction dans laquelle elles sont déclarées.

Cependant, elles ne sont pas allouées sur la pile, mais dans un segment de données allouées de façon permanente pendant toute la durée d'exécution.

Elles ont donc une localisation figée. Ainsi, leur valeur se conserve entre deux appels consécutifs à la fonction.

Lorsqu'une variable statique est initialisée en même temps qu'elle est déclarée, cette initialisation n'a lieu qu'une fois, lors du premier appel à la fonction.

Les locales statiques sont initialisées par défaut à 0.

Exemple :

```
void f ( void  ){
        int k  = 0 ;                    / * k initialisé à 0 à chaque appel * /
        static int compteur = 0 ;       / * initialisation au 1er appel * /
        compteur++ ;                    / * avant la sortie du programme, compteur sera
                                          égal au nombre d'appels de la fonction  f * /
        ...
}
```

5 VARIABLE GLOBALE SIMPLE

Il n'y a pas de mot réservé pour définir une variable globale à un fichier. Toute variable déclarée en dehors du corps des fonctions est globale.

Les globales sont initialisées par défaut à 0.

Toutefois, il est très vivement **déconseillé de programmer avec des variables globales** (problèmes de lisibilité, de portabilité, de réutilisation).

exemple :

fichier main.c

```
int tab [100];
char buf [512];
long * 11, * 12;

int main ()
{
        ...
        return 0;
}
```

6 VARIABLE EXTERNE

Une déclaration de variable précédée du mot clé **extern** indique au compilateur que la variable a été déclarée dans un autre fichier.

La visibilité de la variable se trouve ainsi étendue au fichier actuel (ou seulement à une fonction de ce fichier).

exemple :

fichier fonc.c

```
extern int tab[100] ;
```

tab déclaré et alloué dans **main.c**, devient visible des fonctions f et g.

```
void f ( )
{
        tab[5] = 78;
}

void g ( )
{
        extern long *11 ;       /* seule g peut manipuler 11,
                                   l1 est défini et alloué dans main.c */
        ...
}
```

Les références externes non résolues (définitions oubliées) sont listées lors de l'édition de liens.

7 VARIABLE GLOBALE STATIQUE ET FONCTION STATIQUE

Une variable globale précédée du mot clé **static** reste globale au fichier où elle est définie, mais sa visibilité ne peut être étendue aux autres fichiers de l'application.

Une fonction peut être déclarée static.

exemple :

fichier main.c :

```
static int tab1[100] ;
long tab2 [5] ;
static int f (int );

int main ( )
{
        int i;
        i = f (tab1[5]) ;
       return 0;
}

static int f ( int x )
{
        tab2[6] = 25000;
}
```

fichier fonc.c :

```
extern long  tab2 [] ;
void g ()
{
        int i ;
        i = f (4) ; /* erreur de compilation */
        tab2[0] = 11;
        ...
}
```

CHAPITRE 9 : TABLEAUX A DEUX DIMENSIONS

I VISION CLASSIQUE 2D D'UN TABLEAU 2D

Un tableau à deux dimensions permet d'accéder à des informations comme une image (en donnant pour chaque point (i; j) une valeur comme la couleur).

L'élément en ligne i et colonne j d'un tableau tab à deux dimensions est noté tab[i][j]

Du point de vue mathématique, c'est une matrice à deux dimensions.

Voici par exemple comment on initialise un tableau 4 x 3 à 0 :

```
int main ()
{
        int tab[4][3];
        int i, j;

        for (i = 0; i < 4, i++) {
                for (j = 0; j < 3; j++) {
                        tab[i][j] = 0;
                }
        }

        return 0;
}
```

Pour passer un tableau à deux dimensions en paramètre, on doit donner explicitement la deuxième dimension.

Exemple : lecture d'un tableau à une et deux dimensions.

```
void lire_tab1D(int a[], int taille)
{
   int i;
   for (i = 0; i  < taille; i++) {
      printf("a[%d] = ", i);
      scanf(" %d", &a[i]);
   }
}

/** uniquement pour les tableaux 2D de 4
colonnes */
void lire_tab2D_4COLS(int a[][4], int
nb_lignes)
{
   int i, j;
   for (i = 0; i < nb_lignes; i++) {
      for(j = 0; j < 4; j++) {
         printf("a[%d][%d] = ", i, j);
         scanf(" %d", &a[i][j]);
      }
   }
}
```

Dans la fonction lire_tab2D, l'expression a[i] designe la ligne d'indice i du tableau. On peut donc aussi écrire :

```
/** uniquement pour les tableaux 2D de 4
colonnes */
void lire_tab2D_4COLS_bis(int a[][4], int
nb_lignes)
{
   int i;
   for (i = 0; i < nb_lignes; i++) {
      printf("ligne %d : \n", i);
      lire_tab1D(a[i], 4);
   }
}
```

Inconvénient majeur : il y aura autant de fonctions que de valeurs différentes du nombre de colonnes.

```
void lire_tab2D_3COLS(int a[][3], int nb_lignes);
void lire_tab2D_4COLS(int a[][4], int nb_lignes);
void lire_tab2D_5COLS(int a[][5], int nb_lignes);
....
```

II VISION ORIGINALE 1D D'UN TABLEAU 2D

Un tableau à plusieurs dimensions est organisé en mémoire comme un tableau à une seule dimension.

Si on range les éléments par ligne alors **tab[i][j]** est rangé en place **i * m + j** (pour un tableau de **n lignes** et de **m colonnes).**

```
/** pour tous les tableaux 2D
 * de dim1 lignes et dim2 colonnes
*/
void ecrire_tab2D(int * m, int dim1, int dim2)
{
   int i, j;

   for (i = 0; i < dim1; i++) {
      for (j = 0; j < dim2; j++) {
         printf("%d ", m[dim2 * i  + j]);
      }
      printf("\n");
   }
   printf("\n");
}
```

```
/** pour tous les tableaux 2D carrés */
void ecrire_tab2D_Carre(int * m, int dim)
{
   ecrire_tab2D(m, dim, dim);
}
```

Pour tout tableau 2D de dim1 x dim2 :

tab[i][j] <=> tab[dim2 * i + j]

Pour tout tableau 3D de dim1 x dim2 x dim3 :

tab[i][j][k] <=> tab[dim2 *dim3 * i + dim2 * j + k]

CHAPITRE 10 :
PASSAGE DE PARAMETRES A LA FONCTION MAIN

Tout exécutable C doit comporter une fonction **main** dont le nom est réservé et qui est le point d'entrée de l'application, du point de vue du système d'exploitation.

1 EXÉCUTABLE NE RECEVANT AUCUN ARGUMENT

int main ()
{
…
return 0;
}
	void main ()
	{
	...
	}
		void main ()
		{
		int status
		…
		exit(status);
		}

*Dans le cadre de la programmation système, la variable **status** est récupérée par le processus appelant. Si status vaut 0 le processus s'est bien déroulé, sinon sa valeur non nulle indique un type d'erreur.*

Une application comportant une telle fonction main se lancera sous shell par son seul nom exécutable :

par défaut : $ **a.out**

si renommage : $ **myexec**

2) EXÉCUTABLE RECEVANT DES ARGUMENTS

rappel : les 3 notations suivantes sont équivalentes

char ** argv, char * argv[], char argv[][]

int main (int argc , char ** argv) { ***…*** ***}***	***int main (int argc , char * argv[]) {*** ***…*** ***}***
int main (int argc , char argv[][]) { ***…*** ***}***	***int main (int argc , char ** argv) {*** *int status;* *…* ***exit(status);*** ***}***
int main (int argc , char * argv[]) { *int status;* *…* ***exit(status);*** ***}***	***int main (int argc , char argv[][]) {*** *int status;* *…* ***exit(status);*** ***}***

Une application comportant une telle fonction main se lancera sous shell par son nom exécutable suivi des arguments :

par défaut : $ **a.out arg1 ... argN**

si renommage : $ **myexec arg1 ... argN**

La signification des paramètres de main est la suivante :

argc	**nombre de paramètres**. Le nom du programme est considéré comme un paramètre.
argv	**tableau de pointeurs de chaînes de caractères**. **Chaque chaîne est un paramètre.**

Remarque: Les paramètres de la fonction main sont nommés conventionnellement arg et argv. argc pour "argument count" et argv pour "arguments en nombre variable".

Exemple :

$ gcc - 0 myexec file.c

argc vaut 4		
argv [0] ou *argv	pointe sur le nom de l'exécutable	"cc"
argv [1] ou *(argv + 1)	pointe sur le paramètre	"-O"
argv [2] ou *(argv + 2)	pointe sur le paramètre	"myexec"
argv [3] ou *(argv + 3)	pointe sur le paramètre	"file.c"
argv [4] ou *(argv + 4)	est le pointeur	NULL

La mise en forme des arguments, leur allocation mémoire et la mise en place des pointeurs est prise en charge par le système d'exploitation.

Pour parcourir la liste des paramètres, on peut utiliser :

- le nombre de paramètres argc
- le fait que le tableau de paramètres se termine par un pointeur NULL (la sentinelle).

Les paramètres sont transmis sous la forme de chaînes de caractères. Si un paramètre est utilisé dans le programme comme un numérique, il devra être converti en numérique.

La librairie standard fournit des **fonctions de conversion**.

Les déclarations prototypes des fonctions de conversion figurent dans le fichier **stdlib.h** :

*double **atof** (const char * s);*

*int **atoi** (const char * s);*

*long **atol** (const char * s);*

exemples :

```
$ a.out 5

int main ( int argc, char * argv [] )
{
        int x;
        if (argc >= 1)
                x = atoi(argv[1]);
        ...
        return 0;
}

x vaut 5
```

```
#include <stdio.h>
int main(int argc, char * argv[])
{
   int i;

   /* Affiche le nom du programme : */
   printf("Nom du programme : %s \n", argv[0]);

   /* Affiche la ligne de commande : */
   for (i = 1;  i < argc;  i++)
      printf("Argument %d : %s \n", i,  argv[i]);
   return 0;
}
```

```
#include <stdio.h>
int main(int argc, char * argv[])
{
   /* Affiche le nom du programme : */
   printf("Nom du programme : %s \n", *argv);

   /* Affiche la ligne de commande : */
   for (;  *argc != NULL;  argv++)
      printf("Argument: %s \n", *argv);
   return 0;
}
```

```
#include <stdio.h>
int main(int argc, char * argv[])
{
   /* Affiche le nom du programme : */
   printf("Nom du programme : %s \n", *argv);

   /* Affiche la ligne de commande : */
   for (i = 1;  i < argc;  i++)
      printf("Argument %d : %s \n", i,  *(argv + i));
   return 0;
}
```

CHAPITRE 11 :
ALLOCATION DYNAMIQUE

1 ALLOCATION DYNAMIQUE DE MEMOIRE : FONCTION MALLOC

La définition du type **size_t** dans les fichiers **stdlib.h, stddef.h**:

typedef unsigned int size_t;

La déclaration prototype de la fonction **malloc** se trouve dans le fichier **stdlib.h** :

void * malloc (size_t n);

La fonction malloc permet de demander une allocation dynamique de mémoire d'une zone de taille **n octets** lors de l'exécution.

Les octets sont disponibles dans une zone mémoire appelée le **TAS** (**HEAP** en anglais).

Un pointeur générique de type void * sur la zone allouée est retourné.

Si la requête n'a pu être satisfaite, le pointeur retourné vaut **NULL**.

La requête échoue si le tas est saturé : plus d'octets libres.

exemple :

```
int  * pi ;
pi = (int *) malloc (10 * sizeof (int)) ; // tableau dynamique de 10 entiers
if ( pi == NULL )
{
        printf("plus de place dans le tas \n");
        exit(2);
}
…
```

2 DESALLOCATION DE MEMOIRE : FREE

La déclaration prototype de la fonction free se trouve dans le fichier stdlib.h :

void free (void * p);

free désalloue une zone allouée nécessairement par **malloc** et identifiée par le pointeur p passé en paramètre.
Si le pointeur p vaut NULL, la fonction est sans effet.
Si le pointeur p ne correspond pas à une zone mémoire allouée dynamiquement alors le comportement du programme peut devenir incohérent.

Attention : le pointeur p n'est pas remis à NULL. Il est prudent de le faire afin de ne pas être amené à utiliser un pointeur obsolète.

Ne pas désallouer des zones mémoires dont on n'a plus besoin est une très mauvaise pratique. Le tas risque d'être saturé.
Le **logiciel valgrind** permet de détecter des fuites mémoires.

Exemple :

```
extern char * allocation ( size_t n);

void main ()
{
        char * p = NULL;
        p =  allocation (100);
        if ( p == NULL )
        {
            printf(("plus de place dans le tas! ");
            exit(2);
         }
        ...
        free(p);
        p = NULL;
        exit(0);
}

char * allocation ( size_t n)
{
        char * s = NULL;
        s =  (char *) malloc (n * sizeof(char));
        return s;
}
```

3 DECLARATIONS PROTOTYPES DES FONCTIONS D'ALLOCATION DYNAMIQUE

Les fonctions permettant de faire de l'allocation dynamique de memoire sont :

malloc et **calloc** pour allouer un bloc memoire (initialise avec des zéros si calloc),

realloc pour augmenter ou diminuer sa taille

free pour le libérer.

Leurs déclarations prototypes se trouvent dans le chier en-tête stdlib.h.

void * malloc(size_t nb_octets)

void * calloc(size_t nb_elements, size_t taille_elt)

void * realloc(void *pointeur, size_t nb_octets)

void free(void *pointeur)

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

Reallocate memory block

Changes the size of the memory block pointed to by *ptr*.

The function may move the memory block to a new location (whose address is returned by the function).

The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new *size* is larger, the value of the newly allocated portion is indeterminate.

In case that *ptr* is a null pointer, the function behaves like malloc, assigning a new block of *size* bytes and returning a pointer to its beginning.

If the function fails to allocate the requested block of memory, a null pointer is returned, and the memory block pointed to by argument *ptr* is not deallocated (it is still valid, and with its contents unchanged).

Parameters

ptr

Pointer to a memory block previously allocated with malloc, calloc or realloc.
Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if malloc was called).

size

New size for the memory block, in bytes.
size_t is an unsigned integral type.

4 ALLOCATION DYNAMIQUE D'UN TABLEAU MULTIDIMENSIONNEL

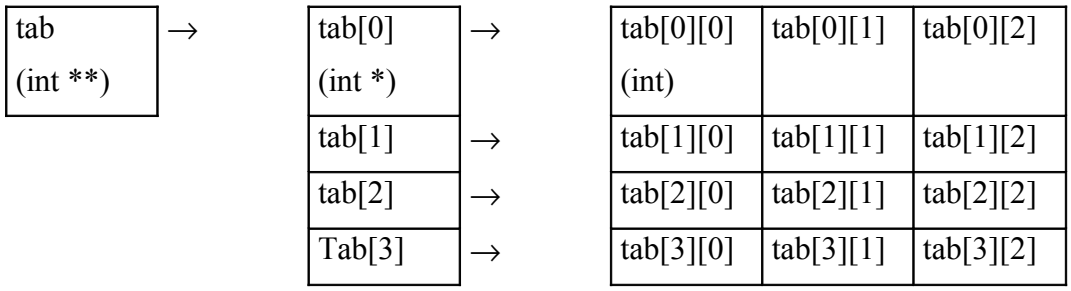
Tableau dynamique d'entiers de 4 lignes et 3 colonnes.

L'allocation dynamique d'un tel tableau est réalisée en 3 étapes :

déclaration d'un pointeur sur pointeur d'entiers

allocation d'un tableau de 4 pointeurs sur entiers

allocation des 4 tableaux de 3 entiers.




```
int i;
 /* déclaration d'un pointeur sur pointeur d'entiers */
int ** tab;
/* allocation d'un tableau de 4 pointeurs sur entiers */
tab = (int **)  malloc (4 * sizeof(int *));
if (tab == NULL) {
        printf(("plus de place dans le tas! ");
        exit(2);
}
/* allocation des 4 tableaux de 3 entiers */
for (i = 0; i < 4; i++) {
        tab[i] = (int *) malloc (3 * sizeof(int));
        if (tab[i] == NULL) {
                printf(("plus de place dans le tas!");
                exit(2);
        }
}
```

5 EXEMPLES

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define SIZE 100

typedef struct
{
   int nb;
   float * ptr;
} ArrayFloat;

const   int     INCREMENT = 3 ;
typedef double * PtrDouble;

extern void exemple1();
extern void exemple2();
extern void exemple3();

extern void resizeOld(PtrDouble * ptab, int * pnbElmts);
extern void resize(PtrDouble * ptab, int * pnbElmts);

int main()
{
   exemple1();
   exemple2();
   exemple3();
   return 0;
}
```

```
void exemple1()
{
   ArrayFloat * p;
   int i;

   p = (ArrayFloat *) calloc(SIZE, sizeof(ArrayFloat));
   if (p == NULL)
   {
      fprintf( stderr, "Erreur a l'allocation\n\n" );
      exit(1);
   }

   for (i = 0; i < SIZE; i++)
   {
      p[i].nb = SIZE;
      p[i].ptr = (float *) malloc(p[i].nb * sizeof(float));
      if (p[i].ptr == NULL)
      {
         fprintf( stderr, "Erreur a l'allocation\n\n" );
         exit(1);
      }

      p[i].ptr[i] = 3.14159f;
   }

   for (i = 0; i < SIZE; i++)
   {
      printf("%f\t", p[i].ptr[i]);
   }
   printf("\n");
```

```
   for (i = 0; i < SIZE; i++)
      free(p[i].ptr);
   free(p);
}

void exemple2()
{
   char ** ptr;
   char buffer[BUFSIZ];
   int nb = 0;
   int i;

   for ( ; ; )
   {
      printf("Entrer une chaîne (fin pour terminer): " );
      scanf("%s", buffer );
      if (!strcmp(buffer, "fin"))
         break;
      ptr = (char **) realloc(ptr, ++nb * sizeof(char *));
      if (ptr == NULL)
      {
         fprintf( stderr, "Erreur a l'allocation\n\n" );
         exit(1);
      }

      ptr[nb-1] = (char *) malloc((strlen(buffer)+1) * sizeof(char));
      if (ptr[nb-1] == NULL)
      {
         fprintf( stderr, "Erreur a l'allocation\n\n" );
```

```
            exit(1);
        }
        strcpy(ptr[nb-1], buffer);
    }

    for (i = 0; i < nb; i++)
    {
        printf("%s\t", ptr[i]);
    }
    printf("\n");

    for (i = 0; i < nb; i++)
    {
        free(ptr[i]);
    }
    free(ptr);
}
```

```
void exemple3()
{
   int i, j, length, n;
   PtrDouble t;

   printf("dimension de votre tableau: ");
   scanf("%d", &length);
   t = (PtrDouble) malloc(length * sizeof(double));
   assert(t);

   i = 0;
   do
   {
      printf("entier %d (fin < 0 : ", i+1);
      scanf("%d", &n);
      if (i == length - 1)
      {
         printf("tableau saturé, on l'aggrandit\n");
         resize(&t, &length);
      }
      t[i] = n;
      i++;
   }
   while (n > 0);

   for (j = 0; j < i; j++)
      printf("%.2f ", t[j]);
   printf("\n");
}
```

```
void resizeOld(PtrDouble * ptab, int * pnbElmts)
{
   int i;

   PtrDouble tab = (PtrDouble)malloc(((*pnbElmts)+INCREMENT)*sizeof(double));
   assert(tab);

    for (i = 0; i < *pnbElmts; i++)
       tab[i] = (*ptab)[i];

   free(*ptab);
   *ptab = tab;
   *pnbElmts += INCREMENT;
}

void resize(PtrDouble * ptab, int * pnbElmts)
{
   *ptab = (PtrDouble)realloc((void *) *ptab, ((*pnbElmts)+INCREMENT)*sizeof(double));
   assert(*ptab);

   *pnbElmts += INCREMENT;
}
```

CHAPITRE 12 : POINTEURS DE FONCTIONS

1 DEFINITION

Le nom d'une fonction peut être considéré comme une adresse sur le début des instructions en mémoire de la fonction (similitude avec le nom d'un tableau). Il est donc possible de définir des pointeurs vers des fonctions.

f <==>&f (cf pour les tableaux : tab <==> &tab[0])

Déclarations : **int (*pf) (int) ;**

extern int f (int);

pf est un pointeur vers une fonction ***avec un paramètre de type int et un retour de type int.***

f est une fonction ***avec un paramètre de type int et un retour de type int.***

	Fonction	Pointeur de fonction adresse de fonction **notation conventionnelle**	Pointeur de fonction adresse de fonction **notation "option confort"**
Affectation du pointeur de fonction		**pf = &f;**	**pf = f;**
Appel de la fonction	int i; i = **f(5)** ;	int i; i = **(*pf)(5)** ;	int i; i = **pf(5)** ;

L'utilisation la plus fréquente et la plus pertinente des pointeurs de fonctions est en **paramètre de fonction.**

On parle de fonction paramétrée par des fonctions, ou encore de traitement paramétré par des traitements.

2 EXEMPLES

Exemple 1 : Appel d'une fonction via un pointeur sur cette fonction

```
#include <stdio.h>
int f(int i, int j){ return i + j;}

int main()
{
   int l, m;
   int (*pf)(int, int);  /* Déclare un pointeur de fonction. */

    pf = &f;   /* Initialise pf avec l'adresse de la fonction f. */
   printf("Entrez le premier entier : "); scanf("%d",&l);
   printf("\nEntrez le deuxième entier : "); scanf("%d",&m);

   printf("\nLeur somme est de : %u\n", (*pf)(l,m));
   return 0;
}
```

Exemple 2 : Tableau de pointeurs de fonctions

```
#include <stdio.h>

/* le type fptr est celui d'un pointeur de fonction à 2 paramètres de type int et de valeur
de retour int */
typedef int (*fptr)(int, int);

/* Définitions de plusieurs fonctions de même signature: */
int somme(int i, int j){ return i + j; }
int multiplication(int i, int j) { return i * j; }
int quotient(int i, int j) { return i / j; }
int modulo(int i, int j) { return i % j; }
```

```
int main()
{
   int i ,j, n;
   /* fptr ftab[4] à la place de int (*ftab[4])(int, int) */
   fptr ftab[4] = {somme, multiplication, quotient, modulo};

   printf("Entrez le premier entier : ");
   scanf(" %d",&i);
   printf("\nEntrez le deuxième entier : ");
   scanf(" %d",&j);
   printf("\nEntrez la fonction 1 somme 2 multiplication 3 quotient 4 modulo : ");
   scanf(" %d",&n);
   if ((n > 0) && (n <= 4))
      printf("\n Résultat : %d.\n", (*(ftab[n - 1]))(i,j) );
   else
      printf("\n Mauvais numéro de fonction.\n");
   return 0;
}
```

Exemple 3 : pointeur de fonction en paramètre de fonction

```
#include <stdio.h>
/* le type fptr est celui d'un pointeur de fonction à 2 paramètres de type int et de valeur de retour int */
typedef int (*fptr)(int, int);

/* Déclarations de plusieurs fonctions de même signature: */
int somme(int i, int j);
int multiplication(int i, int j);
int quotient(int i, int j);
int modulo(int i, int j);

/* déclaration d'une fonction de calcul selon un pointeur de fonction passé en paramètre avec les pararamètres correspondants
exemple : int x = calcul(somme, 9, 6); x vaut 15
*/
int calcul (fptr f, int a, int b);

/* Définitions plusieurs fonctions de même signature: */
int somme(int i, int j){ return i + j; }
int multiplication(int i, int j) { return i * j; }
int quotient(int i, int j) { return i / j; }
int modulo(int i, int j) { return i % j; }

/* définition d'une fonction de calcul selon un pointeur de fonction passé en paramètre avec les pararamètres correspondants*/
int calcul (fptr f, int a, int b)
{
   return (*f)(a, b);
}
```

```
int main()
{
   int i,j,n;

   /* fptr ftab[4] à la place de int (*ftab[4])(int, int) */
   fptr ftab[4] = {somme, multiplication, quotient, modulo};

   printf("Entrez le premier entier : ");
   scanf(" %d",&i);
   printf("\nEntrez le deuxième entier : ");
   scanf(" %d",&j);
   printf("\nEntrez la fonction 1 somme 2 multiplication 3 quotient 4 modulo : ");
   scanf(" %d",&n);
   if ((n > 0) && (n <= 4))
      printf("\n Résultat : %d.\n", calcul(ftab[n - 1], i, j));
   else
      printf("\n Mauvais numéro de fonction.\n");
   return 0;
}
```

Détermination d'un type complexe :

Que signifie char (*(*x())[])() ?

x	est un nom
x()	est une fonction retournant
***x()**	est une fonction retournant un pointeur sur
(*x())	est une fonction retournant un pointeur sur
(*x())[]	est une fonction retournant un pointeur sur un tableau de
***(*x())[]**	est une fonction retournant un pointeur sur un tableau de pointeurs sur
(*(*x())[])	est une fonction retournant un pointeur sur un tableau de pointeurs sur
(*(*x())[])()	est une fonction retournant un pointeur sur un tableau de pointeurs sur fonction retournant
char (*(*x())[])()	est une fonction retournant un pointeur sur un tableau de pointeurs sur fonction retournant un char

CHAPITRE 13 : LES FICHIERS

Les données stockées en mémoire sont perdues dès la sortie du programme. Les fichiers sur support magnétique (bande, disquette, disque) sont par contre permanents, mais au prix d'un temps d'accès aux données très supérieur.

On distingue les **fichiers séquentiels** (on accède au contenu dans l'ordre du stockage) ou **à accès direct** (on peut directement accéder à n'importe quel endroit du fichier). Les fichiers que nous allons considérer sont des fichiers séquentiels. Les fichiers sont soit **binaires** (un float sera stocké comme il est codé en mémoire , d'où gain de place mais incompatibilité entre logiciels), soit **formatés ASCII (fichiers textes,** un float binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre).

Les fichiers étant dépendants du matériel, ils ne sont pas prévus dans la syntaxe du C mais par l'intermédiaire de fonctions spécifiques.

1 FICHIER TEXTE

1.1 CRÉATION ET OUVERTURE D'UN FICHIER

Un fichier peut être représenté de 2 manières:

1) **Son nom externe**

C'est le nom que lui a donné son propriétaire au moment de sa création sous la forme d'une chaîne de caractères.

2) **Son nom interne**

C'est le nom que lui a donné le système au moment de l'ouverture de ce fichier. Ce nom interne est une structure de fichier de type **FILE** (obligatoirement en majuscules) et qui contient les informations suivantes: *mode d'ouverture* du fichier, numéro du descripteur de fichier, informations concernant le tampon associé au fichier pour les opérations de lecture/écriture, pointeur courant dans le fichier.

A chaque fois que l'on prévoit d'appeler des fonctions d'E/S de la bibliothèque standard, il convient d'inclure le fichier **stdio.h** qui contient les définitions et les prototypes des fonctions d'E/S.

Il existe 3 fichiers *standard* associés à l'écran du terminal :

stdin: fichier d'entrée (nom interne = 0)

stdout: fichier de sortie (nom interne = 1)

stderr: fichier de sortie erreur (nom interne = 2)

Pour ouvrir un fichier il faut déclarer un pointeur de type **FILE *** et faire appel à la fonction **fopen** dont le prototype est le suivant :

FILE * fopen(const char * path, const char * mode);

La fonction **fopen** retourne un pointeur sur une structure de type FILE ou bien un pointeur NULL si l'opération a échoué.

Il existe plusieurs modes d'ouverture d'un fichier :

Mode	Fonction
"r"	Ouvre un fichier existant pour la lecture
"w"	Crée et ouvre un fichier pour l'écriture. Tout fichier existant est remplacé. Un nouveau fichier est créé, s'il n'en existe pas.
"a"	Ouvre un fichier en ajout. L'ajout se fait à la fin d'un fichier existant. Un nouveau fichier est créé, s'il n'en existe pas.
"r+"	Ouvre un fichier existant pour la lecture et l'écriture
"w+"	Crée et ouvre un fichier existant pour la lecture et l'écriture. Tout fichier existant est remplacé.
"a+"	Ouvre un fichier pour la lecture et l'ajout de données à la fin d'un fichier existant. Un fichier est créé s'il n'en existe pas

Exemples :

```
int main() {
        FILE *fp;
        fp = fopen ("nom-de-fichier", "r");
        if (fp == NULL) {
                printf("Echec ouverture du fichier \n");
        }
        else {
                printf("Succès ouverture du fichier \n");
                fclose(fp);
        }
}

int main() {
        FILE *fp;
        if ((fp = fopen("test.c", "r")) != NULL) {
                printf("le fichier test.c existe\n");
                fclose(fp);
        }
        else {
                printf("le fichier test.c n'existe pas\n");
                if ((fp = fopen("test.c", "w")) != NULL) {
                        printf("le fichier test.c a ete cree\n");
                        fclose(fp);
                }
                else
                        printf("erreur création fichier text.c \n");
        }
        return 0;
}
```

Spécifier un chemin d'accès

La constante ***FILENAME_MAX*** définit la taille maximum du plus long mon possible de fichier (chemin compris) supporté par le système.

Exemple chemin relatif :

```
int main()
{
        FILE * fp;
        if ((fp = fopen("../../test.txt","w")) != NULL) {
                printf("le fichier est créé 2 repertoires au dessus du repertoire \
                                ou se trouve le programme\n");
                fclose(fp);
        }
        else
                printf("erreur création fichier\n");
        return 0;
}
```

1.2 LECTURE/ECRITURE

De nombreuses fonctions permettent l'écriture ou la lecture.

Lecture:

*int **fgetc** (FILE * stream);*

*int **fgets** (char * string, int n, FILE * stream);*

*int **fscanf** (FILE * stream, const char * format[,argument,...];*

Ecriture:

*int **fputc** (int c, FILE * stream);*

*int **fputs** (const char * string, FILE * stream);*

*int **fprintf** (FILE * stream, const char * format[,argument,...];*

1.2.1 LECTURE / ÉCRITURE DE CARACTERE

int fgetc(FILE *)

Cette fonction retourne un caractère lu sur le fichier passé en paramètre et **EOF** en cas d'erreur où de détection de la fin du fichier.
Elle est équivalente sous la forme d'une fonction véritable à la macro **getc().**

int fputc(int, FILE *)

Cette fonction écrit le caractère passé en paramètre sur le fichier passé en paramètre. Elle retourne le caractère passé en paramètre, en cas d'erreur elle retourne **EOF**.
Elle est équivalente sous la forme d'une fonction véritable à la macro **putc().**

Exemple :

Le programme ci-dessous ouvre en lecture seule le fichier **entree.txt**.
Ce fichier est sensé exister et se trouver dans le même répertoire que le programme.
Pour les besoins du test il doit également contenir du texte.
Un deuxième fichier **sortie.txt** est ouvert ou créé s'il n'existe pas encore, en écriture seule.
Le contenu d'**entrée.txt** est récupéré caractère par caractère et copié sur **sortie.txt** et également sur **stdout** afin de vérifier visuellement le fonctionnement :

```
#include <stdio.h>
#include <stdlib.h>
#define ERREUR(msg)     {\
                        printf("%s\n",msg);\
                        exit(1);\
                        }
```

```
int main()
{
        FILE * in, * out;
        int c;

        /* ouverture fichier entree en lecture seule */
        in = fopen("entree.txt","r");
        if (in == NULL)
                ERREUR("ouverture du fichier entree.txt");

        /* ouverture sortie en écriture seule */
        out = fopen("sortie.txt","w");
        if (out == NULL)
                ERREUR("ouverture du fichier sortie.txt");

        /* lecture écriture caractère par caractère */
        while ((c = fgetc(in)) != EOF) {
                fputc(c, out);
                fputc(c, stdout); /* contrôle sur la sortie standard */
        }
        fclose(in);
        fclose(out);
        return 0;
}
```

1.2.2 LECTURE / ÉCRITURE DE CHAÎNES

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

Cette fonction lit une chaîne entrée sur le fichier passé en paramètre (dans le cas de l'entrée standard ce fichier sera **stdin**) jusqu'à un **'\n'** ou jusqu'à ce que **n-1** caractères aient été lus, le **'\n'** final compris. Tous les caractères lus sont placés à partir de l'adresse **s**.
Un **'\0'** est ajouté comme dernier caractère après le **'\n'** dans le cas d'une lecture sur l'entrée standard stdin. La fonction retourne l'adresse **s** ou NULL en cas d'erreur.

int fputs(char *, FILE *)

Cette fonction permet d'écrire (copier) la chaîne passée en paramètre dans le fichier passé en paramètre. Elle retourne EOF si une erreur se produit.

Exemple :

Idem programme précédent mais chaîne de caractères par chaîne de caractères

```
#define ERREUR(msg)        {\
                                printf("%s\n",msg);\
                                exit(1);\
                           }

int main()
{
        FILE * in, * out;
        char buf[1000];

        /* ouverture fichier entree en lecture seule */
        in = fopen("entree.txt","r");
        if (in == NULL)
                ERREUR("ouverture du fichier entree.txt");
```

```
        /* ouverture sortie en écriture seule */
        if ((out = fopen("sortie.txt","w")) == NULL)
                ERREUR("ouverture du fichier sortie.txt");

        /* lecture écriture avec chaînes de caractères */
        while (fgets(buf, 1000, in) != NULL) {
                fputs(buf, out);
                fputs(buf, stdout);
        }
        fclose(in) ;
        fclose(out) ;
        return 0;
}
```

1.2.3 LECTURE / ÉCRITURE FORMATEE

int fscanf (FILE * stream, const char * format, …)

Cette fonction analyse une suite de caractères lus sur le fichier .

Elle la découpe en "tokens" et convertit chaque token en une valeur.

Le découpage et la conversion sont effectués selon la spécification de format.

Pour chaque spécification de conversion que contient celle-ci, il doit exister un paramètre de type pointeur dans la liste des paramètres variables (notée par ...).

Chaque valeur résultant d'une conversion est copiée à l'adresse du paramètre correspondant.

La fonction retourne le nombre de valeurs correctes trouvées compte tenu de l'adéquation entre les formats et les références de paramètre prévues.

int fprintf (FILE * stream, const char* format, …)

Cette fonction formate la liste des paramètres variables selon la spécification de format.

La chaîne de caractères résultante est écrite dans le fichier.

Exemples :

Même principe que dans les programmes précédents mais le contenu du fichier ***entree.txt*** contient maintenant trois nombres ainsi qu'une suite de mots pour tester les formats %d et %s.

Voici ce contenu : ***10, 15, 4, toto fait des courses***

où 4 indique le nombre de mots qui suivent

Tout ce qui est récupéré dans le fichier en ***entrée*** est recopié dans le fichier en ***sortie*** et également dans le fichier standard ***stdout*** d'affichage console.

```
#define ERREUR(msg)        {\
                                printf("%s\n",msg);\
                                exit(1);\
                           }

int main()
{
        FILE * in, * out;
        int a ,b, nb, i;
        char buf[1000] ;

        /* ouverture fichier entree en lecture seule */
        in = fopen("entree.txt","r");
        if (in == NULL)
                ERREUR("ouverture du fichier entree.txt");

        /* ouverture sortie en écriture seule */
        if ((out = fopen("sortie.txt","w")) == NULL)
                ERREUR("ouverture du fichier sortie.txt");

        /* lecture des valeurs des nombres avec format %d */
        fscanf(in, "%d, %d, %d, ", &a, &b, &nb);
```

```
/* écriture des valeurs des nombres en chaînes de caractères avec format %d */
        fprintf(out, "%d, %d, %d, ", a, b, nb);
        /* équivalent printf, contrôle dans fenêtre console */
        fprintf(stdout, "%d, %d, %d,\n", a, b, nb);

        /* récupération des nb mots */
        for (i = 0; i < nb; i++) {
                fscanf(in,"%s",buf);
                fprintf(out,"%s",buf);
                fprintf(stdout, "%s ", buf); /* contrôle sur la sortie standard */
        }
        return 0;
}
```

1.3 FERMETURE D'UN FICHIER

Tout fichier ouvert doit être fermé. La fonction fclose( ) permet de fermer le fichier repéré par son pointeur.

int fclose (FILE *stream);

1.4 DEPLACEMENTS DANS UN FICHIER

Pour se repositionner n'importe où dans le fichier :

int fseek(FILE * stream, long offset, int origin);

Cette fonction permet de modifier la position courante dans le fichier à partir du positionnement ***origin*** de la façon suivante :

SEEK_SET à partir du début du fichier

SEEK_CUR à partir de la position courante

SEEK_END à partir de la fin.

Le second paramètre ***offset*** indique un déplacement en octets.

Exemple :

fseek (monFichier, 16, SEEK_SET)

déplace la position courante de 16 octets (16 caractères) à partir du début du fichier.

En cas d'erreur fseek renvoie une valeur négative et 0 si pas d'erreur.

Pour connaître la position courante :

long ftell (FILE * stream);

Cette fonction retourne la position courante.

Elle échoue si la taille à retourner est supérieure à ***LONG_MAX*** (Dans ce cas il faut utiliser la fonction ***fgetpos***).

Pour se repositionner en début de fichier :

int rewind(FILE * stream);

Cette fonction réinitialise la position courante associée au fichier au début du fichier.

Remarque : ***rewind(fp)*** est équivalent à ***fseek(fp, 0L, SEEK_SET)***.

Exemple:

```
#include <stdio.h>

main( )
{
        FILE * fp;
        char c;
        char t[100], s[100], tab1[100], tab2[100];

         /* ouvrir le fichier en lecture */
        fp = fopen ("fichier.txt","r");

        if (fp != NULL)
        {
                /* lire les 51 premiers caractères dans le tableau s */
                fgets(s, 52, fp);

                 /* lire le caractère suivant dans c */
                c = fgetc(fp);

                /* lire le mot suivant dans le tableau t */
                fscanf(fp, "%s", t);

                /* afficher le contenu des variables s, c, t à l'écran */
                printf("s=%s c=%c t=%s", s, c, t);
                /* se positionner en début de fichier */
                rewind(fp);

                /* lire dans tab1 les 99 premiers caractères */
                fgets (tab1, 100, fp);
                /* afficher à l'écran le contenu de tab1 */
                printf("\n tab1=%s", tab1);

                /* se positionner à un déplacement de -40 octets par rapport au pointeur courant */
                fseek (fp, -40L, SEEK_CUR);

                /* lire les 99 caractères à partir du pointeur courant dans tab2 */
                fgets (tab2, 100, fp);

                /* afficher le contenu de tab2 à l'écran */
                printf("\n tab2=%s", tab2);

                /* fermer le fichier */
                fclose(fp);
        }
}
```

1.5 LE CARACTÈRE DE FIN DE FICHIER (EOF)

Tous les fichiers texte ou binaire possèdent un caractère qui marque la fin du fichier.

Cette marque de fin de fichier appelée génériquement **EOF** est souvent différent selon le type de machine utilisée.

La valeur EOF est définie dans le fichier d'inclusion *stdio.h*.

Pour un fichier texte, cette valeur est le plus souvent égale à -1.

remarque : EOF est aussi utilisée comme retour d'erreur, par exemple pour la fonction fclose.

Exemple :

```
/* lire le fichier caractère après caractère jusqu'à la marque de fin de fichier */
#include <stdio.h>
main() {
        int i;
        FILE * fp;
        // ouverture et lescture du fichier
        if ((fp = fopen("fichier","r")) != NULL) {
                while ((i = fgetc(fp)) != EOF)
                        printf("%c", i);
        }
}
```

Pour savoir si la fin de fichier est atteinte :

int feof (FILE * stream);

Cette fonction retourne ***vrai*** si le la position courante dans le fichier est à la fin et ***faux*** sinon.

Peut-être utilisée pour les deux modes, texte et binaire.

2 FICHIERS BINAIRES

Certaines applications nécessitent de mémoriser les données par blocs dans les fichiers. Par exemple, si dans une applications un ensemble de structures ayant la même composition est utilisé, il est intéressant de pouvoir écrire dans le fichier le contenu d'une structure entière (sans être obligé d'écrire le contenu de la structure champ par champ).

Les fonctions ***fread*** et ***fwrite*** sont destinées à ce type d'entrée/sorties et sont désignées sous le terme fonctions d'entrées/sorties binaires.

Ces fonctions requièrent quatre arguments :

- un pointeur sur le bloc de données (par exemple un pointeur de structure)
- la taille du bloc
- le nombre de blocs transférés
- le pointeur de fichier.

size_t fread(void * ptr, size_t size, size_t n, FILE * stream);

n objets lus de même taille size, stockés à l'adresse ptr.

retourne le nombre d'objets lus, soit n, sinon erreur ou fin de fichier

size_t fwrite(const void * ptr, size_t size, size_t n, FILE * stream);

n objets écrits de même taille size, pris à partir de l'adresse ptr.

Les autres fonctions de manipulation de fichiers (***fopen, fclose***) ont la même utilisation avec les fichiers binaires.

Il est nécessaire d'ajouter un **'b'** aux droits requis à l'ouverture (exemple: **"rb+"**).

Exemple :

```
#include <stdio.h>

 typedef struct
{
        char nom[30];
        char prenom[40];
        int age;
        char sexe;
  } personne;


void main( )
{
        FILE * fp;

        personne dupont = {"DUPONT","RENE",28,'M'};
        personne dupontBis;

        if ((fp = fopen("essai.dat","wb")) != NULL)
        {
                /*ecriture de dupont dans le fichier */
                fwrite(&dupont, sizeof(personne), 1, fp);
                fclose(fp);
        }

        if (fp = fopen("essai.dat","rb"))  != NULL)
        {
                /*lecture de dupontBis dans le fichier */
                fread(&dupontBis, sizeof(personne), 1, fp);
                fclose(fp);

                printf("\n\n NOM :%s PRENOM : %s", dupontBis.nom,
                                                        dupontBis.prenom);
                printf("\n AGE %d", dupontBis.age);
                printf("\n SEXE : %c\n",( dupontBis.sexe ==' M') ? 'M': 'F');
        }
}
```

CHAPITRE 14 : OBTENTION D'UN EXECUTABLE

1 DIFFERENTES PHASES AVANT L'EXECUTION

Pour arriver à l'exécution d'un programme en partant d'un ou plusieurs fichiers sources en C, les étapes suivantes sont nécessaires :

l'édition de textes	éditeur
l'expansion du ou des fichiers source	préprocesseur
la compilation	compilateur
l'assemblage	assembleur
l'édition de liens	éditeurs de liens
le chargement en mémoire	chargeur

1.1 ÉDITION DE TEXTES

Les fichiers sources en langage C sont saisis à l'aide d'un éditeur de textes (emacs, vi sous Unix). Le suffixe d'un nom d'un fichier source est ".c".

1.2 EXPANSION DU FICHIER SOURCE

Seules les lignes commençant par un dièse (#) sont traitées par le préprocesseur.
Ce sont les directives du préprocesseur (inclusion de fichiers, définition de constantes et macros, …).

exemples de directives :

```
#include <stdio.h>
#define MAX 1000
```

Le préprocesseur :

remplace les commentaires par du blanc
insère le texte d'autres fichiers
remplace les constantes et les macros dans le texte

1.3 COMPILATION

Le compilateur vérifie la syntaxe et traduit le source en langage assembleur.

1.4 ASSEMBLAGE

Les instructions du langage assembleur sont traduites en langage machine.

Cette phase produit pour chaque source une fichier objet dont le nom se termine par ".o"

Un fichier objet peut contenir des références externes d'objets définis dans d'autres sources.

1.5 ÉDITION DE LIENS

A partir du ou des fichiers objets passés en paramètres :

résoud les références externes

éventuellement intègre des fonctions de la librairie standard ou autres

génère un fichier exécutable.

1.6 CHARGEMENT

Le chargeur copie le fichier exécutable en mémoire centrale et initialise l'éxécution.

Il existe des différences dans les commandes correspondantes à ces phases selon le système d'exploitation et selon le compilateur utilisé.

Sur certains compilateurs, le préprocesseur, le compilateur et l'assembleur correspondent à une seule et même phase.

La plupart des compilateurs offre la possibilité d'obtenir un fichier source en assembleur (".s" ou ".a") à partir du fichier en C.

Exemple sur UNIX avec le compilateur gcc :

toutes les phases :

gcc - o prog prog1.c prog2.c exécutable prog

ou

g**cc prog1.c prog2.c** exécutable a.out

préprocesseur seul :

gcc -E prog.c sur l'écran

préprocesseur + compilation :

gcc -S prog.c prog.s

préprocesseur + compilation + assemblage :

gcc -c prog.c prog.o

édition de liens seule :

gcc -o prog prog1.o prog2.o exécutable prog

avec librairie mathématique :

gcc -o prog prog1.o prog2.o -lm exécutable prog

2 PREPROCESSEUR

Le préprocesseur est la première phase du compilateur C. Son rôle est de créer un fichier source intermédiaire, obtenu à partir de l'original par inclusion d'autres fichiers sources, substitution de parties de textes ...

Ses directives sont introduites par le caractère #.

2.1 SUBSTITUTION SYMBOLIQUE

2.1.1 CONSTANTES

#define IDENT expr

IDENT doit être un identificateur régulier.

Le préprocesseur substituera alors chaque occurence de IDENT par expr dans le fichier source, lorsque le contexte fait appaître IDENT comme identificateur.

```
#define BUFSIZ 512
void f ()
{
        char buf [BUFSIZ] ;                  /* substitution */
        char * s = "buf de taille BUFSIZ" ;  /* pas de subtitution dans la chaîne */
}

#define CONSTSTRING             "chaîne constante \ n"
# define then
# define forever                while (1)
```

2.1.2 MACROS

 # define macro (m1, ..., mN)expr (m1, ..., mN)

m1, ..., mN sont des identificateurs formels arbitraires.

Il est conseillé, pour éviter les effets inattendus de précédence d'opérateurs, de parenthèser m1, ..., nM dans expr.

Une macro volumineuse peut être définie sur plusieurs lignes en backslashant toutes les lignes sauf la dernière.

Une macro peut être définie à partir d'autres macros précédemment définies.

Noter l'absence de ';' en fin de définition.

Exemples :

```
# define abs(n)              ((n) > = 0  ?  (n) : - (n))
# define max(a,b)            ((a) > = (b) ? (a) : (b))
#define squareBug(x)          (x * x)
#define square(x)             ((x) * (x))
```

squareBug(3) est substitué par (3 * 3) et vaut 9

squareBug(3 + 5) est substitué par (3 + 5 * 3 + 5) et vaut 23 ==> BUG!!!

square(3) est substitué par ((3) * (3)) et vaut 9

square(3 + 5) est substitué par ((3 + 5) * (3 + 5)) et vaut 64

2.2 INCLUSION DE FICHIERS SOURCES

 # include "chemin-fichier"

 # include < chemin-fichier >

Cette dernière sert à inclure dans le fichier source actuel d'autres fichiers, appelés en général "headers", et dont le suffixe traditionnel est ".h".

Ces fichiers peuvent contenir :

des définitions de constantes symboliques (#define)

des définitions de type (FILE, …)

des déclarations d'objets globaux ou externes.

Dans la première syntaxe, le fichier est recherché dans la liste de priorité des répertoires définie par une variable d'environnement (le répertoire courant en fait partie).

Dans la deuxième syntaxe, le fichier est exclusivement recherché dans le répertoire /usr/include. Cette deuxième syntaxe est réservée aux fichiers headers livrés avec le compilateur.

2.3 COMPILATION CONDITIONNELLE

2.3.1 SUR LA VALEUR DE CONSTANTES

 # if expression_constantel

séquence compilée si expression_constantel est vraie.

 # else

 # if expression_constante2

...

 # endif

#endif

Expression_constante peut être formée de constantes entières liées par les opérateurs suivants :

unaires -

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

L'expression peut contenir des parenthèses.

 # if (PROCESSEUR == 1)

...

 # else

...

 # endif

2.3.2 SUR LA DÉFINITION DE CONSTANTES

 # ifdef constante_symbolique

séquence compilée si la constante est connue du préprocesseur

(par #define ou option -D de gcc)

 # else

...

 # endif

 # ifdef DEBUG

...

 # endif

Remarque 1 : la directive contraire **#ifndef** permet d'éviter les conflits de redéfinition :

 # ifndef NULL

*# define NULL (char *) 0*

 # endif

Remarque 2 : la directive **# undef** constante permet d'annuler la définition précédente d'une constante (pour lui attribuer une nouvelle valeur par exemple).

Chapitre 15 : LA LIBRAIRIE STANDARD

1 TRAITEMENTS DES CHAÎNES DE CARACTÈRES

Le type chaîne de caractères n'existe pas en C.

Seules les constantes chaînes de caractères sont reconnues par le compilateur.

La librairie standard offre des **fonctions qui manipulent les chaînes**.

Les déclarations prototypes figurent dans le fichier **string.h**

Le nom d'une chaîne correspond à son adresse de début, elle est représentée en mémoire par une suite d'octets contigus terminée par une **sentinelle valant '\0'**.

Les algorithmes sur les chaînes utilisent ce caractère de fin de chaîne.

1.1 LONGUEUR D'UNE CHAÎNE : STRLEN

int strlen (const char * chaîne);

strlen retourne le nombre de caractères d'une chaîne sans compter le '\0'.

exemple : *int lg = strlen ("ceci est une chaîne") ;*

lg vaut 19, sizeof("ceci est une chaîne") vaut 20.

1.2 COPIE D'UNE CHAÎNE DANS UNE AUTRE : STRCPY

char * strcpy (char * destination, const char * source);

strcpy copie la chaîne source à l'adresse destination.

La chaîne destination doit être suffisament grande pour accueillir la chaîne source.

L'adresse de la chaîne destination est retournée.

exemple : *char ch1 [30] ;*

char ch2 [30] ;

strcpy (ch1, "je tiens dans 28 caractères") ;

ch1 vaut "je tiens dans 28 caractères"

strcpy(ch2, strcpy (ch1, "je tiens dans 28 caractères"));

ch2 vaut aussi "je tiens dans 28 caractères" car l'appel strcpy (ch1, "je tiens dans 28 caractères") retourne un pointeur sur ch1.

1.3 CONCATENATION DE CHAÎNES : STRCAT

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

strcat concatène à la fin de la chaîne destination la chaîne source

La chaîne destination doit être suffisamment grande pour accueillir la chaîne source en supplément.

L'adresse de la chaîne destination est retournée.

exemple :

```
char ch1 [32] ;
strcpy (ch1, "ceci est");
ch1 vaut "ceci est"
strcat (ch1, "une chaîne");
ch1 vaut "ceci est une chaîne"
```

1.4 COMPARAISON DE CHAÎNES : STRCMP

int strcmp (const char * ch1, const char * ch2);

strcmp compare 2 chaînes lexicographiquement et retourne :

0	si les chaînes sont identiques
>0	si ch1 est plus grand lexicographiquement que ch2
< 0	si ch2 est plus grand lexicographiquement que ch1.

exemple :

```
if (! strcmp (ch1, ch2))
        printf ("chaînes identiques\n") ;
```

2 LIMITES ET FONCTIONS DE CONVERSION

2.1 LIMITES POUR LES ENTIERS

fichier <limits.h>

SHRT_MIN, SHRT_MAX

LONG_MIN, LONG, MAX

..

2.2 LIMITES POUR LES REELS

fichier <float.h>

FLT_MIN, FLT_MAX

DBL_MIN, DBL_MAX

LDBL_MIN, LDBL_MAX

FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON

the smallest X of type xxxxx such that 1.0 + X != 1.0.

2.3 CONVERSION DE CHAÎNES EN NOMBRES : ATOF, ATOI, ATOL

La librairie standard fournit des **fonctions de conversion**.

Les déclarations prototypes des fonctions figurent dans le fichier **stdlib.h** :

```
double atof (const char * s);
int    atoi (const char * s);
long   atol (const char * s);
```

exemple :

ligne de commande : *$ a.out 5*

donc argv[1] vaut "5"

```
int main ( int argc, char * argv [] )
{
        int x;
        ...
        x = atoi(argv[1]); /* soit atoi("5"), x vaut 5 */
        ...
        return 0;
}
```

2.4 CONVERSIONS EN MINUSCULE OU EN MAJUSCULE : TOLOWER, TOUPPER

Les déclarations prototypes des fonctions figurent dans le fichier **ctype.h**.

int tolower (int c);

int toupper (int c);

tolower convertit une lettre majuscule en une lettre minuscule. Si la lettre passée en paramètre est une minuscule, la lettre retournée est inchangée (et inversement pour **toupper**).

exemple :

avec indices du tableau	avec des pointeurs
char chaine[8]= "abcdef"; *int i.* *for (i = 0; i < chaine[i] != '\0'; i++)* ***chaine[i]= toupper(chaine[i]);*** *printf("%s\n", chaine);*	*char * ch1, * ch2 ;* ***ch2 = ch1 = chaine;*** *while(*ch1)* ****ch2++ = tolower(*ch1++);*** *printf("%s\n", chaine);*
chaine vaut ABCDEF	chaine vaut abcdef

3 FONCTIONS D'ENTREE - SORTIE

3.1 GÉNÉRALITÉS

Les opérations d'entrée/sortie sont réalisées par des fonctions utilisant un mécanisme de bufferisation et un pointeur de fichier de type **FILE ***, défini dans le fichier s**tdio.h**.

A l'entrée dans un programme, trois pointeurs de fichiers sont automatiquement attribués :

stdin pour l'entrée standard

stdout pour la sortie standard

stderr pour la sortie erreur

3.2 DÉCLARATIONS PROTOTYPES

Les déclarations prototypes des fonctions figurent dans le fichier **stdio.h.**

FILE * fopen(const char * path, const char * mode);

mode :

"r" lecture, le fichier doit exister

"w" [création] + écriture, si le fichier existe déjà alors il est écrasé

"a" écriture en fin uniquement

"r+" lecture + écriture, le fichier doit exister

"w+" [création] + lecture + écriture, si le fichier existe déjà alors il est écrasé

"a+" lecture + écriture en fin uniquement

retourne un pointeur nul si erreur

size_t fread(void *ptr, size_t size, size_t n, FILE * stream);

n objets lus de même taille size, stockés à l'adresse ptr.

retourne le nombre d'objets lus, soit n, sinon erreur ou fin de fichier

size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);

n objets écrits de même taille size, pris à partir de l'adresse ptr.

int ***fseek(FILE * stream, long offset, int whence);***

Constantes utilisées comme 3ème argument pour "fseek"

#define SEEK_CUR 1

#define SEEK_END 2

#define SEEK_SET 0

int ***fclose(FILE * stream);***

int ***feof(FILE * stream);***

0 si fin de fichier non atteinte

1 sinon

int ***ferror(FILE * stream);***

retourne la valeur de la variable globale système **errno**

0 si pas d'erreur d'E/S

! 0 sinon

long ***ftell(FILE * stream);***

donne la position courante

void ***rewind(FILE * stream);***

rembobine, soit *fseek (stream, OL, SEEK_SET)*

void ***perror(const char * s);***

affiche le message d'erreur du système

void ***clearerr(FILE * stream);***

reset les flags d'erreur et de fin de fichier

int ***fflush(FILE * stream);***

vide le buffer associé au pointeur de fichier

int ***getc(FILE * stream);***

lit et retourne un caractère du fichier pointé par stream

macro

int ***fgetc(FILE * stream);***

idem getc mais fonction

int ***ungetc(int c, FILE * stream);***

remet le caractère c dans le flux d'entrée

int ***putc(const int c, FILE * stream);***

écrit un caractère dans le fichier pointé par stream

macro

int ***fputc(const int c, FILE * stream);***

idem putc mais fonction

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

lit au maximum n-1 caractères jusqu'au 1er '\n'

résultat stocké dans s

int ***fputs(const char * s, int n, FILE * stream);***

écrit la chaîne s dans le fichier pointé par stream

int ***fscanf(FILE *stream, const char * format, ...);***

int ***fprintf(FILE *stream, const char * format, ...);***

sur l'entrée et la sortie standart :

int ***getchar(void);***

int ***putchar(const int c);***

char * gets(char * s);

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

int ***scanf(const char * format, ...);***

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

La fonction ***printf*** permet de réaliser une écriture formatée sur la sortie standard stdout.
Cette fonction renvoie le nombre de caractères affichés.
Si une erreur a lieu, elle renvoie un entier négatif.

Exemple :

```
int num;
num = printf("%s et %d\n", "test",  200);
printf("%d\n", num);
```

Indicateurs de conversion

On utilise les indicateurs de conversion suivants pour interpréter chaque valeur à afficher de manière adéquate :

Indicateur de conversion	Affichage
d, i	Entier en base dix
u	Entier non signé en base dix
x, X	Entier en hexadécimal
c	Caractère
e, E	Flottant en notation scientifique
f, g	Flottant
s	Chaîne de caractères
p	Pointeur

Caractères d'attribut

Il est possible de faire suivre le % d'un indicateur de conversion de caractères d'attribut pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
0N	Affichage du nombre sur N chiffres (ajout de 0)
N	Affichage du nombre sur N chiffres (ajout d'espaces)
+	Force l'affichage du signe d'un nombre
-	Justifie à gauche un nombre (à droite par défaut)

e**xemple**

printf("%+5d\n", 23); *affiche +23 précédé de 2 espaces*

printf("%+-5d %d\n", 23, 5); *affiche +23 5 avec 3 espaces*

CHAPITRE 16 : PROGRAMMATION MODULAIRE

Rappel: organisation (provisoire, plus modulaire par la suite) d'un programme C:

- **Inclusion des fichiers d'entête** comprenant entre autres des déclarations prototypes de fonctions: ce sont les modes d'emploi des fonctions

  exemple:

  #include <stdio.h> /* pour utiliser entre autres les fonctions d'entrées-sorties */

  #include <math.h> /* pour utiliser les fonctions mathématiques et M_PI */
- **les déclarations prototypes des fonctions** que l'on doit définir
- **la définition de la fonction principale main** (point d'entrée du programme) comprenant des appels de fonctions dont celles déclarées au préalable
- **les définitions des fonctions** déclarées au préalable

Programmation non modulaire en un seul fichier

```
/* Inclusion des fichiers d'entête */
#include <stdio.h>
#include <math.h>

/* les déclarations prototypes de fonctions */
/* Conversion kilomètres-miles*/
extern double km_vers_mile(double km);
/* Conversion Fahrenheit – C */
extern double F_vers_C(double F);
/* Volume d'une sphère */
extern double volume_sphere(double rayon);
/* Notes */
extern char note (int bonnes_reponses);
```

Programmation modulaire plusieurs fichiers

```
/* Fichier d'entête avec les déclarations prototypes de fonctions: fonctions.h */

#ifndef __FONCTION_H__
#define __FONCTION_H__

/* Inclusion des fichiers d'entête */
#include <stdio.h>
#include <math.h>

/* les déclarations prototypes de fonctions */
/* Conversion kilomètres-miles*/
extern double km_vers_mile(double km);
/* Conversion Fahrenheit – C */
extern double F_vers_C(double F);
/* Volume d'une sphère */
extern double volume_sphere(double rayon);
/* Notes */
extern char note (int bonnes_reponses);

#endif
```

/* utilisation avec les appels des fonctions */

```
/* pour afficher le menu */
static void afficheMenu(void);

static void menu(void);

void main(void)
{
   menu();
}

static void afficheMenu(void)
{
   printf(" 1 Conversion kilomètres-miles \n");
   printf(" 2 Conversion Fahrenheit – C  \n");
   printf(" 3 Volume d'une sphère \n");
   printf(" 4 Notes  \n");
   printf("taper le numero choisi: ");
}

static void menu(void)
{
   int choix;

   do
   {
      afficheMenu();
      choix = getchar();
      switch( choix)
      {
      case '1':
      {
         double km;
         printf("km = ");
         scanf("%lf",&km);
         printf("%.2lf kms valent %.2lf miles \n\n",
km, km_vers_mile(km));
         break;
      }
      case '2':
      {
         double F;
         printf("F = ");
         scanf("%lf",&F);
```

/* Fichier d'utilsation avec les appels des fonctions: exercice2.c */

```
#include "fonctions.h"

/
* pour afficher le menu */
static void afficheMenu(void);

static void menu(void);

void main(void)
{
   menu();
}

static void afficheMenu(void)
{
   printf(" 1 Conversion kilomètres-miles \n");
   printf(" 2 Conversion Fahrenheit – C  \n");
   printf(" 3 Volume d'une sphère \n");
   printf(" 4 Notes  \n");
   printf("taper le numero choisi: ");
}

static void menu(void)
{
   int choix;

   do
   {
      afficheMenu();
      choix = getchar();
      switch( choix)
      {
      case '1':
      {
         double km;
         printf("km = ");
         scanf("%lf",&km);
         printf("%.2lf kms valent %.2lf miles \n\n",
km, km_vers_mile(km));
         break;
      }
      case '2':
      {
         double F;
         printf("F = ");
         scanf("%lf",&F);
```

```
            printf("%.2lf F valent %.2lf C \n\n", F,
F_vers_C(F));
            break;
        }
        case '3':
        {
            double rayon;
            printf("rayon = ");
            scanf("%lf",&rayon);
            printf("une sphère de rayon %.2lf m a un
volume de %.2lf m3 \n\n", rayon,
volume_sphere(rayon));
            break;
        }
        case '4':
        {
            int n;
            printf("nombre de bonnes réponses = ");
            scanf("%d",&n);
            printf("%d bonnes réponses valent %c \n\n",
n, note(n));
            break;
        }
        default :
            printf("choix impossible \n");
            break;
        }
    }
    while ((choix = getchar()) != EOF);
}
```

/* les définitions des fonctions */

```
/* Conversion kilomètres-miles */
double km_vers_mile ( double km)
{
   return (km /1.609);
}

/* Conversion Fahrenheit – C */
double F_vers_C ( double f)
```

```
            printf("%.2lf F valent %.2lf C \n\n", F,
F_vers_C(F));
            break;
        }
        case '3':
        {
            double rayon;
            printf("rayon = ");
            scanf("%lf",&rayon);
            printf("une sphère de rayon %.2lf m a un
volume de %.2lf m3 \n\n", rayon,
volume_sphere(rayon));
            break;
        }
        case '4':
        {
            int n;
            printf("nombre de bonnes réponses = ");
            scanf("%d",&n);
            printf("%d bonnes réponses valent %c
\n\n", n, note(n));
            break;
        }
        default :
            printf("choix impossible \n");
            break;
        }
    }
    while ((choix = getchar()) != EOF);
}
```

/* Fichier de défintions des fonctions: fonctions.c */

```
#include "fonctions.h"

/* Conversion kilomètres-miles */
double km_vers_mile ( double km)
{
   return (km /1.609);
}

/* Conversion Fahrenheit – C */
double F_vers_C ( double f)
{
```

```
{
   return (5.0/9.0*(f -32.0));
}

/* Volume d'une sphère */
double volume_sphere ( double rayon )
{
   return (4.0/3.0 * M_PI * rayon * rayon * rayon );
}

/* Notes */
char note (int bonnes_reponses )
{
   if ( bonnes_reponses <0 || bonnes_reponses >50)
return '#';
   else if ( bonnes_reponses <=10)  return 'E';
   else if ( bonnes_reponses <=20)  return 'D';
   else if ( bonnes_reponses <=30)  return 'C';
   else if ( bonnes_reponses <=40)  return 'B';
   else if ( bonnes_reponses <=50)  return 'A';

   return '\0';
}
```

Compilation et génération d'un exécutable dont le nom par défaut est a.out

gcc exercice2.c

lancement de l'exécutable

./aout

Compilation et génération d'un exécutable renommé

gcc exercice2.c -o exercice2

lancement de l'exécutable

./exercice2

Si la librairie mathématique est nécessaire (inclure math.h et faire l'édition de lien avec l'option -lm)

```
   return (5.0/9.0*(f -32.0));
}

/* Volume d'une sphère */
double volume_sphere ( double rayon )
{
   return (4.0/3.0 * M_PI * rayon * rayon *
rayon );
}

/* Notes */
char note (int bonnes_reponses )
{
   if ( bonnes_reponses <0 || bonnes_reponses >50)
return '#';
   else if ( bonnes_reponses <=10)  return 'E';
   else if ( bonnes_reponses <=20)  return 'D';
   else if ( bonnes_reponses <=30)  return 'C';
   else if ( bonnes_reponses <=40)  return 'B';
   else if ( bonnes_reponses <=50)  return 'A';

   return '\0';
}
```

Compilation séparée:

gcc -c fonctions.c

gcc -c exercice2.c

2 fichiers binaires sont générés:

fonctions.o et exercice2.o

Edition de lien et génération d'un exécutable renommé

gcc -c fonctions.o exercice2.o -o exercice2

lancement de l'exécutable

./exercice2

Si la **librairie mathématique** est nécessaire (inclure math.h et faire l'édition de lien avec l'option -lm)

gcc exercice2.c -o exercice2 -lm	**gcc -c fonctions.o exercice2.o -o exercice2 -lm**
Remarque: pas d'extension particulière pour les exécutables sous Unix (notion de magic number)	

Annexe 1 : CODAGE BINAIRE

Les seules valeurs que peut traiter l'ordinateur (et d'ailleurs tout système numérique) est le **0** et le **1** (en fait, c'est nous qui représentons par 0 ou 1 le fait que le système numérique ait vu **du courant ou non**).

1 Représentation des types d'informations

Un problème important est que différents types d'informations doivent être codés :

- instructions machine (lignes de programmes)
- valeurs numériques
- caractères
- adresses
- ...

et que **rien ne permet de distinguer dans un groupe de 0 et de 1 le type d'information qu'il est censé représenter.**

Il est donc capital d'**associer à chaque mémoire que l'on utilise le type de donnée** que l'on y mettra. C'est ce que permet la **déclaration des variables dans les langages évolués** (explicite en C, C++ et en Java, implicite en Fortran).

Néanmoins l'**erreur reste possible** (en C, lors d'une erreur sur des ***pointeurs***, par l'utilisation des ***unions***, ou simplement par ***erreur de format dans un printf***).

2 Représentation des nombres négatifs

Une autre erreur est due à la **représentation des nombres négatifs**.

En effet, le signe ne peut être, lui aussi, représenté que par 0 ou 1.

Le codage choisi (pour les entiers 16 bits) fait que l'ajout de 1 à 32767 (le plus grand entier signé) donne -32768 (cela devait de toute façon donner un nombre car soit il y a du courant, soit il n'y en a pas, il n'est pas prévu de combinaison de 0 et 1 représentant une erreur).

La plupart des compilateurs ne signalent pas d'erreur en cas de dépassement de capacité de nombres entiers.

3 Représentation des nombres réels

Autre problème, la **codification des réels**.

Représenter la présence ou non d'une virgule par un 0 ou un 1 est évidement impossible (comment la reconnaître ?).

La solution envisagée est celle de la **"virgule flottante"** (d'où le nom de flottants ou float), représentée par **un mot pour la mantisse** (qui est un réel sans partie entière), **un autre mot (souvent de taille différente) pour l'exposant** (entier signé).

Ceci implique deux limitations :

- le nombre de chiffres significatifs (dû à la taille de la mantisse)
- le plus grand réel codifiable (dû à la taille de l'exposant, par exemple $2^{127} = 1{,}7.10^{38}$).

On cherchera donc à ne combiner que des **réels du même ordre de grandeur**.

Par exemple en mécanique, ajouter à une dimension d'un mètre une dilatation thermique d'un micron (10^{-6} m) n'a de sens que si les flottants possèdent plus de 6 chiffres significatifs, donc par exemple un algorithme cumulatif ne donnera pas de résultat si le pas en température est trop faible..

En fait on ne peut représenter que les nombres pouvant s'écrire sous forme d'une partie fractionnaire comportant un nombre fini de chiffres (en binaire).

La représentation en binaire de 1/10 donne une suite infinie, qui est donc toujours tronquée ($0,1_d = 0,0001100100100100100...{}_b$). donc sur tout ordinateur calculant en binaire, (1/10)*10 donne 0,999999999...

Cette erreur devient gênante dans le cas où le résultat du calcul précédent est utilisé dans le calcul suivant, pour de grandes suites de calculs.

exemple (flottant):

#include <stdio.h> *void main()* *{* ***float x = 1000;*** *int i;* *for (i = 0; i < 10000; i = i + 1)* *x = x + 0.1;* *printf("On obtient %12.4f au lieu de 2000.0000\n",x);* *}*	*#include <stdio.h>* *void main()* *{* ***double x = 1000;*** *int i;* *for (i = 0; i < 10000; i = i + 1)* *x = x + 0.1;* *printf("On obtient %12.4f \n",x);* *}*
1999.7559	**2000.0000**

Ce problème est moins flagrant en C que dans les autres langages, le C effectuant toujours les calculs sur **des réels en double précision**.

Il en résulte néanmoins que, par exemple, **il ne faut pas tester dans un programme si le résultat d'un calcul flottant est nul mais si sa valeur absolue est inférieure à un petit nombre.**

On cherchera aussi, autant que possible, à choisir des algorithmes utilisant des entiers plutôt que des réels, d'autant plus que les calculs sur des réels sont plus lents que sur des entiers.

ANNEXE 2 : OPERATEURS BIT A BIT

EXEMPLE 1 : LES COULEURS

Les couleurs sont la plupart du temps codés sur **24bits**, avec un octet pour le rouge, un autre pour le vert et le dernier pour le bleu, soit **8 bits pour chaque couleur**, c'est le format RGB (pour Red Green Blue).

Une couleur est donc composée de rouge, de vert et de bleu avec chacune de ces couleurs primaires allant de **0 à 255**.

En hexadécimal on a donc toutes les valeurs possibles du **noir 000000** au **blanc FFFFFF** soit environ **16 millions de couleurs** (exactement 16 777 216 couleurs possibles)!

En supposant que le rouge est codé sur le premier octet a droite, le vert sur l'octet du milieu et le bleu l'octet a gauche, on a :

Couleur : [Bleu][Vert][Rouge]

Donc pour créer une couleur avec du rouge, du vert et du bleu, il faut remplir le premier octet a droite avec le rouge, le second avec le vert et le dernier avec le bleu.
Soit : **couleur = (bleu<< 2 octets) + (vert << 1 octet) + (rouge << 0 octet)**
ce qui donne : **couleur = bleu<<16 | vert<<8 | rouge**
Faire << 8 revient a multiplier par 256 mais les opérateurs binaires sont plus rapides donc il vaut mieux les utiliser, de plus c'est plus propre et plus compréhensible.

Pour récupérer la tonalité de bleu d'une couleur , il suffit de récupérer le premier octet de celle-ci.
Par exemple pour la couleur **94F388** la tonalité de bleu est **94**.
Il suffit de décaler les bits a droite jusqu'à qu'il ne reste que l'octet qui nous intéresse soit le premier en partant de la gauche.
On a donc : **bleu = couleur >> 16**

Pour récupérer le vert on va procéder de la même manière : **vert = couleur >> 8**

Mais ensuite comment fait-on ? Il reste l'**octet codant le vert** mais aussi celui qui **code le bleu : [bleu][vert]**, on a réussi à éliminer l'octet **codant le rouge** mais il faut aussi éliminer celui qui **code le bleu**.

C'est là qu'intervient l'opérateur &. En fait il va nous permettre de récupérer notre octet :

vert = (couleur >> 8) & FF

Explications:

Soit x un bit : 0 & x = 0 et 1 & x = x,

XXYY & FF <==> XXYY & 00FF

XX & 00 = 0

FF (en hexa) = 11111111 (en base 2)

YY & FF = YY.

Pour récupérer le rouge :

Le **rouge** étant dans [XX][YY]**[ZZ]** à la troisième position, il suffit donc de faire [XX][YY][**ZZ**] **& FF** car on ne récupère anisi que le premier octet de droite.

On a donc : **rouge = couleur & FF**

Résumé :

couleur = bleu<<16 | vert<<8 | rouge

bleu = couleur >> 16

vert = (couleur >> 8) & FF

rouge = couleur & FF

EXEMPLE 2 : MODIFIER UN OCTET BIT A BIT

Si **[XX]** est un **octet**, soit **[abcdefgh]** en base 2 avec abcdefgh des **bits**.

comment récupérer un des bits ou même le modifier ?

En fait c'est très simple, et les seuls opérateurs nécessaires sont &, << et >>.

Récupérer c :

Pour récupérer c, il faut éliminer tous les autres bits :

[ab**c**defgh] & [00**1**00000] = [00**c**00000]

Maintenant pour isoler c, il faut le décaler pour qu'il soit en première position a droite :

[00**c**00000] >> **5** = **c**

Récupérer un bit en général :

Pour récupérer un bit en général dans un nombre, on utilise d'abord & puis on décale.

Il est nécessaire de connaître la position du bit dans le nombre (ici ^ signifie puissance).

bit = (nombre & 2^position) >> position

soit :

bit = (nombre & (1 << position)) >> position

EXEMPLE 3 : XOR ET LA CRYPTOGRAPHIE

^ permet de crypter des bits avec d'autres bits.

La propiété suivante est exploitée :

a ^ b = c

c ^ b = a

Ainsi :

Cryptage →	**[octet secret] ^ [octet clé] = [octet crypté]**
Décryptage ←	**[octet crypté] ^ [octet clé] = [octet secret]**

ANNEXE 3 : BIBLIOGRAPHIE

POUR DEBUTER:

BRAQUELAIRE : Méthodologie de programmation en C chez Masson

A.R. FEUER : "The C puzzle book" : Prentice Hall 1982.
Traduit en français sous le titre "Langage C : problèmes et exercices". Masson 1988.

P. DRIX : "Langage C, norme ANSI, vers une approche orientée objet". Masson 1989.

T. PLUM : "Learning to program in C". Prentice Hall 1983.
Traduit en français sous le titre "Le langage C. Introduction à la programmation" chez InterEditions.

POUR APPROFONDIR:

B.W. KERNIGHAN AND D.M. RITCHIE : "The C programming language". Second edition, ANSI-C. Prentice Hall Software Series 1988. Traduit en français. Masson 1989.

HARBISON ET G. STEELE : "Langage C. Manuel de référence". Masson 1990.

A. KOENIG : "Pièges du langage C". Addison Wesley 1994. 188F / 180 pages

STRUCTURES DE DONNEES ET ALGORITHMES

J. ESAKOV AND T. WEISS : "An advanced approach using C". Prentice Hall International Editions 1989.

A. ROBERT L. KRUSE, BRUCE P. LUNG, CLOVIS L. TONDO : "Data structures and program design in C". Prentice Hall International Editions 1991.