Langage C

Support de cours

Patrick Corde

Messagerie : corde@idris.fr

Liste des transparents

1 – Présentation du langage C

1.1 – Historique

Langage de programmation développé en 1970 par Dennie Ritchie aux Laboratoires **Bell** d'**AT&T.**

Il est l'aboutissement de deux langages :

- **BPCL** développé en 1967 par Martin Richards.
- **B** développé en 1970 chez **AT&T** par Ken Thompson.

Il fut limité à l'usage interne de **Bell** jusqu'en 1978 date à laquelle Brian Kernighan et Dennie Ritchie publièrent les spécifications définitives du langage : « **The C programming Language** ».

Au milieu des années 1980 la popularité du langage était établie.

De nombreux compilateurs ont été écrits, mais comportant quelques incompatibilités portant atteinte à l'objectif de portabilité.

Il s'est ensuivi un travail de normalisation effectué par le comité de normalisation **X3J11** de l'**ANSI** qui a abouti en 1988 avec la parution par la suite du manuel :

« **The C programming Language - 2ème édition** ».

1.2 – Intérêts du langage

- Langage polyvalent permettant le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- Langage structuré.
- Langage évolué qui permet néanmoins d'effectuer des opérations de bas niveau (« assembleur d'Unix »).
- Portabilité (en respectant la norme !) due à l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine.
- Grande efficacité et puissance.
- Langage permissif ! ! !

1.3 – Qualités attendues d'un programme

- Clarté
- Simplicité
- Modularité
- Extensibilité

2 – Généralités

2.1 – Jeu de caractères

- 26 lettres de l'alphabet (minuscules, majuscules)
- chiffres 0 à 9
- caractères spéciaux :

!	*	+	\	"	<
#	(	=	|	{	>
%	)	~	;	]	/
^	-	[	:	,	?
&	_	}	'	.	(espace)

- séquences d'échappement telles :
  - passage à la ligne (`\n`),
  - tabulation (`\t`),
  - backspace (`\b`).

2.2 – Identificateurs et mots-clés

☞ Identificateur :

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

- ⇾ Formé de lettres et de chiffres ainsi que du caractère _ permettant une plus grande lisibilité. Le 1ercaractère doit obligatoirement être une lettre ou bien le caractère _.
- ⇾ Peut contenir jusqu'à 31 caractères minuscules et majuscules.
- ⇾ Il est d'usage de réserver les identificateurs entièrement en majuscules aux variables du préprocesseur.

Exemples

- Identificateurs valides :

x	y12	somme_1	_temperature
noms	surface	fin_de_fichier	`TABLE`

- Identificateurs invalides :

4eme	commence par un chiffre
x#y	caractère non autorisé (#)
no-commande	caractère non autorisé (-)
taux change	caractère non autorisé (espace)

☞ Mots réservés (mots-clés)

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

2.3 – Structure d'un programme C

☞ Programme C :

une ou plusieurs **fonctions** dont l'une doit s'appeler **main** stockées dans un ou plusieurs fichiers.

☞ Fonction :

- ⇾ entête (type et nom de la **fonction** suivis d'une liste d'arguments entre parenthèses),
- ⇾ instruction composée constituant le corps de la **fonction**.

- Instruction composée :

  délimitée par les caractères { et }
- Instruction simple :

  se termine par ;
- commentaire :

  encadré par les délimiteurs /* et */
- Instruction préprocesseur :

  commence par #

Exemple :

```
#include <stdio.h>
#define  PI 3.14159
/*  calcul de la surface d'un cercle */
main()
{
  float rayon, surface;
  float calcul(float rayon);

  printf("Rayon = ? ");
  scanf("%f", &rayon);
  surface = calcul(rayon);
  printf("Surface = %f\n", surface);
}
/* définition de fonction */
float calcul(float r)
{
/* définition de la variable locale */
  float a;

  a = PI * r * r;
  return(a);
}
```

2.4 – Compilation et édition des liens

Le source d'une application écrite en langage C peut être stocké dans un ou plusieurs fichiers dont le suffixe est « **.c** »

La compilation de ce source s'effectue à l'aide de la commande **cc**. Sans autre spécification, cette commande enchaîne 3 étapes :

❶ appel au pré-processeur (**cpp**),

❷ appel au compilateur,

❸ appel à l'éditeur de liens.

Cette commande admet plusieurs options telles :

- **-E** permet de n'exécuter que la première étape. Le résultat s'affiche sur la sortie standard. Il est commode de rediriger la sortie dans un fichier,
- **-P** idem **-E** mais la sortie est stockée dans un fichier dont le nom est celui du source suffixé par « **.i** »,
- **-c** permet l'exécution des 2 premières étapes uniquement, on récupère alors le module objet stocké dans un fichier dont le nom est identique à celui du source mais suffixé par « **.o** »,
- **-O** permet d'activer l'optimiseur,
- **-o** permet de renommer le module exécutable, le nom par défaut étant « **a.out** ».

3 – Les déclarations

3.1 – Les types de base

Le langage contient des types de base qui sont les **entiers**, les **réels** simple et double précision et les caractères que l'on identifie à l'aide des mots-clés **int**, **float**, **double** et **char** respectivement.

De plus il existe un type **ensemble vide** : le type **void**.

Les mots-clés **short** et **long** permettent d'influer sur la taille mémoire des entiers et des réels.

TAB. 1 – Liste des différents types de base

Syntaxe	Type
void	ensemble vide
char	caractère
short int	entier court
int	entier par défaut
long int	entier long
float	réel simple précision
double	réel double précision
long double	réel précision étendue

Remarques

- La taille d'un **entier** par défaut est soit 2 soit 4 octets (dépend de la machine). 4 octets est la taille la plus courante.
- La taille d'un **entier court** est en général 2 octets et celle d'un **entier long** 4 octets.
- La taille d'un *entier court* est inférieure ou égale à la taille d'un *entier par défaut* qui est elle-même inférieure ou égale à celle d'un *entier long*.
- Les types **short int** et **long int** peuvent être abrégés en **short** et **long**.
- Le type **char** occupe un octet. Un caractère est considéré comme un entier qu'on pourra donc utiliser dans une expression arithmétique.

Les deux mots-clés **unsigned** et **signed** peuvent s'appliquer aux types caractère et entier pour indiquer si le bit de poids fort doit être considéré ou non comme un **bit de signe**.

Les entiers sont **signés** par défaut, tandis que les caractères peuvent l'être ou pas suivant le compilateur utilisé.

Une déclaration telle que **unsigned char** permettra de désigner une quantité comprise entre **0** et **255**, tandis que **signed char** désignera une quantité comprise entre **-128** et **+127**.

De même **unsigned long** permettra de désigner une quantité comprise entre **0** et $\mathbf{2^{32}-1}$, et **long** une quantité comprise entre $\mathbf{-2^{31}}$ et $\mathbf{2^{31}-1}$.

Exemple

```
#include <stdio.h>
unsigned char mask;
long          val;
main()
{
  unsigned int indice;
  float        x;
  double       y;
  char         c;
        ...
  return;
}
double f(double x)
{
  unsigned short taille;
  int            i;
  unsigned long  temps;
  double         y;
       ...
  return y;
}
```

3.2 – Les énumérations de constantes

Les **énumérations** sont des types définissant un ensemble de constantes qui portent un nom que l'on appelle **énumérateur**.

Elles servent à rajouter du sens à de simples numéros, à définir des variables qui ne peuvent prendre leur valeur que dans un ensemble fini de valeurs possibles identifiées par un nom symbolique.

Syntaxe

```
enum [nom]
{
   énumérateur1,
   énumérateur2,
   énumérateur3,
        ...
   énumérateurn
};
```

Les constantes figurant dans les **énumérations** ont une valeur entière affectée de façon automatique par le compilateur en partant de **0** par défaut et avec une progression de **1**.

Les valeurs initiales peuvent être forcées lors de la définition.

```
enum couleurs {noir, bleu, vert, rouge, blanc,
               jaune};
enum couleurs
{
   noir = -1,
   bleu,
   vert,
   rouge = 5,
   blanc,
   jaune
};
```

Dans le 1^er exemple, les valeurs générées par le compilateur seront :

noir	**0**	vert	**2**	blanc	**4**
bleu	**1**	rouge	**3**	jaune	**5**

et dans le 2^e :

noir	**-1**	vert	**1**	blanc	**6**
bleu	**0**	rouge	**5**	jaune	**7**

3.3 – Les pointeurs

Un **pointeur** est une variable ou une constante dont la valeur est une adresse.

L'adresse d'un objet est indissociable de son **type**. On pourra se définir par exemple des *pointeurs de caractères*, des *pointeurs d'entiers* voire des *pointeurs* d'objets plus complexes.

L'opération fondamentale effectuée sur les *pointeurs* est l'**indirection**, c'est-à-dire l'évaluation de l'objet pointé. Le résultat de cette *indirection* dépend du **type** de l'objet pointé.

Par exemple si **p_car** et **p_reel** sont respectivement un *pointeur* de caractères et un *pointeur* de réel simple précision référençant la même adresse α, une *indirection* effectuée sur *p_car* désignera le caractère situé à l'adresse α, tandis qu'une *indirection* effectuée sur *p_reel* désignera le réel simple précision située à la même adresse.

Bien qu'ayant le même contenu (l'adresse α), ces deux *pointeurs* ne sont pas identiques !

3.4 – Forme générale d'une déclaration

La forme générale d'une déclaration est la suivante :

```
< type >  < construction > [,< construction >,...];
```

où **type** est un type *élémentaire* (type de base, énumération de constantes) ou un type que l'on s'est défini, et **construction** est soit un identificateur soit un objet plus complexe construit à l'aide de **constructeurs homogènes**.

3.5 – Constructeurs homogènes

Des objets plus complexes peuvent être formés à l'aide des **constructeurs homogènes** :

- ☞ les constructeurs de **pointeurs**,
- ☞ les constructeurs de **vecteur**,
- ☞ les constructeurs de **fonction**.

TAB. 2 – Symboles associés aux constructeurs homogènes

Symbole	Objet construit
$*$	pointeur
[ ]	vecteur
( )	fonction

Exemple

```
char     lignes[100];
int     *p_entier;
double   fonc();
```

Les déclarations précédentes permettent de définir respectivement :

- un **vecteur** de 100 caractères,
- un **pointeur** d'entier,
- une **fonction** retournant un réel double précision.

Ces constructeurs peuvent se combiner entre eux, permettant ainsi de définir des objets encore plus complexes.

Exemple

```
char    *chaines[100];
int      mat[100][40];
char   **argv;
```

Le *constructeur homogène* $*$ est moins prioritaire que les deux autres.

De ce fait, les déclarations précédentes permettent de définir respectivement :

- un **vecteur** de 100 pointeurs de caractère,
- un **vecteur** de 100 éléments, chaque élément étant un vecteur de 40 entiers,
- un **pointeur** de pointeur de caractère.

L'utilisation de parenthèses permet de modifier la priorité et donc l'ordre d'évaluation.

Exemple

```
int    (*tab)[10];
char   (*f)();
char   *(*g)();
float  *(*tabf[20])();
```

Cet exemple permet de définir respectivement :

- un **pointeur** de vecteur de 10 entiers,
- un **pointeur** de fonction retournant un caractère,
- un **pointeur** de fonction retournant un pointeur de caractère,
- un **vecteur** de 20 **pointeurs** de fonction retournant un **pointeur** de réel simple précision.

3.6 – Constructeurs hétérogènes

Les **constructeurs hétérogènes** permettent de définir des objets renfermant des entités de nature différente.

Il en existe 3 :

- les **structures**,
- les **champs de bits**,
- les **unions**.

Les **structures** permettent de regrouper des objets dont les types peuvent être différents.

Syntaxe

```
struct [ nom ]
{
    < liste de déclarations >
};
```

Les objets regroupés sont les **membres** ou **composantes** de la *structure* les contenant.

Remarques

- ☞ Les *structures* sont un exemple de définition de nouveaux types.
- ☞ Lors de la définition d'une *structure* des objets peuvent être déclarés et seront du type associé à celle-ci.

- Ils peuvent être déclarés ultérieurement mais dans ce cas la *structure* devra nécessairement avoir un *nom* et *struct nom* est le nom du type associé.
- La taille d'une *structure* est au moins égale à la somme des tailles de ses membres du fait d'éventuels alignements mémoires. L'opérateur **sizeof** permet d'en connaître la taille.

Exemple

```
struct
{
    char          c;
    unsigned int  i;
    float         tab[10];
    char         *p;
} a, b;
```

Exemples

```
struct cellule
{
    char  **p;
    int    *t[10];
    int   (*f)();
};

struct cellule cel1, *cel2;
struct cellule cel[15];

struct boite
{
    struct cellule  cel1;
    struct cellule *cel2;
    struct boite   *boite_suivante;
    int             ent;
} b1, b2, *b3;
```

Un **champ de bits** est un ensemble de bits contigus à l'intérieur d'un même mot.

Le constructeur de structures permet de définir un découpage mémoire en *champs de bits*. Les membres de cette structure désignent les différents *champs de bits*. Ils doivent être du type *unsigned int* et indiquer le nombre de bits de chaque champ.

Syntaxe

```
struct [ nom ]
{
    unsigned int champ1 : longueur1;
    unsigned int champ2 : longueur2;
                ...
    unsigned int champn : longueurn;
};
```

Exemple

```
struct
{
    unsigned int actif  :  1;
    unsigned int type   :  3;
    unsigned int valeur : 14;
    unsigned int suivant: 14;
} a, b;
```

Un champ peut ne pas avoir de nom. Sa longueur indique alors le nombre de bits que l'on veut ignorer.

Une longueur égale à **0** permet de forcer l'alignement sur le début du mot mémoire suivant.

Exemple

```
struct zone
{
    unsigned int a:  8;
    unsigned int  :  0;
    unsigned int b:  8;
    unsigned int  :  8;
    unsigned int c: 16;
};
struct zone z1, *z2;
```

Remarques

- Les *champs de bits* sont évalués de gauche à droite sur certaines machines et de droite à gauche sur d'autres. Ce type de données n'est donc pas portable.
- On ne peut pas référencer les champs via une adresse.

Le constructeur **union** permet de définir des données de type différent ayant la même adresse mémoire.

Syntaxe

```
union [ nom ]
{
    < liste de déclarations >
};
```

Remarques

- A la définition d'une *union* est associé un nouveau type : *union nom* lorsque *nom* a été précisé à la définition.
- La taille d'une *union* est celle de la composante ayant la taille maximum.

Exemples

```
struct complexe
{
    float x;
    float y;
};

union valeur
{
    long   entier;
    float  reel;
    struct complexe cmplx;
};

enum type {Entier, Reel, Complexe};

struct nombre
{
    enum type    type;
    union valeur valeur;
};

struct nombre n;
```

Exemples

```
struct zone
{
    int    nb_parm;
    char **parm;
    union
    {
      unsigned char mask;
      struct
      {
         unsigned int a: 1;
         unsigned int b: 1;
         unsigned int c: 1;
         unsigned int d: 1;
         unsigned int e: 1;
         unsigned int f: 1;
         unsigned int g: 1;
         unsigned int h: 1;
      } drapeaux;
    } infos;
} z1, *z2;
```

3.7 – Définitions de types

Il existe plusieurs manières de se définir de nouveaux types :

- ☞ au moyen des *constructeurs hétérogènes* **struct** et **union**,
- ☞ au moyen du constructeur **typedef**,
- ☞ au moyen d'**expressions de type**.

A la différence des *constructeurs hétérogènes* qui créent de nouveaux types, le constructeur *typedef* permet seulement de donner un nouveau nom à un type déjà existant.

Syntaxe

```
typedef < déclaration >
```

Exemples

```
typedef long              size_t;
typedef unsigned long     Cardinal;
typedef char             *va_list;
typedef struct complexe  Complexe;
typedef int              Matrice[10][20];


Complexe   c1, *c2;
Cardinal   nombre;
va_list    arg;
size_t     dimension;
Matrice    tab, *ptr_mat;

typedef struct cellule
{
    Cardinal         n;
    struct cellule *ptr_suivant;
} Cellule;

Cellule cel1, *cel2;
```

Une **expression de type** est une expression construite en retirant l'objet de la déclaration qui le définit.

Exemples

```
char      *c;
int       (*f)();
char      (*tab)[100];
char      (*(*x())[6])();
char      (*(*vec[3])())[5];
Complexe  (**ptr)[5][4];
```

Les types des objets déclarés précédemment sont donnés par les *expressions de types* suivant :

```
char      *
int       (*)()
char      (*)[100]
char      (*(*())[6])()
char      (*(*[3])())[5]
Complexe  (**)[5][4]
```

4 – Expressions et opérateurs

4.1 – Constantes littérales

Constantes entières

Une constante entière peut s'écrire dans les systèmes **décimal**, **octal** ou **hexadécimal**.

Une constante entière préfixée :

- ☞ du chiffre **0** est une constante *octale*,
- ☞ des caractères **0x** ou **0X** est une constante *hexadécimale*.

Une constante entière est par défaut de type *int*. Elle est de type *long* si elle est suffixée par les lettres **l** ou **L** et non signée lorsqu'elle est suffixée par les lettres **u** ou **U**.

Exemples

base	**10**	22	56	1789	32765
		22u	56L	29UL	1L
base	**8**	0643	0177	0644	0755
		0177L	0222UL	0777u	0766ul
base	**16**	0xff	0Xabcd	0x80	0X1
		0xffL	0X1uL	0X7fU	0x5fUL

Constantes réelles

Une constante **réelle** (ou constante en virgule flottante) est un nombre exprimé en base 10 contenant un point décimal et éventuellement un exposant séparé du nombre par la lettre **e** ou **E**.

Une constante **réelle** est par défaut de type *double*. Elle sera du type *float* si on la suffixe par la lettre **f** ou **F**.

Exemples

0.	1.	0.2	1789.5629
50000.	0.000743	12.3	315.0066
2E-8	0.006e-3	1.66E+8	3.1415927
1.6021e-19f	6.0225e23F	2.718281	6.6262e-34

Constantes caractères

Une constante caractère est assimilée à un entier sur un octet dont la valeur correspond au rang du caractère dans la table ASCII.

Une constante caractère est constituée soit :

- d'un caractère entre apostrophes ;
- d'une suite de deux caractères entre apostrophes dont le premier est le caractère ****. Ces caractères s'appellent des *codes d'échappement* ;
- d'un mot de la forme **'\nnn'**, *nnn* étant la valeur *octale* de l'entier associé au caractère ;
- d'un mot de la forme **'\xnn'**, *nn* étant la valeur *hexadécimale* de l'entier associé au caractère.

TAB. 3 – Séquences d'échappement

Syntaxe	Séq. d'éch.	Code ASCII
sonnerie	`\a`	7
retour arrière	`\b`	8
tabulation h.	`\t`	9
tabulation v.	`\v`	11
retour à la ligne	`\n`	10
nouvelle page	`\f`	12
retour chariot	`\r`	13
guillemets	`\"`	34
apostrophe	`\'`	39
point d'interr.	`\?`	63
antislash	`\\`	92
caractère nul	`\0`	0

Exemples

		Valeur entière associée
'A'	==>	65
'x'	==>	120
'3'	==>	51
'$'	==>	36
' '	==>	32
'\n'	==>	10
'\t'	==>	9
'\b'	==>	8
'\"'	==>	34
'\\'	==>	92
'\''	==>	39
'\0'	==>	0
'\177'	==>	127
'\x0a'	==>	10
'\000'	==>	0

Constantes chaîne de caractères

Une constante chaîne de caractères est une suite de caractères entre *guillemets*.

En mémoire cette suite de caractères se termine par le caractère **NULL** (**'\0'**).

La valeur d'une chaîne de caractères est l'*adresse* du premier caractère de la chaîne qui est donc du type *pointeur de caractères* (**char ***).

Ne pas confondre "**A**" et '**A**' qui n'ont pas du tout la même signification !

Pour écrire une chaîne de caractères sur plusieurs lignes on peut :

- soit terminer chaque ligne par ****,
- soit la découper en plusieurs constantes chaîne de caractères, le compilateur effectuera automatiquement la concaténation.

Exemples

```
char *chaine = "\
\n\
\t/-------------------------------------------\\\n\
\t| Pour écrire une chaîne sur plusieurs lignes, |\n\
\t|   il suffit de terminer chaque ligne par \\  |\n\
\t\\-------------------------------------------/\n";

char *chaine = "écriture d'une "
               "chaîne de caractères "
               "sur plusieurs "
               "lignes\n\n";
```

4.2 – Constantes symboliques

Les **constantes symboliques** sont de plusieurs types :

- les *constantes énumérées*,
- les *identificateurs de vecteur* dont la valeur est l'adresse du premier élément du vecteur,
- les *identificateurs de fonction* dont la valeur est l'adresse de la première instruction machine de la fonction,
- les objets qui ont été déclarés avec l'attribut **const**.

Exemple

```
char     tab[100];
double   func(int i)
{
        ...
}
const    int   nombre = 100;
const    char *ptr1;
char    const *ptr2;
char * const   ptr3 = tab;
```

Les objets précédents sont respectivement :

- un *identificateur de vecteur*,
- un *identificateur de fonction*,
- un entier constant,
- deux pointeurs sur un caractère constant,
- un pointeur constant de caractères.

4.3 – Opérateurs arithmétiques

Une **expression** est constituée de *variables* et *constantes* (*littérales* et/ou *symboliques*) reliées par des **opérateurs**.

Il existe 5 opérateurs arithmétiques :

- l'addition (+),
- la soustraction (-),
- la multiplication (*),
- la division (/),
- le reste de la division entière (%).

Leurs *opérandes* peuvent être des *entiers* ou des *réels* hormis ceux du dernier qui agit uniquement sur des *entiers*.

Lorsque les types des deux *opérandes* sont différents, il y a *conversion implicite* dans le type le plus fort suivant certaines règles.

Règles de conversions

- si l'un des opérandes est de type **long double**, convertir l'autre en **long double**,
- sinon, si l'un des opérandes est de type **double**, convertir l'autre en **double**,
- sinon, si l'un des opérandes est de type **float**, convertir l'autre en **float**,
- sinon, si l'un des opérandes est de type **unsigned long int**, convertir l'autre en **unsigned long int**,

- sinon, si l'un des opérandes est de type **long int** et l'autre de type **unsigned int**, le résultat dépend du fait qu'un **long int** puisse représenter ou non toutes les valeurs d'un **unsigned int** ; si oui, convertir l'opérande de type **unsigned int** en **long int** ; si non, convertir les deux opérandes en **unsigned long int**,
- sinon, si l'un des opérandes est de type **long int**, convertir l'autre en **long int**,
- sinon, si l'un des opérandes est de type **unsigned int**, convertir l'autre en **unsigned int**,
- sinon, les deux opérandes sont de type **int**.

Les opérateurs `+` et `-` admettent des opérandes de type *pointeur*, ceci pour permettre notamment de faire de la *progression d'adresse*.

Opérateur	Op. 1	Op. 2	Résultat
+	*pointeur*	*entier*	*pointeur*
+	*entier*	*pointeur*	*pointeur*
-	*pointeur*	*entier*	*pointeur*
-	*pointeur*	*pointeur*	*entier*

Exemples

```
char  *pc;
int   *pi;
int    a, b, c;
    ...
    ...
c  = 2*a + b%2;
pc = pc + a;
pi = pi - c;
```

4.4 – Opérateurs logiques

Le type *booléen* n'existe pas. Le résultat d'une expression logique vaut `1` si elle est *vraie* et `0` sinon.

Réciproquement toute valeur *non nulle* est considérée comme *vraie* et la valeur *nulle* comme *fausse*.

Les opérateurs logiques comprennent :

- 4 opérateurs *relationnels* :
  - inférieur à (`<`),
  - inférieur ou égal à (`<=`),
  - supérieur à (`>`),
  - supérieur ou égal à (`>=`).
- l'opérateur de *négation* (`!`).

- 2 opérateurs de *comparaison* :
  - identique à (**==**),
  - différent de (**!=**).
- 2 opérateurs de *conjonction* :
  - le **et** logique (**&&**),
  - le **ou** logique (**||**).

Le résultat de l'expression :

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

Exemple

```
int   i;
float f;
char  c;

i = 7;
f = 5.5;
c = 'w';

                  Expressions :
                  -----------

f > 5                          ==>  vrai (1)
(i + f) <= 1                   ==>  faux (0)
c == 119                       ==>  vrai (1)
c != 'w'                       ==>  faux (0)
c >= 10*(i + f)                ==>  faux (0)
(i >= 6) && (c == 'w')         ==>  vrai (1)
(i >= 6) || (c == 119)         ==>  vrai (1)
(f < 11) && (i > 100)          ==>  faux (0)
```

4.5 – Opérateur de taille

L'opérateur **sizeof** renvoie la taille en octets de son opérande.

L'opérande est soit une expression soit une expression de type.

Syntaxe

```
sizeof  expression
sizeof (expression-de-type)
```

L'opérateur *sizeof* appliqué à une constante chaîne de caractères renvoie le nombre de caractères de la chaîne y compris le caractère `NULL` de fin de chaîne.

Si **p** est un *pointeur* sur un type t et **i** un *entier* :

l'expression	`p + i`
a pour valeur	`p + i*sizeof(t)`

Exemples

```
int menu[1000];
typedef struct cel {
      int          valeur;
      struct cel *ptr;
} Cel;

sizeof menu / sizeof menu[0]
                    ==> nombre d'éléments
                        du vecteur menu.

sizeof(long)        ==> taille d'un entier long.
sizeof(float)       ==> taille d'un flottant
                        simple précision.
sizeof(struct cel)  ==> taille d'un objet du
                        type struct cel.
sizeof(Cel)         ==> taille d'un objet du
                        type Cel.
```

4.6 – Opérateurs d'ad. et d'ind.

L'opérateur **&** appliqué à un objet renvoie l'*adresse* de cet objet.

L'opérateur ***** s'applique à un *pointeur* et permet d'effectuer une *indirection* c'est-à-dire retourne l'*objet pointé*.

Si *vect* est un vecteur, la valeur de la constante symbolique *vect* est égale à **&vect[0]**.

Si **a** est un objet de type **t**, **&a** est de type **t ***.

Réciproquement, si **p** est un objet de type **t ***, ***p** est de type **t**.

Exemples

```
int  u;
int  v;
int *pu;
int *pv;
typedef struct cel
{
  int          valeur;
  struct cel *ptr;
} Cel;
Cel  c1, *c2;

u  = 3 ;
pu = &u ;
v  = *pu ;
pv = &v ;
c2 = &c1 ;
```

4.7 – Opérateur de forçage de type

Il est possible d'effectuer des *conversions explicites* à l'aide de l'opérateur de *forçage de type* ou **cast**.

Syntaxe

```
(type) expression
(expression-de-type) expression
```

Exemples

```
int    n;
int    tab[100];
int    (*p)[2];
double puissance;

n = 10;
puissance = pow((double)2, (double)n);
p = (int (*)[2])tab;
**(p+49)      = 1756;
*(*(p+49)+1) = 1791;
```

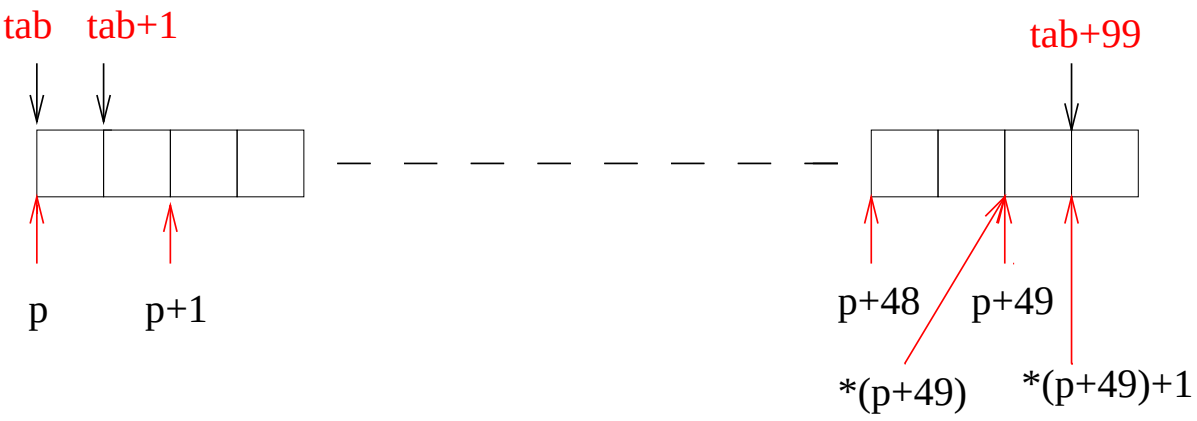

FIG. 1 – Schéma d'adressage

La fonction **pow** est la fonction *exponentiation*, elle renvoie 2^n dans l'exemple précédent.

4.8 – Opérateurs de manipulations de bits

Opérateurs arithmétiques "bit à bit"

Ils correspondent aux 4 opérateurs classiques de l'arithmétique booléenne :

- le *non logique* (`~`),
- le *et logique* (`&`),
- le *ou logique* (`|`),
- le *ou exclusif* (`^`).

Les opérandes sont de type *entier*. Les opérations s'effectuent *bit à bit* suivant la logique binaire.

b1	b2	~b1	b1&b2	b1|b2	b1^b2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Exemples

```
int a, b, c, flag;
int Mask;

a = 0x6db7;  |-0-|-0-|0110 1101|1011 0111|
b = 0xa726;  |-0-|-0-|1010 0111|0010 0110|
c = a&b ;    |-0-|-0-|0010 0101|0010 0110| (0x2526)
c = a|b ;    |-0-|-0-|1110 1111|1011 0111| (0xefb7)
c = a^b ;    |-0-|-0-|1100 1010|1001 0001| (0xca91)
flag = 0x04;
c = Mask & flag;
c = Mask & ~flag;
```

Opérateurs de décalage

Il existe 2 opérateurs de décalage :

- décalage à droite (`>>`),
- décalage à gauche (`<<`).

Le motif binaire du 1er opérande, qui doit être un *entier*, est décalé du nombre de bits indiqué par le 2e opérande.

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

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

Exemple

```
int  etat;
int  oct;
int  ent;
int  a;

a = 0x6db7;       |-0-|-0-|0110 1101|1011 0111|
a = a << 6;       |-0-|0001 1011|0110 1101|1100 0000|
                          (0x1b6dc0)
a = 0x6db7;
a = a >> 6;       |-0-|-0-|0000 0001|1011 0110|
                          (0x1b6)
ent = 0xf0000000;
ent = ent >> 10; |1111 1111|1111 1100|-0-|-0-|
                          (0xfffc0000)
oct = (etat >> 8) & 0xff;
```

4.9 – Opérateurs à effet de bord

Ce sont des opérateurs qui modifient l'ensemble des valeurs courantes des variables intervenant dans l'évaluation d'expressions.

Opérateurs d'affectation

Les opérateurs d'affectation sont :

- L'affectation simple (`=`).
- Les affectations combinées :

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

L'affectation est une expression. La valeur de ce type d'expression est la valeur de l'expression située à droite de l'affectation.

On appelle **g-valeur** toute expression pouvant figurer à gauche d'une affectation.

Un identificateur de vecteur n'est pas une *g-valeur*.

Une expression de la forme :

`e1 op= e2` est équivalente à
`e1 = e1 op e2`

Exemple

```
int           valeur;
int           i;
char          c;
unsigned char masque;
int           n;

i = 2; n = 8;
  ...
i += 3;
n -= 4;
valeur >>= i;
c      &= 0x7f;
masque |= 0x1 << (n - 1);
masque &= ~(0x1 << (n - 1));
```

Opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation (`++`) et de décrémentation (`--`) sont des opérateurs *unaires* permettant respectivement d'ajouter et de retrancher `1` au contenu de leur opérande.

Cette opération est effectuée *après* ou *avant* l'évaluation de l'expression suivant que l'opérateur *suit* ou *précède* son opérande.

Ces opérateurs ne s'appliquent qu'à des *g-valeurs*.

Exemple

```
int   i;
int   j;
int   tab[100];
char  buffer[2048];
char *ptr;
int  *p_ent;

i = 99;
j = 2;
i++;
p_ent  = tab;
*(p_ent + --i) = ++j;
ptr    = buffer;
*ptr++ = '\n';
```

4.10 – Opérateur conditionnel

L'opérateur conditionnel (`? :`) est un opérateur *ternaire*. Ses opérandes sont des expressions.

Syntaxe

```
expr1 ? expr2 : expr3
```

La valeur de l'expression *expr1* est interprétée comme un booléen. Si elle est *vraie*, c'est-à-dire non nulle, seule l'expression *expr2* est évaluée sinon c'est l'expression *expr3* qui est évaluée.

La valeur de l'expression conditionnelle est la valeur de l'une des expressions *expr2* ou *expr3* suivant que l'expression *expr1* est *vraie* ou *fausse*.

Exemple

```
int i;
int indic;
int a, b;
int c;

...

indic = (i < 0) ? 0 : 100;
c    += (a > 0 && a <= 10) ? ++a : a/b;
c     = a > b ? a : b;
```

4.11 – Opérateur séquentiel

L'opérateur séquentiel (,) permet de regrouper des sous-expressions sous forme de liste. Ces sous-expressions sont évaluées en séquence.

La valeur d'une liste d'expressions est celle de la dernière de la liste.

Exemple

```
int    i;
float  r;
double dble, d;
char  *ptr;
char   buffer[100];

d = (i = 1, r = 2.718f, dble = 2.7182818);
r = (float)(ptr = buffer, i = 10);
```

4.12 – Autres opérateurs

Opérateur d'indexation

Cet opérateur (**[ ]**) permet de référencer les différents éléments d'un vecteur.

C'est un opérateur *binaire* dont l'un des opérandes est un *identificateur de vecteur* ou un *pointeur* et l'autre opérande un *entier*.

Si **p** est un *pointeur* ou un identificateur de vecteur et **n** un *entier*, alors l'expression **p[n]** désigne le $(n+1)^e$ élément à partir de l'adresse p, c'est-à-dire l'élément situé à l'adresse $p+n$.

Cette expression est donc équivalente à ***(p+n)**.

Exemple

```
char  buffer[4096];
char *ptr;
int   i;
int   ta[5], tb[5], tc[5];

buffer[10] = 'a';
*(buffer + 9) = 'b';
i = 0;
buffer[i++] = 'c';
i  += 15;
ptr = buffer + i;
ptr[0] = 'd';
*++ptr = 'e';
*ta = 1, *tb = 2;
tc[0] = ta[0] + tb[0];
```

Opérateur d'appel de fonction

C'est l'opérateur () qui permet de déclencher l'appel à la fonction dont le nom précède.

Ce nom peut être soit un *identificateur de fonction*, soit un *pointeur de fonction*.

A l'intérieur de ces parenthèses, apparaît éventuellement une liste d'expressions appelées *paramètres* qui sont évaluées puis transmises à la fonction.

L'ordre d'évaluation de ces expressions est indéterminé.

Exemple

```
char *f( int i, float x );
char *(*pf)( int i, float x );
char *ptr;
int   i;
float x;

i = 2, x = 1.;
ptr = f( i, x );
pf = f;
i = 10, x *= 2.;
ptr = (*pf)( i, x );
/* non portable */
ptr = (*pf)( i++, x = (float)i );
/*                */
i++;
ptr = (*pf)( i, x = (float)i );
```

Opérateurs de sélection de champ

L'opérateur **.** permet d'accéder aux *champs* d'une structure ou d'une union.

Il est *binaire*. Le 1^er^ opérande doit être une *structure* ou une *union* et le 2^e^ opérande doit désigner l'un de ses champs.

Le type et la valeur de l'expression **op1.op2** sont ceux de *op2*.

Pour accéder à un champ *ch* d'une structure ou d'une union pointée par un pointeur *ptr*, on écrira l'expression **(∗ptr).ch** qui est équivalente à **ptr->ch**.

Exemple

```
typedef struct cellule
{
    int    n;
    char  *c;
    int    nb_parm;
    char **parm;
} Cel, *PtrCel;
typedef struct boite
{
    int    nb_boite;
    PtrCel cel;
} Boite;
Cel    c1;
PtrCel ptr_cel;
Boite  b[10], *p_boite;
c1.n          = 10;
c1.c          = "nom de la cellule";
ptr_cel       = &c1 ;
ptr_cel->n    = 20;
b[0].cel      = ptr_cel;
b->nb_boite   = 5;
b[0].cel->n++;
b->cel->n++;
b[1] = b[0]; p_boite = &b[1];
p_boite->cel = (PtrCel)0;
```

5 – Portée et visibilité

5.1 – Niveau d'une variable

Le **niveau** d'une variable est déterminé par l'emplacement de sa déclaration dans le programme.

- ☞ Une variable est de *niveau* **0** lorsqu'elle est déclarée à l'extérieur de toute fonction. Ces variables sont dites **externes**.
- ☞ Une variable est de *niveau* **n (n >= 1)** lorsqu'elle est déclarée à l'intérieur d'un **bloc**. Ces variables sont dites **internes**.

Exemple

```
Cardinal  nb_elements;      /* niveau 0 */
size_t    taille;           /* niveau 0 */
main()
{
    int   i, j;             /* niveau 1 */
    char  c;                /* niveau 1 */
    {
        Complexe  c1, *c2; /* niveau 2 */
        int       i;        /* niveau 2 */
        if (...)
        {
          char car;         /* niveau 3 */
            ...
        }
    }
      ...
}
int   ent;                  /* niveau 0 */
void f(void)
{
    long  i;                /* niveau 1 */
      ...
}
```

5.2 – Durée de vie d'une variable

La **durée de vie** d'une variable est le temps pendant lequel cette variable a une existence en mémoire.

Une variable peut avoir une *durée de vie* :

- **permanente** ou **statique**. L'emplacement mémoire de la variable est alloué lors de la compilation (voire lors de l'édition des liens) et de ce fait existe pendant toute la durée du programme ;
- **temporaire** ou **dynamique**. L'emplacement mémoire de la variable est alloué lors de l'appel de la fonction dans laquelle elle est définie et libéré lors du retour de la fonction.

5.3 – Classes de mémorisation

Il existe quatre **classes de mémorisation** :

- **static**,
- **extern**,
- **auto**,
- **register**.

Les classes *static* et *extern* concernent les variables permanentes.

Les classes *auto* et *register* concernent les variables temporaires.

C'est au moment de la déclaration des variables que ces attributs peuvent être spécifiés.

Exemples

```
extern          int i;
static unsigned int j;
register        int n;
```

5.4 – Portée d'une variable interne

La **portée** ou la **visibilité** d'une variable est l'endroit du programme où elle existe et est accessible.

La portée d'une variable *interne* est le bloc où elle est déclarée ainsi que tous les blocs contenus dedans à l'exception de ceux dans lesquels cette variable a fait l'objet d'une redéclaration.

Pour déclarer une variable *interne permanente* il suffit de mentionner l'attribut **static** lors de sa déclaration.

Par défaut, en l'absence d'attribut de classe mémoire, une variable *interne* est *temporaire* et reçoit l'attribut **auto**. Ce type de variable est alloué dynamiquement dans le « **stack** » ou **pile** d'exécution (pile de type **LIFO**).

Lorsqu'une variable est très utilisée, il peut être avantageux de demander au compilateur qu'elle soit, dans la mesure du possible, rangée dans un **registre** de la machine.

Cette possibilité, qui ne peut s'appliquer qu'à des variables *temporaires*, ne sera satisfaite que s'il existe des *registres* disponibles au format de la donnée.

C'est l'attribut **register** spécifié à la déclaration qui permet de formuler une telle demande.

Exemple

```
main()
{
  int a = 1, b = 2;
  a++, b++;
  {
    char b = 'A'; int  x = 10;
    a++, b++, x++;
    {
      int a = 100, y = 200;
      a++, b++, x++, y++;
      {
        char a = 'L'; int  z = -5;
        a++, b++, x++, y++, z++;
      }
      a++, b++, x++, y++;
    }
    a++, b++, x++;
  }
}
```

5.5 – Portée d'une variable externe

A - Programme monofichier

La *portée* d'une variable externe est l'ensemble du source à partir de l'endroit où celle-ci a été déclarée.

Cela implique que seules les fonctions définies après la déclaration des variables externes peuvent y accéder.

Dans une fonction, une variable externe est *masquée* lorsqu'elle subit une redéclaration au sein de cette fonction.

Exemple

```
int i = 10;
main()
{
    ...             /* la variable externe r
                       n'est pas visible. */
  {
     int i = 20;    /* dans ce bloc la variable
                       externe i est masquée. */
         ...
  }
}
float r = 3.14;
void f(...)
{
    ...
  {
     double r;      /* dans ce bloc la variable
                       externe r est masquée. */
         ...
  }
    ...
}
```

Dans une fonction, il est possible de rendre une variable *externe* visible si elle ne l'était pas déjà. Il suffit de la *référencer* en indiquant l'attribut *extern*.

Exemple

```
double y = 10.123;
   ...
main()
{
  int y;     /* y déclarée en externe
                est masquée. */
   ...
  {
    extern double y; /* On rend la variable
                        externe y a nouveau
                        accessible.  */
      ...
  }
   ...
}
```

Exemple

```
main()
{
   ...      /* la variable externe z
               n'est pas visible */
}
void f(void)
{
   extern float z; /* la variable externe
                      z est visible dans f. */
       ...
}
int g(void)
{
   ...      /* la variable externe z
               n'est pas visible */
}
float z = 2.0;
float h(int i)
{
   /* la variable externe z
      est visible dans h ainsi que
      dans les fonctions suivantes. */
}
```

B - Programme multifichiers

L'unité de compilation est le fichier. Les différents fichiers sources constituant une application sont donc traités de façon indépendante par le compilateur.

Lorsque l'on a besoin de partager une variable entre ces différents fichiers, on devra allouer son emplacement mémoire dans un seul de ces fichiers et la référencer dans les autres.

On parlera de **définition** lorsque la mémoire est allouée et de **déclaration** lors d'une référence.

Tous les compilateurs (avant et après la norme) considèrent :

- qu'une variable *initialisée* sans l'attribut *extern* fait l'objet d'une *définition*,
- qu'une variable avec l'attribut *extern* sans *initialisation* fait l'objet d'une simple *déclaration*.

Les compilateurs de l'*après-norme* admettent l'attribut *extern* dans les deux cas. Pour ceux-là une simple *initialisation* suffit pour indiquer une *définition*.

Exemple

source1.c

```
main()
{
    ...
}
extern int i; /* déclaration de la
                  variable externe i. */
void f(int a)
{
    ...
}
```

source2.c

```
int i = 11; /* définition de la
                variable externe i. */
int g(void)
{
   ...
}
```

source3.c

```
float h(void)
{
   ...
}
extern int i; /* déclaration de la
                 variable externe i. */
void func(void)
{
   ...
}
```

De plus la norme dit qu'une variable sans l'attribut *extern* et sans *initialisation* fait l'objet d'une *définition* **potentielle**.

Si pour une variable n'apparaissent que des *définitions potentielles*, l'une sera considérée comme *définition* et les autres comme *déclarations*. Cette variable sera initialisée avec des zéros binaires.

Exemple

sourceA **sourceB** **sourceC**

```
int x = 10;    | extern int x;     | extern int x;
extern int y;  | extern int y;     | extern int y;
int z;         | extern int z;     | extern int z;
int a;         |                   | int a;
int b = 20;    | int b = 21;       |
int c;         | int c = 30;       |
               |                   |
main()         | int calcul(void)  | int somme(void)
{              | {                 | {
    ...        |       ...         |      ...
}              | }                 | }
```

On peut limiter la *portée* d'une variable au source au sein duquel elle est définie. Pour cela on indiquera l'attribut **static** au moment de sa définition.

Exemple

sourceA

```
static float r = 2.154;
double dble = 17.89;
main()
{
   ...
}
float f1(void)
{
   ...
}
```

sourceB

```
void f2(void)
{
   ...
}
extern double dble;
int f3(int i)
{
   ...
}
static int n = 10;
void f4(float r)
{
   ...
}
```

5.6 – Initialisation des variables

Il est possible d'initialiser une variable lors de sa déclaration.

Syntaxe

```
type construction = expression;
```

L'initialisation des variables permanentes doit se faire à l'aide d'expressions constantes :

- une constante (littérale ou symbolique),
- une expression dont les opérandes sont des constantes.

Par contre l'initialisation des variables temporaires peut se faire à l'aide d'expressions quelconques.

Exemple

```
void f( void )
{
  static int   n    = 10 ;
  static char *ptr  = "Aix-en-Provence" ;
  static int  *p    = &n ;
  static int   etat = 0x1 «~5 ;
  int flag = etat;                  <==>  int flag;
                                          flag = etat;
      ...
}
```

L'initialisation des vecteurs, des structures ou des unions s'effectue au moyen de listes de valeurs entre accolades :

```
{val1, val2, ..., valn}
```

Si l'élément d'un vecteur est lui-même un vecteur on applique récursivement la notation précédente. L'initialisation des vecteurs doit se faire au moyen d'expressions constantes.

Seule la première composante d'une union peut être initialisée.

Exemple

```
int   tab1[5] = { 2, 6, 8, 10, 17};
int   tab2[]  = { 3, 7, 10, 18, 16, 3, 1};
char  v1[]    = "Wolfgang Amadeus Mozart";
char  v2[]    = "musique";
char  v3[]    = { 'm', 'u', 's', 'i',
                  'q', 'u', 'e', '\0' };
char *p       = "musique";
typedef struct date
{
   int jour, mois, annee;
} Date;
typedef struct
{
   char  sexe;
   char *nom;
   Date  annee_naiss;
} Individu;
Individu tab[] = {
           { 'F', "France Nathalie", { 1,  7, 65 } },
           { 'M', "Deneau Michel",   { 8, 10, 61 } }
                 };
union donnees
{
   int   i;
   float r;
} u = {2};
```

Exemple

```
int tab[3][4] = {
                  {1, 2,  7, 11},
                  {2, 3, 12, 13},
                  {4, 8, 10, 11}
                };
int t1[][4]   = {
                  {1},
                  {2, 3},
                  {4, 8, 10}
                };
int t2[3][4]  = {1, 0, 0, 0, 2, 3, 0, 0,
                 4, 8, 10, 0};
int t3[][3]   = {0, 1, 2, 3, 8, 9, 9, 1};
int t4[2][3][4] = {
                     {
                        {1, 2, 3, 8},
                        {3, 2},
                        {1}
                     },
                     {
                        {3, 4, 9},
                        {2}
                     } };
```

5.7 – Visibilité des fonctions

La **définition** d'une fonction comprend un entête (appelé **prototype**), indiquant le type de la valeur qu'elle retourne ainsi que la liste et le type des arguments transmis, et une **instruction composée** (appelée **corps** de la fonction), renfermant des déclarations d'objets et des instructions exécutables.

La **déclaration** d'une fonction s'effectue au moyen de son *prototype*.

Lors de l'appel à une fonction, pour que le compilateur connaisse le type de la valeur qu'elle retourne et puisse vérifier le nombre ainsi que le type des arguments transmis, il est nécessaire qu'il ait visibilité sur le **prototype** de cette fonction.

Cette visibilité existe lorsque :

- ☞ la définition de la fonction ainsi que son appel se situent dans le même fichier, la définition étant positionnée avant l'appel,
- ☞ une **déclaration** de la fonction apparaît avant son appel.

Lorsque que le compilateur n'a pas cette *visibilité*, il suppose que la valeur retournée par la fonction est de type **int**.

Une fonction ne peut être contenue dans une autre fonction, de ce fait toutes les fonctions sont *externes*. C'est pourquoi préciser l'attribut *extern*, que ce soit lors de la *déclaration* ou lors de la *définition* de la fonction, n'apporte aucune information supplémentaire.

A l'instar des variables, une fonction peut n'être connue que dans le fichier dans lequel elle a été définie. Pour cela on indiquera l'attribut *static* lors de sa *définition*.

Exemple

sourceA

```
float f(double d);
main()
{
   float  r;
   double d;

   r = f(d);
   ...
}
static float f(double d)
{
   int g(void);
   int i;

   i = g();
   ...
}
```

sourceB

```
int g(void)
{
   int i;
   int j;
   static int h(int i);
      ...
   j = h(i);
}
void func(int i)
 {
  /* la fonction h n'est
     pas visible ici. */
}
static int h(int i)
{
   ...
}
```

6 – Instructions

6.1 – Instructions élémentaires

Une **instruction élémentaire** est une expression terminée par un ;.

Contrairement aux expressions, les instructions n'ont ni type ni valeur. Lorsqu'on forme une instruction à partir d'une expression la valeur de cette dernière est perdue.

N'importe quelle expression peut former une instruction, même lorsqu'elle ne génère pas d'effet de bord.

Une *instruction composée* ou *bloc* est un ensemble d'*instructions élémentaires* et/ou *composées*, précédées éventuellement de déclarations, délimité par des accolades.

Exemple

```
#include <stdio.h>
#include <math.h>
main()
{
  int    i = 10;
  double r = acos(-1.);

  i *= 2;
  {
     double cosh_pi;

     cosh_pi = (exp(r) + exp(-r)) / 2;
     printf( "cosh_pi : %f\n", cosh_pi );
  }
}
```

6.2 – Structures de contrôle

Les **structures de contrôle** sont les *tests*, les *boucles* et l'*aiguillage*.

6.2.1 - Les tests : syntaxe

```
if (expression)
   partie-alors
[else
   partie-sinon]
```

La *partie-alors* et la *partie-sinon* peuvent être indifféremment une instruction élémentaire ou composée.

La *partie-alors* sera exécutée si la valeur de l'expression entre parenthèses est non nulle. Sinon, si le test comporte une *partie-sinon* c'est celle-ci qui sera exécutée.

Exemple

```
char buffer[2048];
void f( void )
{
   static char *p = (char *)0;

   if( ! p )
     p = buffer;
   else
   {
     *p = '1';
     p++;
   }
}
```

Si plusieurs tests sont imbriqués, chaque *partie-sinon* est reliée au *if* le plus proche qui n'est pas déjà associé à une *partie-sinon*.

Exemple

```
if( x > 0 )                          if( x > 0 )
  ecrire( "positif" );                 ecrire( "positif" );
else if( x < 0 )          <==>       else
  ecrire( "négatif" );                 {
else                                     if( x < 0 )
  ecrire( "nul" );                         ecrire( "négatif" );
                                         else
                                           ecrire( "nul" );
                                       }
```

6.2.2 - Les boucles « tant-que » : syntaxe

```
while (expression)
    corps-de-boucle

do
    corps-de-boucle
while (expression);
```

La partie *corps-de-boucle* peut être soit une instruction élémentaire soit une instruction composée.

Dans la boucle *while* le test de continuation s'effectue avant d'entamer le *corps-de-boucle* qui, de ce fait, peut ne jamais s'exécuter.

Par contre dans la boucle *do-while* ce test est effectué après le *corps-de-boucle*, lequel sera alors exécuté au moins une fois.

Exemple

```
#include <stdio.h>
main()
{
  int chiffre = 0;

  printf( "Boucle \"while\"\n\n" );
  while( chiffre )
  {
    printf( " %d", chiffre++ );
    if( ! (chiffre%5) )
      printf("\n");
  }
  printf( "Boucle \"do-while\"\n\n" );
  do
  {
    printf( " %3d", ++chiffre );
    if( ! (chiffre%5) )
      printf("\n");
    if( chiffre == 100 )
      chiffre = 0;
  }
  while( chiffre );
}
```

6.2.3 - La boucle « pour » : syntaxe

```
for ([expr1]; [expr2]; [expr3])
    corps-de-boucle
```

L'expression *expr1* est évaluée une seule fois, au début de l'exécution de la boucle.

L'expression *expr2* est évaluée et testée avant chaque passage dans la boucle.

L'expression *expr3* est évaluée après chaque passage.

Ces 3 expressions jouent respectivement le rôle :

- d'expression d'*initialisation*,
- de *test d'arrêt*,
- d'*incrémentation*.

Exemple

```
main()
{
  int   tab[] = {1, 2, 9, 10, 7, 8, 11};
  int   i, j;
  char  buffer[] = "Voici une chaîne"
                   " qui se termine "
                   "par un blanc ";
  char *p;
  int   t[4][3];

  for( i=0; i < sizeof tab / sizeof tab[0]; i++ )
    printf( "tab[%d] = %d\n", i, tab[i] );

  for( p=buffer; *p; p++ )
       ;
  *--p = '\0';
  printf( "buffer : %s$\n", buffer );
  for( i=0; i < 4; i++ )
    for( j=0; j < 3; j++ )
      t[i][j] = i + j;
}
```

6.2.4 - L'aiguillage

L'instruction **switch** définit un aiguillage qui permet d'effectuer un branchement à une *étiquette de cas* en fonction de la valeur d'une expression.

Syntaxe

```
switch (expression)
{
    case etiq1 :
    [ liste d'instructions ]
    case etiq2 :
    [ liste d'instructions ]
        ...
    case etiqn :
    [ liste d'instructions ]
[ default:
    [ liste d'instructions ]]
}
```

Les *étiquettes de cas* (*etiq1*, *etiq2*, ..., *etiqn*) doivent être des expressions constantes.

Une fois le branchement à l'étiquette de cas correspondante effectué, l'exécution se poursuit, par défaut, jusqu'à la fin du bloc *switch*. L'instruction d'*échappement* **break ;** permet de forcer la sortie du bloc.

L'expression indiquée au niveau du *switch* doit être de type *entier*.

Exemple

```
#include <stdio.h>
main()
{
  char *buffer = "\nCeci est une chaîne\n"
                 "de caractères\tsur\n\n"
                 "plusieurs     lignes.\n";
  int   NbCar  = 0, NbEsp = 0, NbLignes = 0;

  for( ; *buffer; buffer++, NbCar++ )
    switch( *buffer )
    {
      case '\n': NbLignes++;
                 break;
      case '\t':
      case ' ' : NbEsp++;
      default  : break;
    }
  printf( "NbCar = %d, NbEsp = %d, NbLignes = %d\n",
          NbCar, NbEsp, NbLignes );
}
```

6.3 – Instructions d'échappement

Ces instructions permettent de rompre le déroulement séquentiel d'une suite d'instructions.

Instruction continue ;

Le rôle de l'instruction **continue ;** est de forcer le passage à l'itération suivante de la boucle la plus proche.

Exemple

```
#include <stdio.h>
main()
{
  char *buffer = "\nCeci est une chaîne\n"
                 "de caractères\tsur\n\n"
                 "plusieurs     lignes.\n";
  int   NbCar  = 0, NbEsp = 0, NbLignes = 0;

  for( ; *buffer; buffer++ )
  {
    switch( *buffer )
    {
      case '\n': NbLignes++;
                 break;
      case '\t': continue;
      case ' ' : NbEsp++;
      default  : break;
    }
    NbCar++;
  }
  printf( "NbCar = %d, NbEsp = %d, NbLignes = %d\n",
          NbCar, NbEsp, NbLignes );
}
```

Instruction break ;

L'instruction **break ;** permet de quitter la boucle ou l'aiguillage le plus proche.

Exemple

```
#include <stdio.h>
main()
{
   char  buffer[] = "Wolfgang Amadeus Mozart\n"
                    " est un musicien divin.\n";
   char *p;

   for( p=buffer; *p; p++ )
     if( *p == '\n' )
     {
       *p = '\0';
       break;
     }
   printf( "Nom : %s\n", buffer );
}
```

Instruction return ;

Syntaxe

```
return [expression];
```

Cette instruction permet de sortir de la fonction qui la contient :

- si elle est suivie d'une *expression*, la valeur de celle-ci est transmise à la fonction appelante après avoir été convertie, si nécessaire et si possible, dans le type de celui de la fonction,
- sinon la valeur retournée est indéterminée.

Exemple

```
#include <stdio.h>
int main()
{
  char c;
  char majus( char c );
  void impression( char c );
     ...
  c = majus( c );
  impression( c );

  return 0;
}
char majus( char c )
{
  return c >= 'a' && c <= 'z' ?
         c + 'A' - 'a' : c;
}
void impression( char c )
{
  printf( "%c\n", c );

  return;
}
```

Instruction go to ;

Cette instruction sert à effectuer un transfert inconditionnel vers une autre partie du programme.

Syntaxe

```
goto étiquette;
```

Etiquette fait référence à une instruction étiquetée.

On utilisera cette instruction avec parcimonie car elle nuit à l'écriture de programme structuré.

Elle peut à la rigueur être utilisée lorsque l'on désire sortir de plusieurs boucles imbriquées, ce que ne permet pas l'instruction *break ;*.

Exemple

```
#include <stdio.h>
main()
{
  int tab[][4] = {1, 2, 8, 9, 10, 12, 1, 9, 5};
  int i, j;

  for( i=0; i < sizeof tab / sizeof tab[0]; i++ )
    for( j=0; j < 4; j++ )
      if( tab[i][j] == 10 )
        goto trouve;

  fprintf( stderr, "Elément non trouvé.\n" );
  return 1;

trouve:
  printf( "L'élément tab[%d][%d]"
          " est égal à 10.\n", i, j );

  return 0;
}
```

Un programme peut être interrompu au moyen de la fonction **exit**.

Syntaxe

```
exit(expression);
```

L'argument de cette fonction doit être un entier indiquant le code de terminaison du processus.

Exemple

```
#include <stdio.h>
int valeur = 10;
int tab[][4] = { 1, 2, 8, 9, 10, 12,
                 1, 9, 5, 7, 15, 16 };
int recherche( void )
{
  int i, j;

  for( i=0; i < sizeof tab / sizeof tab[0]; i++ )
    for( j=0; j < 4; j++ )
      if( tab[i][j] == valeur )
      {
        printf( "L'élément tab[%d][%d]"
                " est égal à %d.\n", i, j, valeur );
        return i;
      }
  fprintf( stderr, "Elément non trouvé.\n" );
  exit(1);
}
main()
{
  int ligne = recherche();
    ...
  return 0;
}
```

7 – Préprocesseur

7.1 – Introduction

Le **préprocesseur** effectue un prétraitement du programme source avant qu'il soit compilé.

Ce *préprocesseur* exécute des instructions particulières appelées **directives**.

Ces *directives* sont identifiées par le caractère # en tête. Elles peuvent se continuer sur plusieurs lignes, chaque ligne à continuer étant terminée par le caractère \ suivi d'un *return*.

7.2 – Pseudo-constantes

La directive **#define** permet la définition de **pseudo-constantes**.

Une *pseudo-constante* est un *identificateur* composé de lettres et de chiffres commençant par une lettre. (Le caractère _ est considéré comme une lettre).

Syntaxe

```
#define identificateur [chaîne-de-substitution]
```

Le préprocesseur remplace tous les mots du fichier source identiques à l'identificateur par la *chaîne-de-substitution*.

On préférera n'utiliser que des majuscules pour écrire ces identificateurs afin de les différencier des autres (variables, vecteurs, fonctions).

Exemple

```
#define  TAILLE 256
#define  TAILLE_EN_OCTETS \
          TAILLE*sizeof(int)
main()
{
  int  tab[TAILLE];
  int  i;

  for( i=0; i<TAILLE; i++ )
    tab[i] = i;
  printf( "Le tableau tab contient %d octets\n",
           TAILLE_EN_OCTETS );
}
```

Remarque

La directive **#undef** permet d'annuler la définition d'une *pseudo-constante*.

Pseudo-constantes prédéfinies

La plupart des préprocesseurs reconnaissent les *pseudo-constantes* prédéfinies suivantes :

- **__FILE__** : nom du fichier courant,
- **__LINE__** : numéro de la ligne courante,
- **__STDC__** : valeur non nulle si le compilateur est conforme à la norme ANSI,
- **__DATE__** : date du jour,
- **__TIME__** : heure.

7.3 – Pseudo-fonctions

Les **pseudo-fonctions** ou *macros* sont des substitutions paramétrables.

Exemple

```
#define ABS(x) x>0 ? x : -x
#define NB_ELEMENTS(t) sizeof t / sizeof t[0]
#include <stdio.h>
#include <math.h>
main()
{
  int    tab[][2] = { 1,  2,  3,  9,
                      10, 11, 13, 16};
  double r = -acos(-1.);
  int    i, j;

  for( i=0; i<NB_ELEMENTS(tab); i++ )
    for( j=0; j < 2; j++ )
      tab[i][j] = i + j;

  printf( "%f\n", ABS(r) );
}
```

Remarques

- Dans la définition d'une *pseudo-fonction*, on indiquera les arguments entre parenthèses pour éviter des erreurs lors de la substitution.
- On n'utilisera pas d'expression à effet de bord comme paramètre d'une *pseudo-fonction*.

Exemple

```
#define CARRE(x)   x*x
main()
{
  float x = 1.12;

  /*
   * Erreur : l'instruction suivante
   *          calcule 2*x+1 et non pas
   *          le carré de x+1.
   */

  printf( "%f\n", CARRE(x+1) );
}
```

Exemple

```
#define CARRE(x)   (x)*(x)
#define MAX(a,b)   ( (a) > (b) ? (a) : (b) )
main()
{
    float x = 3.1, y = 4.15;

    printf( "%f\n", CARRE(x+1) );
    printf( "%f\n", MAX(x+10., y) );

    /*
     *  Erreur : l'instruction suivante
     *           provoque une double
     *           incrémentation de x.
     */

    y = CARRE(x++);
}
```

7.4 – Inclusion de fichiers

La directive **#include** permet d'insérer le contenu d'un fichier dans un autre.

Ce mécanisme est en général réservé à l'inclusion de fichiers appelés *fichiers en-tête* contenant des déclarations de fonctions, de variables externes, de pseudo-constantes et pseudo-fonctions, de définition de types. Ces fichiers sont traditionnellement suffixés par **.h**.

Syntaxe

```
#include <nom-de-fichier>
#include "nom-de-fichier"
```

Si le nom du fichier est spécifié entre guillemets, il est recherché dans le répertoire courant. On peut indiquer d'autres répertoires de recherche au moyen de l'option **–I** de la commande **cc**.

Si le nom du fichier est spécifié entre **<>**, il est recherché par défaut dans le répertoire **/usr/include**.

Exemple

def.h

```
#define  NbElements(t)  sizeof t / sizeof t[0]
#define  TAILLE 256
typedef struct cellule
{
     int tab[TAILLE];
     struct cellule *ptr;
} Cel;
typedef enum bool {Faux, Vrai} logical;
extern void init(int t[], logical imp);
Cel c;
```

```c
#include "def.h"
main()
{
  int t[TAILLE] = {1, 2, 9, 10};
  logical imp = Vrai;

  init( t, imp );
}
```

```c
#include "def.h"
#include <stdio.h>
void init(int t[], logical imp)
{
  int i;

  for( i=0; i<NbElements(c.tab); i++ )
  {
    c.tab[i] = t[i];
    if( imp )
      printf( " %d", c.tab[i] );
  }
  printf( "%s", imp ? "\n" : "" );
  c.ptr = NULL;
}
```

Il existe une *bibliothèque standard* de fichiers en-tête nécessaires lors de l'appel de certaines fonctions :

- **stdio.h** (entrées-sorties),
- **string.h** (manipulations de chaînes de caractères),
- **ctype.h** (test du type d'un caractère : lettre, chiffre, séparateur, ...),
- **setjmp.h** (sauvegarde et restauration de contexte),
- **time.h** (manipulation de la date et de l'heure),
- **varargs.h** (fonction avec un nombre variable d'arguments),

- **stdarg.h** (fonction avec un nombre variable d'arguments),
- **errno.h** (codification des erreurs lors d'appels système),
- **signal.h** (manipulation des signaux inter-processus),
- **math.h** (manipulation de fonctions mathématiques),
- **fcntl.h** (définitions concernant les entrées-sorties).

7.5 – Compilation conditionnelle

Test d'existence d'une pseudo-constante

Ce sont les directives **#ifdef** et **#ifndef** qui permettent de tester l'existence d'une pseudo-constante.

Syntaxe

```
#ifdef identificateur
    partie-alors
[#else
    partie-sinon]
#endif

#ifndef identificateur
    partie-alors
[#else
    partie-sinon]
#endif
```

Exemple

```
#ifdef TAILLE_BUF
# undef TAILLE_BUF
#endif /* TAILLE_BUF */
#define TAILLE_BUF 4096
```

def.h **source.c**

```
#ifdef DEBUG                 |  #define DEBUG
  #define trace(s) \         |  #include "def.h"
          printf s           |  #include <stdio.h>
#else                        |  main()
#define trace(s)             | {
#endif/* DEBUG */            |      int f(float x);
                             |      int i;
                             |      float r;
                             |
                             |      i = f(r);
                             |      trace(("%d\n", i));
                             |  }
```

La définition d'une *pseudo-constante* ainsi que sa valorisation peuvent se faire à l'appel de la commande **cc** au moyen de l'option **-D**.

Syntaxe

```
cc -Dpseudo-constante[=valeur] ...
```

On peut appliquer ce principe à la *pseudo-constante* `DEBUG` de l'exemple précédent au lieu de la définir dans le fichier *source.c* :

```
cc -DDEBUG source.c
```

Évaluation de pseudo-expressions

Il est possible de construire des expressions interprétables par le préprocesseur à l'aide :

- de constantes entières ou caractères,
- de parenthèses,
- des opérateurs unaires `-`, `!` `~`,
- des opérateurs binaires `+`, `-`, `*`, `/`, `%`, `&`, `|`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&&` et `||`,
- de l'opérateur conditionnel `? :`,
- de l'opérateur unaire **defined** qui s'applique à une pseudo-constante.

Syntaxe

```
#if pseudo-expression
    partie-alors
[#else
    partie-sinon]
#endif
```

Remarque

Si l'on désire mettre en commentaire une portion de programme, la solution consistant à l'encadrer par les caractères `/*` et `*/` ne marche pas si elle contient elle-même des commentaires.

Une solution simple est de placer en tête de la région à commenter la directive **`#if 0`**, et à la fin la directive **`#endif`** `/*0*/`.

Exemple

source.c

```
#define TBLOC 256
#if !defined  TAILLE
#  define TAILLE TBLOC
#endif
#if TAILLE%TBLOC == 0
#  define TAILLEMAX TAILLE
#else
#  define TAILLEMAX ((TAILLE/TBLOC+1)*TBLOC)
#endif
static char buffer[TAILLEMAX];
main()
{
   printf( "Taille du vecteur : %d caractères\n",
            sizeof buffer );
}
```

❶ `cc -DTAILLE=255 source.c`

❷ `cc -DTAILLE=257 source.c`

❸ `cc source.c`

8 – Les fonctions

8.1 – Passage arguments-paramètres

Dans les langages de programmation il existe deux techniques de passage d'arguments :

- ☞ par **adresse**,
- ☞ par **valeur**.

Des langages comme **Fortran** ou **PL/1** ont choisi la 1^re^solution, tandis qu'un langage comme **Pascal** offre les deux possibilités au programmeur.

Le langage **C** a choisi la 2^e^solution.

Si un argument doit être passé par *adresse*, c'est le programmeur qui en prend l'initiative et ceci grâce à l'opérateur d'adressage (&).

Exemple

```
#include <stdio.h>
void main()
{
   int a, b, c;
   void somme(int a, int b, int *c);

   a = 3;
   b = 8;
   somme(a, b, &c);
   printf("Somme de a et b : %d\n", c);

   return;
}

void somme(int a, int b, int *c)
{
   *c = a + b;

   return;
}
```

8.2 – Fonction retournant un pointeur

Il convient d'être prudent lors de l'utilisation d'une fonction retournant un pointeur.

Il faudra éviter l'erreur qui consiste à retourner l'adresse d'une variable *temporaire*.

Exemple

```
#include <stdio.h>
void main()
{
   char *p;
   char *ini_car(void);
   p = ini_car();
   printf("%c\n", *p);
}
char *ini_car(void)
{
   char c;
   c = '#';
   return(&c);      <===  ERREUR
}
```

8.3 – Passage d'un vecteur comme argument

Un vecteur est une constante symbolique dont la valeur est l'adresse de son 1erélément.

Lorsqu'un vecteur est passé en argument, c'est donc l'adresse de son 1erélément qui est transmise par valeur.

Exemple

```
#define NbElements(t) sizeof t / sizeof t[0]
#include <stdio.h>
main()
{
   int  tab[] = { 1, 9, 10, 14, 18};
   int  somme(int t[], int n);
   void impression(int *t, int n);
   printf("%d\n", somme(tab, NbElements(tab)));
   impression(tab, NbElements(tab));
}
int  somme(int t[], int n)
{
   int i, som=0;
   for (i=0; i < n; i++) som += t[i];
   return som;
}
void impression(int *t, int n)
{
   int i=0, *p;
   for (p=t; t-p < n; t++)
      printf("t[%d] = %d\n", i++, *t);
}
```

Exemple

```
#define NbElements(t) sizeof t / sizeof t[0]
#include <stdio.h>
main()
{
   int tab[][5] = {
                     { 4, 7, 1, 9, 6},
                     { 5, 9, 3, 4, 2},
                     { 2, 9, 5, 9, 13}
                   };
   int somme(int (*t)[5], int n);

   printf("Somme des éléments de tab : %d\n",
           somme(tab, NbElements(tab)));
}
int somme(int (*t)[5], int n)
{
   int i, som = 0;
   int (*p)[5] = t;
   for(; t-p < n; t++)
     for (i=0; i < 5; i++)
        som += (*t)[i];
   return som;
}
```

Exemple

```
#define DIM1 10
#define DIM2  4
#define DIM3  5
main()
{
   int tab[DIM1][DIM2][DIM3];
   void init(int (*t)[DIM3], int n);
   int i, n = DIM2;

   for(i=0; i < DIM1; i++)
     init(tab[i], i);
}
void init(int (*t)[DIM3], int n)
{
   int i, j;

   for(i=0; i < DIM2; i++)
     for(j=0; j < DIM3; j++) {
       t[i][j]      = 2*(i+n*DIM2);
       *(*(t+i)+j) += 1;
     }
}
```

8.4 – Passage d'une structure comme argument

La norme ANSI a introduit la possibilité de transmettre une *structure* en argument d'une fonction, elle-même pouvant retourner un tel objet.

Exemple

```
#include <stdio.h>
#define NbElts(v) ( sizeof v / sizeof v[0] )

typedef struct
{
   float v[6];
} Vecteur;
```

```
main()
{
  Vecteur vec = { {1.34f, 8.78f, 10.f,
                   4.f, 22.12f, 3.145f} };
  Vecteur inv;
  Vecteur inverse( Vecteur vecteur, int n );
  int     i, n = NbElts(vec.v);

  inv = inverse( vec, n );
  for( i=0; i < n; i++ )
    printf( "inv.v[%d] : %f\n", i, inv.v[i] );
}

Vecteur inverse( Vecteur vecteur, int n )
{
  Vecteur w;
  int     i;

  for( i=0; i < n; i++ )
    w.v[i] = vecteur.v[i] ? 1./vecteur.v[i] : 0.f;

  return w;
}
```

8.5 – Passage d'une fonction comme argument

Le nom d'une fonction est une constante symbolique dont la valeur est un pointeur sur la 1re instruction exécutable du code machine de la fonction.

Passer une fonction en argument, c'est donc transmettre l'adresse, par valeur, du début du code machine constituant cette fonction.

Exemple

```
double integrale(double b_inf, double b_sup,
                 int pas, double (*f)(double));
double carre(double x);
int main()
{
   double b_inf, b_sup, aire;
   int    pas;
   b_inf = 1., b_sup = 6., pas = 2000;
   aire = integrale(b_inf, b_sup, pas, carre);
   printf("Aire : %f\n", aire);

   return 0;
}
double integrale(double b_inf, double b_sup,
                 int pas, double (*f)(double))
{
   double surface = 0., h;
   int    i;
   h = (b_sup - b_inf)/pas;
   for(i=0; i < pas; i++)
     surface += h*(*f)(b_inf+i*h);

   return surface;
}
double carre(double x) {return x*x;}
```

8.6 – Passage d'arguments à la fonction main

Lorsqu'un exécutable est lancé sous un interprète de commandes (*shell*), un processus est créé et son exécution commence par la fonction *main* à laquelle des arguments sont transmis après avoir été générés par le *shell*.

Ces arguments sont constitués de :

- ceux fournis au lancement de l'exécutable,
- leur nombre (y compris l'exécutable),
- l'environnement du *shell*.

Les premier et dernier sont transmis sous forme de *vecteurs de pointeurs de caractères*.

Par convention :

- **argc** désigne le nombre d'arguments transmis au moment du lancement de l'exécutable,
- **argv** désigne le vecteur contenant les différents arguments,
- **envp** désigne le vecteur contenant les informations sur l'environnement.

Les arguments précédents sont transmis à la fonction *main* dans cet ordre.

Exemple

La commande **a.out toto titi tata** génère la structure de données suivante :

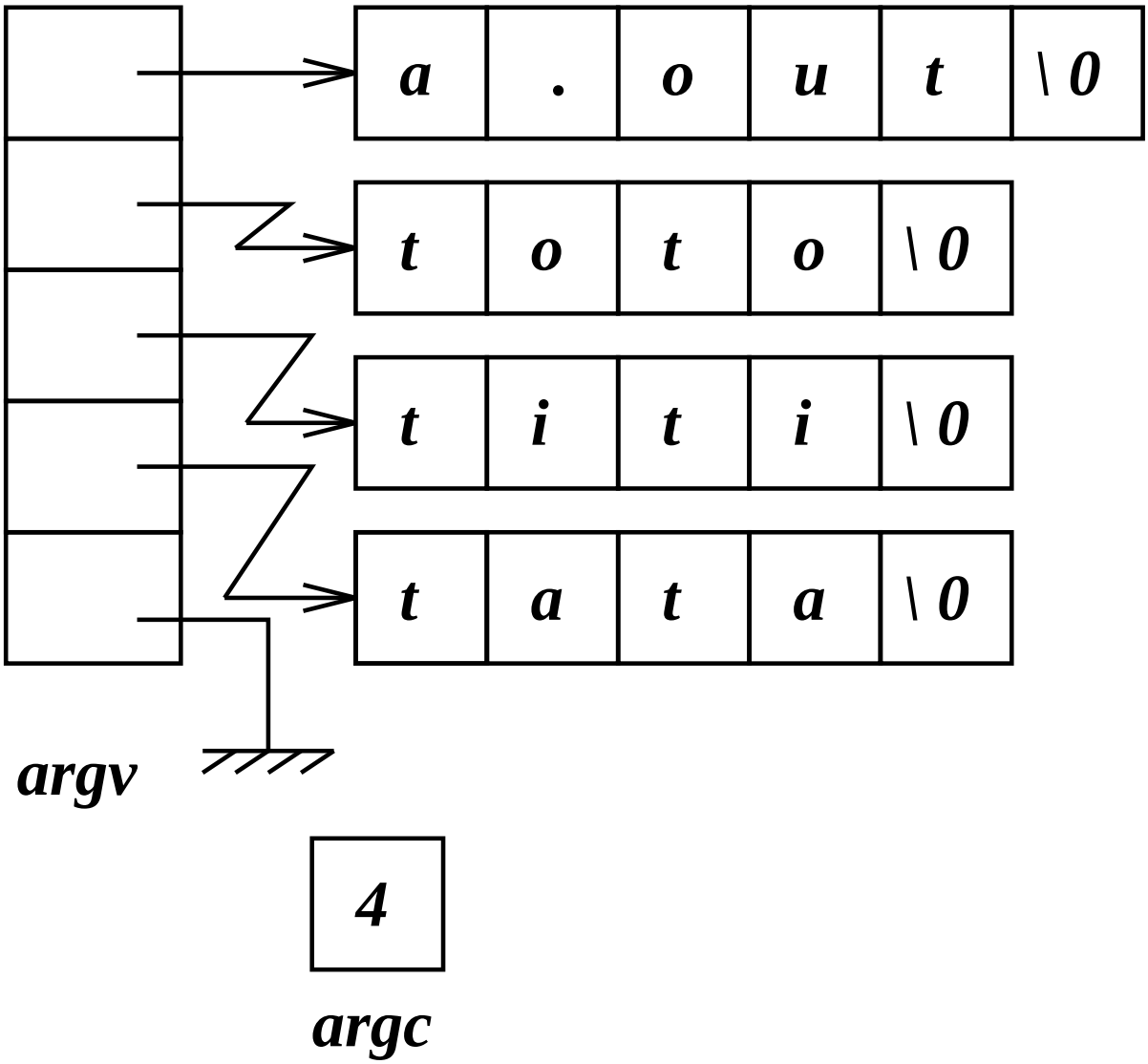

FIG. 2 – Tableau des arguments

Exemple

```
#include <stdio.h>
main( int argc, char **argv, char **envp )
{
  void usage(char *s);

  if( argc != 3 )
    usage( argv[0] );
  for( ; *argv; argv++ )
    printf( "%s\n", *argv );
  for( ; *envp; envp++ )
    printf( "%s\n", *envp );
}

void usage( char *s )
{
  fprintf( stderr, "usage : %s arg1 arg2\n", s );
  exit(1);
}
```

8.7 – Fonction avec un nombre variable d'arguments

Lors de l'appel à une fonction, le compilateur génère une liste des arguments fournis qu'il empile dans la **pile** d'exécution rattachée au processus (pile de type **LIFO**).

Exemple

```
int puissance(int n, int x)
{
  int p = 1;
  while(n--) p *= x;
  return p;
}
void main()
{
  int m, k, r;
  k = 4; m = 2;
  r = puissance(k+3, m);
}
```

A l'appel de la fonction *puissance* de l'exemple précédent il y a :

- allocation dans la *pile* d'autant de variables consécutives qu'il y a d'arguments spécifiés, **(1)**
- copie dans ces variables des valeurs des arguments, **(2)**
- mise en relation des arguments d'appel avec ceux indiqués lors de la définition de la fonction appelée (fonction *puissance*). **(3)**

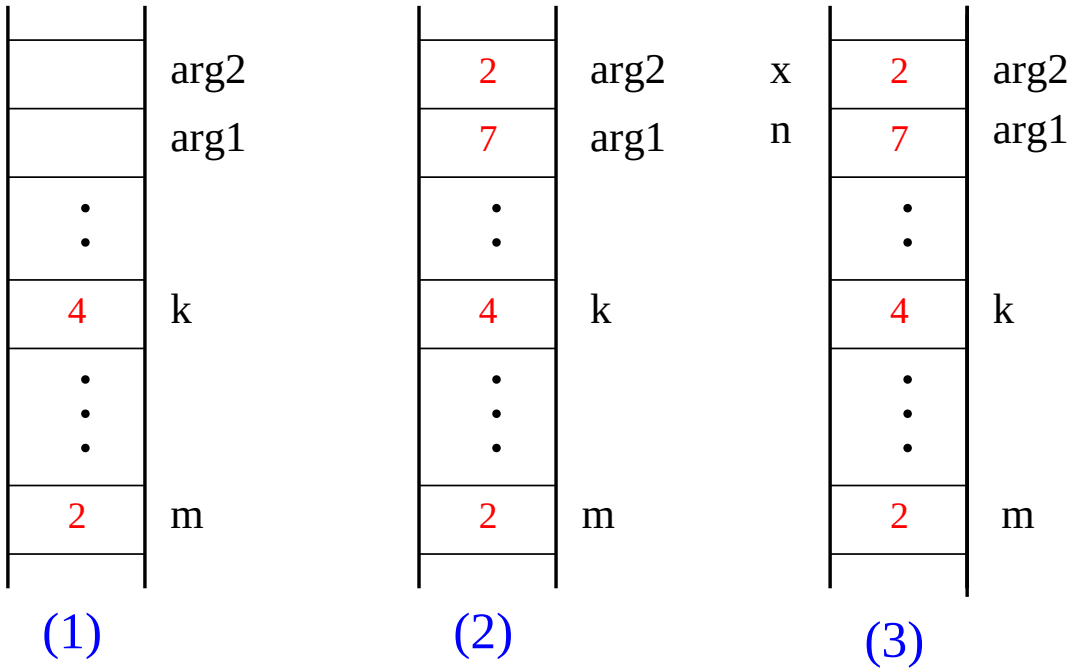

FIG. 3 – Processus de passage d'arguments

Ce type de passage d'arguments permet d'écrire des fonctions avec un nombre *variable* d'arguments.

Dans le *prototype* d'une telle fonction, on indiquera les arguments suivis de :

- ☞ **...** pour les compilateurs `ANSI`,
- ☞ **va_alist** pour les compilateurs avant norme.

Comme les arguments sont rangés de façon consécutive dans la *pile*, le programmeur a la possibilité d'aller chercher les arguments en surnombre.

Exemple

```
void fonction(int a, ...);
main()
{
   int i = 10, j = 11, k = 12;
   printf("Avant appel fonction i = %d\n", i);
   printf("Avant appel fonction j = %d\n", j);
   printf("Avant appel fonction k = %d\n", k);
   fonction(i, j, k);
}
void fonction(int a, ...)
{
   printf("Valeur de a = %d\n", a);
   printf("Récupération de j = %d\n", *(&a + 1));
   printf("Récupération de k = %d\n", *(&a + 2));
}
```

Cette technique de récupération d'arguments dans la *pile*, nécessite cependant que le programmeur connaisse les types des arguments en surnombre et qu'il ait un moyen d'arrêter la recherche.

Pour ce faire, le dernier argument fourni à l'appel de la fonction peut par exemple indiquer le nombre d'arguments en surnombre, ou bien ceux-ci peuvent être suivis par un argument supplémentaire de même type avec une valeur spéciale (*valeur sentinelle*).

Par contre la norme ne précise pas l'ordre dans lequel les arguments doivent être empilés, cette méthode de la *pile* n'est donc pas portable.

Pour assurer cette portabilité, chaque système propose des *pseudo-constantes* et *pseudo-fonctions* permettant au programmeur de gérer cette recherche.

Version Unix System V

Les *pseudo-constantes* et *pseudo-fonctions* sont stockées dans le fichier en-tête **varargs.h** :

- **va_alist** symbolise les arguments en surnombre dans le prototype,
- **va_list** permet de déclarer le pointeur dans la pile,
- **va_dcl** permet de déclarer le 1erargument en surnombre,
- **va_start** permet d'initialiser le pointeur de la pile sur le début de la liste des arguments en surnombre,
- **va_arg** récupère les valeurs des arguments en surnombre,
- **va_end** appelée lorsque la recherche est terminée.

Exemple

```
#include <stdio.h>
#include <varargs.h>
main()
{
   float moyenne();
   printf("moyenne = %f\n", moyenne(4, 1, 2, 3, 4));
   printf("moyenne = %f\n",
           moyenne(5, 1, 2, 3, 4, 5));
}
float moyenne(nombre, va_alist)
int nombre;
va_dcl
{
   int somme = 0, i;
   va_list arg;
   va_start(arg);
   for(i=0; i < nombre; i++)
     somme += va_arg(arg, int);
   va_end(arg);
   return somme/nombre;
}
```

Exemple

```
#include <stdio.h>
#include <varargs.h>
main()
{
   float moyenne();
   printf("moyenne = %f\n",
           moyenne(4, 1.f, 2.f, 3.f, 4.f));
   printf("moyenne = %f\n",
           moyenne(5, 1.f, 2.f, 3.f, 4.f, 5.f));
}
float moyenne(nombre, va_alist)
int nombre;
va_dcl
{
   float   somme = 0.f;
   int     i;
   va_list arg;
   va_start(arg);
   for(i=0; i < nombre; i++)
     somme += va_arg(arg, double);  ===> surtout
   va_end(arg);                              pas float!
   return somme/nombre;
}
```

Version ANSI

Les *pseudo-constantes* et *pseudo-fonctions* sont stockées dans le fichier en-tête **stdarg.h** :

- **va_list** permet de déclarer le pointeur dans la pile,
- **va_start** permet d'initialiser le pointeur de la pile sur le début de la liste des arguments en surnombre,
- **va_arg** récupère les valeurs des arguments en surnombre,
- **va_end** appelée lorsque la recherche est terminée.

Les arguments en surnombre sont symbolisés par **...** dans le *prototype* de la fonction.

Exemple

```
#include <stdio.h>
#include <stdarg.h>
main()
{
   float moyenne(int nombre, ...);
   printf("moyenne = %f\n", moyenne(4, 1, 2, 3, 4));
   printf("moyenne = %f\n",
           moyenne(5, 1, 2, 3, 4, 5));
}
float moyenne(int nombre, ...)
{
   int somme = 0, i;
   va_list arg;
   va_start(arg, nombre);
   for(i=0; i < nombre; i++)
     somme += va_arg(arg, int);
   va_end(arg);
   return somme/nombre;
}
```

9 – La bibliothèque standard

9.1 – Notion de pointeur générique

La norme a défini le type **void** $*$ ou *pointeur générique* afin de faciliter la manipulation des pointeurs et des objets pointés indépendamment de leur type.

On ne pourra pas appliquer les opérateurs d'indirection et d'auto-incrémentation, auto-décrémentation à un *pointeur générique*.

Par contre, si p et q sont deux pointeurs, les affectations :

☞ p = q ;

☞ q = p ;

sont toutes deux correctes si l'un au moins des deux pointeurs p ou q est de type *void* $*$, quel que soit le type de l'autre pointeur.

Exemples

```
int x[5], i, *k;
float *r;
void  *p;
void  *q;

p = &x[0];        /* correct  */
*p = ...          /* interdit */
q = p + 1;        /* interdit */
r = p;            /* correct  */
p = r;            /* correct  */
p[1] = ...;       /* interdit */
```

Exemples

```
void echange (void *p, void *q)
{
  void *r;

   r          = *(void **)p;
  *(void **)p = *(void **)q;
  *(void **)q =  r;
}
main()
{
  int    *i1, *i2;
  float  *f1, *f2;
  double *d1, *d2;
      ...
  echange(&i1, &i2);
  echange(&f1, &f2);
  echange(&d1, &d2);
}
```

9.2 – Entrées-sorties de haut niveau

Les *entrées-sorties de haut niveau* intègrent deux mécanismes distincts :

- le *formatage* des données,
- la mémorisation des données dans une mémoire *tampon*.

Toute opération d'entrée-sortie se fera par l'intermédiaire d'un **flot** (*stream*) qui est une structure de données faisant référence à :

- la nature de l'entrée-sortie,
- la mémoire tampon,
- le fichier sur lequel elle porte,
- la position courante dans le fichier, ...

Cette structure de données est un objet de type **FILE**. Dans le programme, un *flot* sera déclaré de type **FILE** $*$.

Trois *flots* sont prédéfinis au lancement d'un processus :

- **stdin** initialisé en lecture sur l'entrée standard,
- **stdout** initialisé en écriture sur la sortie standard,
- **stderr** initialisé en écriture sur la sortie erreur standard.

Les informations précédentes sont contenues dans le fichier en-tête **stdio.h**.

Ce fichier contient, de plus, les déclarations des différentes fonctions d'entrée-sortie, ainsi que la déclaration d'un vecteur (**_iob**) de type *FILE* dont la dimension est définie à l'aide d'une *pseudo-constante*.

Extrait du fichier *stdio.h* sur *IBM/RS6000*

```
#define _NIOBRW          20
extern FILE     _iob[_NIOBRW];

#define stdin            (&_iob[0])
#define stdout           (&_iob[1])
#define stderr           (&_iob[2])
```

9.2.1 - Fonctions d'ouverture et de fermeture

L'acquisition d'un nouveau *flot* s'effectue par l'appel à la fonction **fopen**. La fonction **fclose** permet de le fermer.

Syntaxe

```
FILE *fopen(const char *file, const char *type);
int   fclose(const FILE *flot);
```

La fonction *fopen* retourne un pointeur sur le 1er élément libre du vecteur *_iob* s'il en existe, sinon sur une zone de type *FILE* allouée dynamiquement.

Un pointeur *NULL*, pseudo-constante définie comme **(void *)0** dans *stdio.h*, indique une fin anormale.

La fonction *fclose* retourne **0** en cas de succès, **-1** sinon.

Le 2e argument de la fonction *fopen* indique le mode d'ouverture du fichier.

Accès	**Paramètre**	**Position**	**Comportement**	
			si le fichier existe	si le fichier n'existe pas
lecture	**r**	début		erreur
écriture	**w**	début	mis à zéro	création
	a	fin		création
lecture et écriture	**r+**	début		erreur
	w+	début	mis à zéro	création
	a+	fin		création

Certains systèmes font la distinction entre les fichiers *texte* et *binaire*. Pour manipuler ces derniers, il suffit de rajouter le caractère **b** dans la chaîne indiquant le mode d'ouverture. Sous `UNIX`, il est ignoré car il n'existe aucune différence entre un fichier *binaire* et un fichier de données quelconques.

Exemple

```
#include <stdio.h>
main()
{
  FILE * flot;

  if( (flot = fopen( "donnees", "r" )) == NULL )
  {
    fprintf( stderr, "Erreur à l'ouverture\n" );
    exit(1);
  }
      ...
      ...
  fclose( flot );
}
```

9.2.2 - Lecture et écriture par caractère

Les fonctions **getc, fgetc** et **putc, fputc** permettent de lire ou écrire un caractère sur un *flot* donné.

getc et *putc* sont des pseudo-fonctions.

Syntaxe

```
int getc(FILE *Stream)
int fgetc(FILE *Stream)
int putc(int c, FILE *Stream)
int fputc(int c, FILE *Stream)
```

Il existe deux pseudo-fonctions supplémentaires :

- **getchar()** identique à **getc(stdin)**,
- **putchar(c)** identique à **putc(c, stdout)**.

Ces fonctions retournent soit le caractère traité, soit la pseudo-constante **EOF**, définie comme **-1** dans le fichier *stdio.h*, en cas d'erreur (fin de fichier par exemple).

Deux *pseudo-fonctions* **feof** et **ferror**, définies dans le fichier *stdio.h*, permettent de tester, respectivement, la *fin de fichier* et une éventuelle erreur d'entrée-sortie sur le *flot* passé en argument.

Dans le cas d'une entrée-sortie au terminal, c'est le *retour chariot* qui provoque l'envoi au programme de la mémoire tampon rattachée au pilote **/dev/tty**.

Exemple

```
#include <stdio.h>
main()
{
  char c;  <=== Attention Erreur !
                ----------------

  while( (c = getchar()) != EOF )
     putchar(c);
}
```

Exemples corrects

```
#include <stdio.h>
main()
{
  int c;

  while( (c = getchar()) != EOF )
     putchar(c);
}
```

```
#include <stdio.h>
main()
{
  int c;

  c = getchar();
  while( ! ferror(stdin) &&
         ! feof(stdin) )
  {
     putchar(c);
     c = getchar();
  }
}
```

9.2.3 - Lecture et écriture de mots

Les fonctions **getw** et **putw** permettent de lire ou écrire des mots.

Syntaxe

```
int getw(FILE *flot)
int putw(int c, FILE *flot)
```

Exemple

```
#include <stdio.h>
#define DIM 100
main()
{
   FILE *flot;
   int   tab[DIM];
   int   i;

   if( (flot = fopen( "resultat", "w" )) == NULL )
   {
     perror("fopen");
     exit(1);
   }
   for( i=0; i < DIM; i++ )
   {
     tab[i] = i*i;
     putw( tab[i], flot );
   }
   fclose( flot );
}
```

9.2.4 - Lecture et écriture d'une chaîne de caractères

Les fonctions **gets, fgets** et **puts, fputs** permettent de lire et écrire des chaînes de caractères.

Syntaxe

```
char *gets(char *string)
int   puts(char *string)
char *fgets(char *string, int nombre, FILE *flot)
int   fputs(char *string, FILE *flot)
```

☞ `gets` lit sur le flot `stdin` jusqu'à la présence du caractère *retour chariot* et range le résultat dans la chaîne passée en argument. Le *retour chariot* est remplacé par le caractère `\0` de fin de chaîne.

- `puts` écrit sur le flot `stdout` la chaîne passée en argument suivie d'un *retour chariot*.
- `fgets` lit sur le *flot* fourni en 3^{e}argument jusqu'à ce que l'un des évènements suivants se produise :
  - « *nombre-1* » octets ont été lus,
  - rencontre d'un *retour chariot*,
  - `fin de fichier` atteinte.

  Le caractère `\0` est ensuite ajouté en fin de chaîne. Dans le deuxième cas le *retour chariot* est stocké dans la chaîne.
- `fputs` écrit la chaîne fournie en 1erargument sur le flot spécifié en 2^{e}argument. Cette fonction n'ajoute pas de *retour chariot*.

Les fonctions `gets, fgets` renvoient la chaîne lue ou le pointeur `NULL` si fin de fichier. Les fonctions `puts, fputs` renvoient le nombre de caractères écrits ou `EOF` si erreur.

Exemple

```
#include <stdio.h>
main()
{
  char *mus1 = "Wolfgang Amadeus Mozart\n";
  char *mus2 = "Ludwig van Beethoven\n";
  char  buffer[BUFSIZ+1];
  FILE *f;
  if( (f = fopen( "musiciens", "w" )) == NULL )
  {
    perror( "fopen" );
    exit(1);
  }
  fputs( mus1, f ); fputs( mus2, f );
  fclose(f);
  if( (f = fopen( "musiciens", "r" )) == NULL )
  {
    perror( "fopen" );
    exit(2);
  }
  while( fgets( buffer, sizeof(buffer), f ) )
     fputs( buffer, stdout );
  fclose(f);
  puts( "\nExecution terminée." );
}
```

9.2.5 - Lecture et écriture de blocs

Les fonctions **fread** et **fwrite** permettent de lire et d'écrire des blocs de données tels des *structures* ou des *tableaux*.

Syntaxe

```
size_t fread(void *p, size_t t, size_t n, FILE *f)
size_t fwrite(void *p, size_t t, size_t n, FILE *f)
```

- **p** désigne la mémoire tampon réceptrice ou émettrice,
- **t** indique la taille du bloc à lire ou écrire,
- **n** indique le nombre de blocs à lire ou écrire,
- **f** désigne le flot.

Ces fonctions retournent le nombre de blocs traités. Utiliser les *pseudo-fonctions* `feof` et `ferror` pour tester la fin de fichier et une erreur d'entrée-sortie.

Exemple

```
#include <stdio.h>
#define NbElt(t) ( sizeof t / sizeof t[0] )
main() {
  typedef struct { int   n; float t[10]; char  c;
                 } Donnee;
  Donnee s1   = { 1, { 1.,  2., 3.}, 'a'};
  Donnee s2[] = { {4, {10., 32., 3.}, 'z'},
                  {5, { 2., 11., 2., 4.}, 'h'} };
  FILE *f, *f_sauve; Donnee s;
  if( (f = fopen( "donnee", "w" )) == NULL )
    perror("fopen"), exit(1);
  fwrite( &s1, sizeof(Donnee), 1, f );
  fwrite(  s2, sizeof(Donnee), NbElt(s2), f );
  fclose(f);
  if( (f       = fopen( "donnee", "r" ))     == NULL ||
      (f_sauve = fopen( "sauvegarde", "w" )) == NULL )
    perror("fopen"), exit(2);
  fread( &s, sizeof(Donnee), 1, f );
  while( ! feof(f) )
  {
    fwrite( &s, sizeof(Donnee), 1, f_sauve );
    fread( &s, sizeof(Donnee), 1, f );
  }
  fclose(f); fclose(f_sauve);
}
```

9.2.6 - Accès direct

Par défaut, les fonctions précédentes travaillent en *mode séquentiel*. Chaque lecture ou écriture s'effectue à partir d'une position courante, et incrémente cette position du nombre de caractères lus ou écrits.

Les fonctions **fseek** et **ftell** permettent, respectivement, de modifier et récupérer la position courante.

```
int  fseek(FILE *f, long decalage, int position);
long ftell(FILE *f);
```

La fonction *ftell* retourne la position courante en octets.

La fonction *fseek* permet de la modifier :

- la valeur du *décalage* est exprimée en octets,
- la position est celle à partir de laquelle est calculé le décalage. Elle s'exprime à l'aide de 3 pseudo-constantes définies dans le fichier *stdio.h* :
  - `SEEK_SET` (0 : début de fichier),
  - `SEEK_CUR` (1 : position courante),
  - `SEEK_END` (2 : fin de fichier).

Exemple

```
#include <stdio.h>
main( int argc, char **argv )
{
  FILE *f;
  void  usage( char *s );

  if( argc != 2 )
    usage( argv[0] );
  if( (f = fopen( argv[1], "r" )) == NULL )
  {
    perror( "fopen" );
    exit(2);
  }
  fseek( f, 0L, SEEK_END );
  printf( "Taille(octets) : %d\n", ftell(f) );
  fclose(f);
}

void usage(char *s)
{
  fprintf( stderr, "usage : %s fichier\n", s );
  exit(1);
}
```

Exemple

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>

main( int argc, char **argv )
{
  void  conversion( char *nom, char *mode );
  FILE *f;
  char  nom[100];
  char *mus[] = { "Frédéric Chopin\n",
                  "Maurice Ravel\n", };
  int   n;

  if ( argc != 2 ) exit( 1 );
  if ( ( f = fopen( argv[1], "r+" ) ) == NULL )
  {
    if ( errno != ENOENT ||
       ( f = fopen( argv[1], "w+" ) ) == NULL )
    {
      perror( "fopen" );
      exit( 2 );
    }
  }
```

```
  fgets( nom, 100, f );
  while ( !feof( f ) )
  {
    fseek( f, -1L*strlen(nom), SEEK_CUR );
    conversion( nom, "majuscule" );
    fputs( nom, f );
    fseek( f, 0L, SEEK_CUR );
    fgets( nom, 100, f );
  }
  for( n=0; n < sizeof mus / sizeof mus[0]; n++ )
    fputs( mus[n], f );
  fclose( f );
}
void conversion( char *nom, char *mode )
{
  char  *ptr;
  int  (*conv)(int);

  if ( !strcmp( mode, "majuscule" ) )
    conv = toupper;
  else if ( !strcmp( mode, "minuscule" ) )
    conv = tolower;
  for( ptr=nom; *ptr++=conv(*ptr); )
     ;
  return;
}
```

9.2.7 - Entrées-sorties formatées

Les fonctions **scanf, fscanf, sscanf** et **printf, fprintf, sprintf** permettent d'effectuer des entrées-sorties de données avec *conversions*.

Syntaxe

```
int scanf(const char *format [, ...])
int fscanf(FILE *f, const char *format [, ...])
int sscanf(const char *buffer,
           const char *format [, ...])
int printf(const char *format [, ...])
int fprintf(FILE *f, const char *format [, ...])
int sprintf(char *buffer,
            const char *format [, ...])
```

Fonctions printf, fprintf, sprintf

Le paramètre *format* désigne une chaîne de caractères comprenant :

- des caractères ordinaires,
- des **spécifications de conversions** : suite de caractères précédée du symbole %.

Après les conversions effectuées, cette chaîne est reproduite :

- sur le `flot stdout` si *printf*,
- sur le `flot` indiqué si *fprintf*,
- dans la mémoire tampon spécifiée si *sprintf*.

Les *spécifications de conversions* portent successivement sur les arguments passés à la suite du paramètre *format*.

Une *spécification de conversion* est constituée du caractère **%**, suivie dans l'ordre :

- du caractère **-** qui cadre l'argument converti à gauche dans la zone réceptrice (optionnel),
- d'un caractère de gestion du *signe* pour les données numériques (optionnel) :
  - **+** pour forcer la présence du signe,
  - **espace** pour placer un *espace* à la place du signe lorsque la donnée est positive.
- du caractère **0** qui, pour les données numériques, remplit, la zone réceptrice, à gauche par des **0** (optionnel),

- du caractère **#** (optionnel) qui permet :
  - de préfixer par un **0** les nombres entiers écrits en octal,
  - de préfixer par **0x** les nombres entiers écrits en hexadécimal,
  - de forcer la présence du point décimal pour les réels.
- de la taille minimum de la zone réceptrice (optionnelle),
- de la précision numérique précédée d'un **.** (optionnelle) :
  - le nombre de chiffres à droite du point décimal pour un réel (6 par défaut),
  - le nombre maximum de caractères d'une chaîne que l'on veut stocker dans la zone réceptrice,
  - le nombre minimum de chiffres d'un entier que l'on désire voir apparaître dans la zone réceptrice.

☞ du type de la donnée à écrire, défini par l'un des caractères suivants (obligatoire) :

- **c** pour un caractère,
- **u** pour un entier non signé,
- **d** pour un entier signé sous forme décimale,
- **o** pour un entier signé sous forme octale,
- **x** pour un entier signé sous forme hexadécimale,
- **f** pour un réel sous forme décimale,
- **e** pour un réel sous forme exponentielle,
- **g** pour un réel sous forme générale :
  - équivalent à **e** si l'exposant est inférieur à **-4** ou supérieur ou égal à la précision,
  - équivalent à **f** sinon.

  Dans ce cas, la précision indique le nombre maximun de chiffres significatifs.
- **s** pour une chaîne de caractères.

Exemples

```
printf("|%d|\n", 1234);                |1234|
printf("|%-d|\n", 1234);               |1234|
printf("|%+d|\n", 1234);               |+1234|
printf("|% d|\n", 1234);               | 1234|
printf("|%10d|\n", 1234);             |      1234|
printf("|%10.6d|\n", 1234);           |    001234|
printf("|%10.2d|\n", 1234);           |      1234|
printf("|%.6d|\n", 1234);             |001234|
printf("|%06d|\n", 1234);             |001234|
printf("|%.2d|\n", 1234);             |1234|
printf("|%*.6d|\n", 10, 1234);        |    001234|
printf("|%*.*d|\n", 10, 6, 1234);     |    001234|
printf("|%x|\n", 0x56ab);             |56ab|
printf("|%#x|\n", 0x56ab);            |0x56ab|
printf("|%X|\n", 0x56ab);             |56AB|
printf("|%#X|\n", 0x56ab);            |0X56AB|
```

```
printf("|%f|\n",1.23456789012345678e5);
                |123456.789012|
printf("|%.4f|\n",1.23456789012345678e5);
                |123456.7890|
printf("|%.15f|\n",1.23456789012345678e5);
                |123456.789012345670000|
printf("|%15.4f|\n",1.23456789012345678e5);
                |    123456.7890|

printf("|%e|\n",1.23456789012345678e5);
                |1.234568e+05|
printf("|%.4e|\n",1.23456789012345678e5);
                |1.2346e+05|
printf("|%.18e|\n",1.23456789012345678e5);
                |1.234567890123456700e+05|
printf("|%18.4e|\n",1.23456789012345678e5);
                |         1.2346e+05|
printf("|%.4g|\n",1.23456789012345678e-5);
                |1.235e-05|
printf("|%.4g|\n",1.23456789012345678e+5);
                |1.235e+05|
printf("|%.4g|\n",1.23456789012345678e-3);
                |0.001235|
printf("|%.8g|\n",1.23456789012345678e5);
                |123456.79|
```

```
#include <stdio.h>
main()
{
  char *chaine = "Wolfgang Amadeus Mozart";

  printf("|%s|\n", chaine);        ===> 1)
  printf("|%.16s|\n", chaine);     ===> 2)
  printf("|%-23.16s|\n", chaine); ===> 3)
  printf("|%23.16s|\n", chaine);  ===> 4)
}
```

1) |Wolfgang Amadeus Mozart|

2) |Wolfgang Amadeus|

3) |Wolfgang Amadeus |

4) | Wolfgang Amadeus|

Fonctions scanf, fscanf, sscanf

Ces fonctions permettent d'effectuer des `entrées` formatées. Les données lues sont converties suivant les *spécifications de conversions* indiquées dans la chaîne `format`, puis stockées dans les arguments successifs fournis à sa suite. Ces arguments doivent être des *pointeurs*.

La valeur retournée correspond au nombre d'arguments correctement affectés.

La chaîne `format` peut contenir :

- des espaces ou des caractères de tabulation qui seront ignorés,
- des caractères ordinaires qui s'identifieront à ceux de l'entrée,
- des `spécifications de conversions`.

Les données en entrée sont découpées en `champs`.

Chaque champ est défini comme une chaîne de caractères qui s'étend soit :

- jusqu'à la rencontre d'un caractère d'`espacement` (« `espace` », « `tabulation` », « `fin de ligne` »),
- jusqu'à ce que la largeur du champ soit atteinte, dans le cas où celle-ci a été précisée.

Une *spécification de conversion* est constituée du caractère **%** suivi dans l'ordre :

- du signe ***** pour supprimer l'affectation de l'argument correspondant (optionnel),
- d'un nombre indiquant la largeur *maximum* du champ (optionnel),
- du type de la donnée à lire défini par l'un des caractères suivants :
  - **d** pour un entier sous forme décimale,
  - **i** pour un entier. Il peut être sous forme octale (précédé par 0) ou hexadécimale (précédé par 0x ou 0X),
  - **o** pour un entier sous forme octale (précédé ou non par 0),
  - **x** pour un entier sous forme hexadécimale (précédé ou non par 0x ou 0X),
  - **u** pour un entier non signé sous forme décimale,

- **c** pour un caractère. Il est à noter que dans ce cas il n'y a plus de notion de caractère d'espacement. Le prochain caractère est lu même s'il s'agit d'un caractère d'espacement. Si l'on veut récupérer le prochain caractère différent d'un caractère d'espacement il faut utiliser la spécification **%1s**.
- **s** pour une chaîne de caractères. La lecture continue jusqu'au prochain caractère d'espacement ou jusqu'à ce que la largeur du champ ait été atteinte. Le caractère \0 est ensuite ajouté en fin de chaîne,
- **e, f, g** pour une constante réelle.

- `[...]` pour une chaîne de caractères. Comme dans le cas de `c`, il n'y a plus de notion de caractère d'espacement. Entre « `crochets` » apparaît une suite de caractères précédée ou non du caractère `^`.

  La lecture s'effectue :

  - jusqu'au caractère différent de ceux indiqués entre « `crochets` » si ceux-ci ne sont pas précédés du caractère `^`,
  - tant que le caractère lu est différent de ceux indiqués entre « `crochets` » si ceux-ci sont précédés du caractère `^`.

  Le caractère `\0` est ensuite ajouté en fin de chaîne.

Remarques

Les arguments correspondant aux spécifications :

- **d**, **i**, **o**, **x**, **u** doivent être de type *int* $*$,
- **e**, **f**, **g** doivent être de type *float* $*$,
- **c**, **s** doivent être de type *char* $*$.

On peut faire précéder les spécifications **d**, **i**, **o**, **x**, **u** par la lettre **h** ou **l** pour référencer un *short* $*$ ou un *long* $*$.

De même les spécifications **e**, **f**, **g** peuvent être précédées de la lettre **l** pour référencer un *double* $*$.

Exemples

```
#include <stdio.h>
main()
{
  int   i;
  float x, y;
  char  buffer[BUFSIZ];
  char *p = "12/11/94";
  int   jour, mois, annee;

  scanf( "%d%f%f%*c", &i, &x, &y );
  printf( "i = %d, x = %f, y = %f\n", i, x, y );
  scanf( "%[^\n]%*c", buffer );
  while( ! feof(stdin) )
  {
    fprintf( stderr, "%s\n", buffer );
    scanf( "%[^\n]%*c", buffer );
  }
  sscanf( p, "%d/%d/%d", &jour, &mois, &annee );
  printf( "jour  : %d\n", jour );
  printf( "mois  : %d\n", mois );
  printf( "annee : %d\n", annee );
}
```

```
#include <stdio.h>
main()
{
  char   mois[10], buffer[BUFSIZ];
  int    quantite;
  double prix;
  FILE  *f;
  if( (f = fopen( "donnees", "r" )) == NULL )
    perror( "fopen" ), exit(1);
  fgets( buffer, sizeof(buffer), f );
  while( ! feof(f) )
  {
    sscanf( buffer, "%s%d%lf",
            mois, &quantite, &prix );
    printf( "mois : %s, qte : %d, prix : %f\n",
            mois, quantite, prix );
    fgets( buffer, sizeof(buffer), f );
  }
  fseek( f, 0L, SEEK_SET );
  fscanf( f, "%s%d%lf", mois, &quantite, &prix );
  while( ! feof(f) )
  {
    printf( "mois : %s, qte : %d, prix : %f\n",
            mois, quantite, prix );
    fscanf( f, "%s%d%lf", mois, &quantite, &prix );
  }
  fclose(f);
}
```

9.2.8 - Autres fonctions

La fonction **freopen** permet de redéfinir un flot déjà initialisé. Elle est principalement utilisée avec les flots *stdin*, *stdout*, *stderr*, ce qui correspond à une redirection d'entrées-sorties.

La fonction **fflush** permet de forcer le vidage de la mémoire tampon associée à un flot en sortie. Sur un flot en entrée l'effet est imprévisible.

Syntaxe

```
FILE *freopen(char *fichier, char *mode, FILE *flot);
int   fflush(FILE *flot);
```

Exemple

```
#include <stdio.h>
main( int argc, char **argv )
{
  void usage( char *s );

  if( argc != 2 )
    usage( argv[0] );
  if( freopen( argv[1], "w", stdout ) == NULL )
  {
    perror( "freopen" );
    exit(2);
  }
  printf( "Ce message est redirigé  ");
  printf( "dans le fichier  ");
  printf( "dont le nom est  ");
  printf( "passé en argument.\n ");
}

void usage(char *s)
{
  fprintf(stderr, "usage : %s fichier\n", s);
  exit(1);
}
```

9.3 – Manipulation de caractères

Le fichier en-tête **ctype.h** contient des déclarations de fonctions permettant de tester les caractères. Elles admettent un argument de type entier et retourne un entier :

- **isalnum** caractère alphanumérique,
- **isalpha** caractère alphabétique,
- **iscntrl** caractère de contrôle,
- **isdigit** caractère numérique,
- **isgraph** caractère imprimable sauf l'espace,
- **islower** caractère minuscule,
- **isupper** caractère majuscule,
- **isprint** caractère imprimable y compris l'espace,

- **ispunct** caractère imprimable différent de l'espace, des lettres et des chiffres,
- **isspace** espace, saut de page, fin de ligne, retour chariot, tabulation,
- **isxdigit** chiffre hexadécimal.

Les caractères imprimables sont compris entre **0x20** et **0x7e**, les caractères de contrôle sont compris entre **0** et **0x1f** ainsi que **0x7f**.

Il existe, de plus, deux fonctions permettant de convertir les majuscules en minuscules et réciproquement :

- **tolower** convertit en minuscule le caractère passé en argument,
- **toupper** convertit en majuscule le caractère passé en argument.

Exemple

```
#include <stdio.h>
#include <ctype.h>
main()
{
  int c;
  int NbMaj = 0;
  int NbMin = 0;
  int NbNum = 0;
  int NbAutres = 0;
  while( (c=getchar()) != EOF )
    if( isupper(c) )
      NbMaj++;
    else if( islower(c) )
      NbMin++;
    else if( isdigit(c) )
      NbNum++;
    else
      NbAutres++;
  printf( "NbMaj    : %d\n", NbMaj );
  printf( "NbMin    : %d\n", NbMin );
  printf( "NbNum    : %d\n", NbNum );
  printf( "NbAutres : %d\n", NbAutres );
}
```

9.4 – Fonctions de conversions

Le fichier en-tête **stdlib.h** contient des déclarations de fonctions permettant la conversion de données de type chaîne de caractères en données numériques :

- **double atof(const char *s)** convertit l'argument **s** en un **double**,
- **int atoi(const char *s)** convertit l'argument **s** en un **int**,
- **long atol(const char *s)** convertit l'argument **s** en un **long**,

☞ **double strtod(const char *s, char **endp)** convertit le début de l'argument **s** en un **double**, en ignorant les éventuels caractères d'espacement situés en-tête. Elle place dans l'argument **endp** l'adresse de la partie non convertie de *s*, si elle existe, sauf si *endp* vaut `NULL`. Si la valeur convertie est trop grande, la fonction retourne la pseudo-constante `HUGE_VAL` définie dans le fichier en-tête *math.h*, si elle est trop petite la valeur retournée est **0**,

☞ **long strtol(const char *s, char **endp, int base)** convertit le début de l'argument **s** en un **long** avec un traitement analogue à *strtod*. L'argument **base** permet d'indiquer la base (comprise entre **2** et **36**) dans laquelle le nombre à convertir est écrit. Si *base* vaut **0**, la base considérée est **8**, **10** ou **16** :

- ➪ un **0** en tête indique le format octal,
- ➪ les caractères **0x** ou **0X** en tête indique le format hexadécimal.

Si la valeur retournée est trop grande, la fonction retourne les pseudo-constantes `LONG_MAX` ou `LONG_MIN` suivant le signe du résultat,

☞ **unsigned long strtoul(const char *s, char **endp, int base)**

est équivalente à *strtol* mis à part que le résultat est de type *unsigned long*, et que la valeur de retour, en cas d'erreur, est la pseudo-constante `ULONG_MAX` définie dans le fichier en-tête *limits.h*.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  char *s = "     37.657a54";
  char *cerr;

  printf( "%f\n", strtod( s, &cerr ) );    ===> 37.657
  if( *cerr != '\0' )
    fprintf(  stderr,
              "Caractère erroné : %c\n", *cerr );
  s = "11001110101110";
  printf( "%ld\n", strtol( s, NULL, 2 ) ); ===> 13230
  s = "0x7fff";
  printf( "%ld\n", strtol( s, NULL, 0 ) ); ===> 32767
  s = "0777";
  printf( "%ld\n", strtol( s, NULL, 0 ) ); ===>   511
  s = "777";
  printf( "%ld\n", strtol( s, NULL, 0 ) ); ===>   777
}
```

9.5 – Manipulation de chaînes de caractères

Le fichier en-tête **string.h** contient des déclarations de fonctions permettant la manipulation de chaînes de caractères :

- **char *strcpy(char *s1, const char *s2)** copie la chaîne **s2**, y compris le caractère \0, dans la chaîne **s1**. Elle retourne la chaîne **s1**,
- **char *strncpy(char *s1, const char *s2, int n)** copie au plus **n** caractères de la chaîne **s2** dans la chaîne **s1** laquelle est complétée par des \0 dans le cas où la chaîne **s2** contient moins de **n** caractères. Cette fonction retourne la chaîne **s1**,
- **char *strdup(char *s)** retourne un pointeur sur une zone allouée dynamiquement contenant la duplication de l'argument **s**,

- **char *strcat(char *s1, const char *s2)** concatène la chaîne **s2** à la chaîne **s1**. Cette fonction retourne la chaîne **s1**,
- **char *strncat(char *s1, const char *s2, int n)** concatène au plus **n** caractères de la chaîne **s2** à la chaîne **s1** qui est ensuite terminée par `\0`. Cette fonction retourne la chaîne **s1**.

- **int strcmp(const char *s1, const char *s2)** compare la chaîne **s1** à la chaîne **s2**. Les chaînes sont comparées caractère par caractère en partant de la gauche. Cette fonction retourne :
  - une valeur négative dès qu'un caractère de la chaîne **s1** est plus petit que celui de la chaîne **s2**,
  - une valeur positive dès qu'un caractère de la chaîne **s1** est plus grand que celui de la chaîne **s2**,
  - **0** si les deux chaînes sont identiques.
- **int strncmp(const char *s1, const char *s2, int n)** compare au plus **n** caractères de la chaîne **s1** à la chaîne **s2**. La valeur retournée par cette fonction est identique à celle retournée par la fonction *strcmp*,

- **char *strchr(const char *s, int c)** retourne un pointeur sur la première occurrence du caractère **c** dans la chaîne **s**, ou **NULL** si **c** ne figure pas dans **s**,
- **char *strrchr(const char *s, int c)** retourne un pointeur sur la dernière occurrence du caractère **c** dans la chaîne **s**, ou **NULL** si **c** ne figure pas dans **s**,
- **char *strstr(const char *s1, const char *s2)** retourne un pointeur sur la première occurrence de la chaîne **s2** dans la chaîne **s1**, ou **NULL** si elle n'y figure pas.
- **size_t strlen(const char *s)** retourne la longueur de la chaîne **s**.

Le type **size_t** est un alias du type `unsigned long`.

Exemple

```
#include <stdio.h>
#include <string.h>
main( int argc, char **argv )
{
  char *parm1, *parm2, buffer[BUFSIZ];
  if( argc != 3 ) usage( argv[0] );
  parm1 = strdup( argv[1] );
  parm2 = strdup( argv[2] );
  strcat( strcpy( buffer, "Résultat de la "
                          "concaténation : " ),
          parm1 );
  strcat( strcat( buffer, parm2 ), "\n" );
  printf( "%s", buffer );
  sprintf( buffer, "%s%s%s\n", "Résultat de la "
                               "concaténation : ",
                               parm1, parm2 );
  printf( "%s", buffer );
  free( parm1 ); free( parm2 );
}
void usage( char *s )
{
  fprintf( stderr, "usage : %s ch1 ch2.\n", s );
  exit(1);
}
```

Exemple

```
#include <stdio.h>
#include <string.h>
main( int argc, char **argv )
{
  void  usage( char *s );
  char *s  = "/usr/include/string.h", *p;
  int   NbSlash = 0;

  if( argc != 3 ) usage( argv[0] );
  if( ! strcmp( argv[1], argv[2] ) )
    printf( "Les 2 arguments sont identiques.\n" );
  else if( strcmp( argv[1], argv[2] ) > 0 )
    printf( "arg1 > arg2\n" );
  else printf( "arg1 < arg2\n" );
  for( p = s-1; p = strchr( ++p, '/' ); NbSlash++ )
                ;
  printf( "La chaîne s contient %d\n", NbSlash );
  printf( "slashs sur %d caractères.\n", strlen(s) );
}
void usage( char *s )
{
  fprintf( stderr, "usage : %s ch1 ch2.\n", s );
  exit(1);
}
```

Il existe d'autres fonctions qui agissent sur des tableaux de caractères plutôt que des chaînes de caractères :

- **void *memcpy(void *m1, void *m2, size_t n)** copie **n** caractères de la zone mémoire **m2** dans la zone mémoire **m1** et retourne **m1**,
- **void *memmove(void *m1, void *m2, size_t n)** est identique à `memcpy` mais fonctionne également dans le cas où les zones mémoires **m1** et **m2** se chevauchent,

- **int memcmp(void *m1, void *m2, size_t n)** compare les **n** premiers caractères des zones mémoires **m1** et **m2**. La valeur de retour se détermine comme pour `strcmp`,
- **void *memchr(void *m, int c, size_t n)** retourne un pointeur sur la première occurrence du caractère **c** dans la zone mémoire **m**, ou `NULL` si **c** n'y figure pas,
- **void *memset(void *m, int c, size_t n)** remplit les **n** premiers caractères de la zone mémoire **m** avec le caractère **c** et retourne **m**.

Exemple

```
#include <stdio.h>
#include <string.h>
main()
{
  char  buffer[100];
  char  tab[] = "Voici\0une chaîne qui"
                "\0\0contient\0des"
                "\0caractères \"null\".";
  char *p, *ptr;
  int   taille = sizeof tab / sizeof tab[0];
  int   n;

  memset( buffer, ' ', 100 );
  memcpy( buffer, tab, taille );
  n = --taille;
  for( p=ptr=tab; p=memchr( ptr, '\0', n ); )
  {
    *p = ' ';
    n -= p - ptr + 1;
    ptr = ++p;
  }
  printf( "%s\n", buffer );
  printf( "%.*s\n", taille, tab );
}
```

9.6 – Allocation dynamique de mémoire

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

- **malloc** et **calloc** pour allouer un bloc mémoire (initialisé avec des zéros si *calloc*),
- **realloc** pour étendre sa taille,
- **free** pour le libérer.

Leurs déclarations se trouvent dans le fichier en-tête **stdlib.h**.

Syntaxe

```
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)
```

Exemple

```
#include <stdio.h>
#include <stdlib.h>
typedef struct { int nb; float *ptr; } TAB_REEL;
#define TAILLE 100
main()
{
  TAB_REEL *p; int i;
  p = (TAB_REEL *)calloc( TAILLE, sizeof(TAB_REEL) );
  if( ! p )
  {
    fprintf( stderr, "Erreur à l'allocation\n\n" );
    exit(1);
  }
  for( i=0; i < TAILLE; i++ )
  {
    p[i].nb  = TAILLE;
    p[i].ptr = (float *)malloc( p[i].nb*
                                sizeof(float) );
    p[i].ptr[i] = 3.14159f;
  }
  for( i=0; i < TAILLE; i++ ) free( p[i].ptr );
  free( p );
}
```

Exemple

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
  char **ptr = (char **)NULL;
  char   buffer[BUFSIZ];
  int    nb = 0, i;
  for( ;; )
  {
    printf( "Entrer une chaîne : " );
    scanf( "%s", buffer );
    if( ! strcmp( buffer, "fin" ) ) break;
    ptr = (char **)realloc( ptr,
                            ++nb*sizeof(char *) );
    ptr[nb-1] = (char *)malloc(
                (strlen(buffer)+1)*sizeof(char) );
    strcpy( ptr[nb-1], buffer );
  }
  for( i=0; i < nb; i++ )
  {
    printf( "%s\n", ptr[i] ); free( ptr[i] );
  }
  free( ptr );
}
```

9.7 – Date et heure courantes

Le fichier en-tête **time.h** contient des déclarations de types et de fonctions permettant de manipuler la date et l'heure :

- **clock_t clock(void)** retourne le temps CPU (en microsecondes) consommé depuis le premier appel à `clock`. Une division par la pseudo-constante `CLOCKS_PER_SEC` permet d'obtenir le temps en secondes,
- **time_t time(time_t *t)** retourne le temps en secondes écoulé depuis le 1er janvier `1970 00 :00 :00 GMT`. Si **t** est différent de `NULL`, ***t** reçoit également cette valeur,
- **time_t mktime(struct tm *t)** convertit la date et l'heure décrites par la structure pointée par **t**, en nombre de secondes écoulées depuis le 1er janvier `1970 00 :00 :00 GMT`, lequel est retourné par la fonction (ou **-1** si impossibilité),

- **char *asctime(const struct tm *t)** convertit la date et l'heure décrites par la structure pointée par **t** en une chaîne de caractères de la forme : `"Thu Oct 19 16 :45 :02 1995\n"`,
- **struct tm *localtime(const time_t *t)** convertit un temps exprimé en secondes écoulées depuis le 1er janvier `1970 00 :00 :00 GMT`, sous forme date et heure décrites par une structure de type **struct tm**. Cette fonction retourne un pointeur sur cette structure.
- **char *ctime(const time_t *t)** convertit un temps exprimé en secondes écoulées depuis le 1er janvier `1970 00 :00 :00 GMT`, en une chaîne de caractères de la même forme que la fonction `asctime`. Cette fonction équivaut à : `asctime(localtime(t))`,

Les 3 fonctions précédentes retournent des pointeurs sur des objets statiques qui peuvent être écrasés par d'autres appels.

Les types **time_t** et **clock_t** sont, respectivement, des alias de **long** et **int**.

La structure **struct tm**, explicitant la date et l'heure, contient les champs suivant (de type **int**) :

- **tm_year** : année,
- **tm_mon** : mois (`0-11`),
- **tm_wday** : jour (`0-6`),
- **tm_mday** : jour du mois,
- **tm_yday** : jour de l'année, (`0-365`),
- **tm_hour** : heures,
- **tm_min** : minutes,
- **tm_sec** : secondes.

Exemple

```
#include <stdio.h>
#include <time.h>
main()
{
  time_t t;
  struct tm *tm;

  time( &t );
  puts( ctime( &t ) );    /* Thu Feb 27 11:26:36 1997 */
  tm = localtime( &t );
  puts( asctime( tm ) ); /* Thu Feb 27 11:26:36 1997 */
  tm->tm_year = 94;
  tm->tm_mday = 16;
  tm->tm_mon  = 10;
  t = mktime( tm );
  puts( ctime( &t ) );    /* Wed Nov 16 11:26:36 1994 */
  t -= 20*86400;
  puts( ctime( &t ) );    /* Thu Oct 27 11:26:36 1994 */

  return 0;
}
```

9.8 – Accès à l'environnement

La fonction **getenv** permet d'obtenir la valeur d'une variable d'environnement du SHELL dont le nom est passé en argument.

Syntaxe

```
char *getenv(char *nom)
```

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#define RACINE        "RACINE"
#define DEFAUT_RACINE "."
main()
{
  char *racine;
  if( (racine = getenv( RACINE )) == NULL )
    racine = DEFAUT_RACINE;
  printf( "Répertoire de travail: \"%s\"\n", racine );

  return 0;
}
```

9.9 – Sauvegarde et restauration du contexte

A chaque appel d'une fonction, son **contexte** (c'est-à-dire ses paramètres, ses variables dynamiques et son adresse de retour) est empilé dans la **pile d'exécution** du processus.

Chaque retour de fonction entraine le dépilement du `contexte` courant et le retour dans le `contexte` de la fonction appelante.

Sauvegarder le `contexte` d'un processus, consiste à sauvegarder la valeur des registres à cet instant (notamment la valeur du **compteur ordinal** ou `registre d'instruction` qui contient l'adresse de la prochaine instruction machine à exécuter).

Les fonctions **setjmp** et **longjmp** permettent, respectivement, de sauvegarder et restaurer le `contexte` d'un processus. Leurs déclarations se trouvent dans le fichier en-tête **setjmp.h**.

Syntaxe

```
int  setjmp(jmp_buf cntx)
void longjmp(jmp_buf cntx, int valeur)
```

La fonction `setjmp` permet de sauvegarder le `contexte` dans le vecteur `cntx` et retourne la valeur entière `0`.

La restauration de ce `contexte` est effectuée par la fonction `longjmp` à laquelle on passe en argument le `contexte` sauvegardé. L'appel de cette fonction provoque alors une reprise de l'exécution de la fonction `setjmp`, qui a servi à sauvegarder le `contexte`, laquelle retourne cette fois-ci l'entier transmis, comme 2^{e}argument, à la fonction `longjmp`.

Exemple

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf cntx;
main()
{
  int appel_boucle();

  printf( "Terminaison avec \"%c\"\n",
           appel_boucle() );

  return 0;
}
int appel_boucle( void )
{
  void boucle( void );
  int  retour = setjmp( cntx );

  printf( "setjmp retourne %d\n", retour );
  if( retour == 0 ) boucle();

  return retour;
}
```

Exemple (suite)

```
void boucle( void )
{
  for( ;; )
  {
    char getcmd( char * );
    char c = getcmd( "-> " );

    switch( c )
    {
      case 'q':
        longjmp( cntx, c );
      default:
        printf( "Traitement de %c\n", c );
      break;
    }
  }
}
char getcmd( char *s )
{
  char c = (printf( "%s", s ), getchar());
  while( getchar() != '\n' )
      ;
  return c;
}
```

9.10 – Aide à la mise au point de programme

La pseudo-fonction **assert**, définie dans le fichier en-tête **assert.h**, émet un message d'erreur lorsque l'expression passée en argument est fausse. Ce message contient le nom du fichier source ainsi que le numéro de la ligne correspondant à l'évaluation de l'expression.

Certains compilateurs provoquent de plus l'arrêt du programme avec création d'un fichier « `image mémoire` » (*core*).

Ce mécanisme peut être désactivé en compilant le programme avec l'option `-DNDEBUG`.

Exemple

```
#include <string.h>
#include <assert.h>
#define DIM 50
main()
{
  char tab[DIM];
  void f( char *p, int n );

  memset( tab, ' ', DIM );
  f( tab, DIM+1 );
}

void f( char *p, int n )
{
  char *ptr = p;
  int   i;

  for( i=0; i < n; i++ )
  {
    assert( ptr - p < DIM );
    *ptr++ = '$';
  }
}
```

9.11 – Récupération des erreurs

En cas d'erreur, certaines fonctions (`fopen` par exemple) positionnent une variable externe **errno** déclarée dans le fichier en-tête **errno.h**. Ce fichier contient également une liste de pseudo-constantes correspondant aux différentes valeurs que peut prendre la variable `errno`.

La fonction **perror**, dont la déclaration figure dans le fichier en-tête `stdio.h`, permet d'émettre le message correspondant à la valeur positionnée dans la variable `errno`.

De plus, la fonction **strerror**, déclarée dans le fichier en-tête `string.h`, retourne un pointeur sur ce message.

Syntaxe

```
void  perror(const char *s)
char *strerror(int erreur)
```

La fonction `perror` émet, sur le flot **stderr**, le message d'erreur précédé de la chaîne passée en argument ainsi que du caractère `:`.

La fonction `strerror` retourne un pointeur sur le message d'erreur dont le numéro est passé en argument.

Exemple

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
main()
{
  FILE *f;

  if( (f = fopen( "ExistePas", "r" )) == NULL )
  {
    perror( "fopen" );
    puts( strerror( errno ) );
    exit(1);
  }
     ...
  return 0;
}
```

9.12 – Fonctions mathématiques

Le fichier en-tête **math.h** contient des déclarations de fonctions mathématiques.

- **sin(x)** : sinus de x,
- **cos(x)** : cosinus de x,
- **tan(x)** : tangente de x,
- **asin(x)** : arc sinus de x,
- **acos(x)** : arc cosinus de x,
- **atan(x)** : arc tangente de x,
- **sinh(x)** : sinus hyperbolique de x,
- **cosh(x)** : cosinus hyperbolique de x,
- **tanh(x)** : tangente hyperbolique de x,

- **exp(x)** : exponentielle de x (e^x),
- **log(x)** : logarithme népérien de x (ln(x)),
- **log10(x)** : logarithme décimal de x ($log_{10}(x)$),
- **pow(x, y)** : x^y,
- **sqrt(x)** : racine carrée de x,
- **ceil(x)** : le plus petit entier supérieur ou égal à x,
- **floor(x)** : le plus grand entier inférieur ou égal à x,
- **fabs(x)** : $|x|$,

Les arguments de ces fonctions ainsi que les valeurs qu'elles retournent sont du type **double**.

La compilation d'un programme faisant appel à ces fonctions doit être effectuée avec l'option **-lm** afin que l'éditeur de liens puissent résoudre les références externes correspondantes.

Sous UNIX SYSTEM V, la fonction **matherr** permet de gérer une erreur qui s'est produite lors de l'utilisation d'une fonction mathématique.

Le programmeur peut écrire sa propre fonction **matherr**, laquelle doit respecter la syntaxe suivante :

```
int matherr(struct exception *exp)
```

Le type **struct exception** est défini dans le fichier en-tête `math.h` comme :

```
struct exception
{
    int    type;
    char  *name;
    double arg1, arg2, retval;
};
```

- **type** : type de l'erreur,
  - `DOMAIN` : domaine erroné,
  - `SING` : valeur singulière,
  - `OVERFLOW` : dépassement de capacité,
  - `UNDERFLOW` : sous-dépassement de capacité,
  - `PLOSS` : perte partielle de chiffres significatifs,
  - `TLOSS` : perte totale de chiffres significatifs,
- **name** : nom de la fonction générant l'exception,
- **arg1, arg2** : arguments avec lesquels la fonction a été invoquée,
- **retval** : valeur retournée par défaut, laquelle peut être modifiée par la fonction `matherr`.

Si `matherr` retourne **0**, les messages d'erreurs standards et la valorisation d'`errno` interviendront, sinon ce ne sera pas le cas.

Exemple

```
#include <stdio.h>
#include <math.h>
main()
{
  double x, y;

  scanf( "%lf", &x );
  y = log( x );
  printf( "%f\n", y );

  return 0;
}
int matherr( struct exception *p )
{
  puts( "erreur détectée" );
  printf( " type : %d\n", p->type );
  printf( " name : %s\n", p->name );
  printf( " arg1 : %f\n", p->arg1 );
  printf( " arg2 : %f\n", p->arg2 );
  printf( " valeur retournée : %f\n", p->retval );
  p->retval = -1.;

  return 0;
}
```

9.13 – Fonctions de recherche et de tri

La fonction **bsearch**, dont la déclaration se trouve dans le fichier en-tête **stdlib.h**, permet de rechercher un élément d'un vecteur trié.

Syntaxe

```
void *bsearch(const void *key, const void *base,
              size_t NbElt, size_t TailleElt,
              int (*cmp)(const void *, const void *))
```

Cette fonction recherche dans le vecteur trié **base**, contenant **NbElt** éléments de taille **TailleElt**, l'élément pointé par **key**. La fonction **cmp** fournit le critère de recherche. Elle est appelée avec 2 arguments, le 1erest un pointeur sur l'élément à rechercher et le 2^{e}un pointeur sur un élément du vecteur. Cette fonction doit retourner un entier négatif, nul ou positif suivant que son 1erargument est inférieur, égal ou supérieur à son 2^{e}argument (en terme de rang dans le vecteur).

Exemple

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main( int argc, char **argv )
{
  int cmp( const void *key, const void *elt );
  int tab[] = {10, 8, 7, 4, 2, 1, 0};
  int *ptr;
  int NbElt = sizeof tab / sizeof tab[0];
  int key = (int)strtol( argv[1], NULL, 0 );

  ptr = bsearch( &key, tab, NbElt,
                              sizeof(int), cmp );
  if( ptr )
    printf( "Rg de l'élt rech. : %d\n", ptr-tab+1 );
}
int cmp( const void *key, const void *elt )
{
  if( *(int *)key < *(int *)elt )
    return 1;
  else if( *(int *)key > *(int *)elt )
    return -1;
  else return 0;
}
```

La fonction **qsort**, dont la déclaration se trouve dans le fichier en-tête **stdlib.h**, permet de trier un vecteur. Cette fonction est une réalisation de l'algorithme de tri rapide (« `quick-sort` ») dû à `Hoare (1962)`.

Syntaxe

```
void qsort(const void *base,
           size_t NbElt, size_t TailleElt,
           int (*cmp)(const void *, const void *))
```

Cette fonction trie le vecteur **base** contenant **NbElt** éléments de taille **TailleElt**. La fonction **cmp**, qui sert de critère de tri, admet 2 arguments pointant sur 2 éléments du vecteur à comparer. Elle doit renvoyer un entier obéissant aux mêmes règles que pour **bsearch**.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int cmp( const void *elt1, const void *elt2 );
  double tab[] = {3., 10., 6., 101.,
                  7., 11., 6., 14.};
  int    NbElt = sizeof tab / sizeof tab[0];
  int    i;

  qsort( tab, NbElt, sizeof(double), cmp );
  printf( "Vecteur tab trié : \n" );
  for( i=0; i < NbElt; i++ )
    printf( "%f\n", tab[i] );
}
int cmp( const void *elt1, const void *elt2 )
{
  if( *(double *)elt1 < *(double *)elt2 )
    return 1;
  else if( *(double *)elt1 > *(double *)elt2 )
    return -1;
  else return 0;
}
```

10 – Entrées-sorties de bas niveau

10.1 – Notion de descripteur de fichier

Une entrée-sortie de bas niveau est identifiée par un **descripteur de fichier** (« `file descriptor` »). C'est un entier positif ou nul. Les flots `stdin`, `stdout` et `stderr` ont comme `descripteur de fichier` respectif 0, 1 et 2.

Il existe une table des `descripteurs de fichiers` rattachée à chaque processus.

La première entrée libre dans cette table est affectée lors de la création d'un `descripteur de fichier`.

Le nombre d'entrées de cette table, correspondant au nombre maximum de fichiers que peut ouvrir simultanément un processus, est donné par la pseudo-constante `NOFILE` définie dans le fichier en-tête « `sys/param.h` ».

10.2 – Fonctions d'ouverture et de fermeture de fichier

L'affectation d'un `descripteur de fichier`, c'est-à-dire l'initialisation d'une entrée-sortie, s'effectue par l'appel aux fonctions **open** et **creat** qui sont déclarées dans le fichier en-tête **fcntl.h**.

La version `ANSI` de l'ordre **open** contient dorénavant l'appel **creat**.

Syntaxe

```
int  open(const char *fichier, int mode, mode_t acces)
int creat(const char *fichier, mode_t acces)
```

Le type **mode_t** est un alias du type `unsigned long`, défini dans le fichier en-tête « `sys/types.h` ».

L'argument **fichier** indique le fichier à ouvrir.

L'argument **mode** indique le mode d'ouverture du fichier que l'on spécifie à l'aide de pseudo-constantes définies dans le fichier en-tête « `fcntl.h` » :

- **O_RDONLY** : lecture seulement,
- **O_WRONLY** : écriture seulement,
- **O_RDWR** : lecture et écriture,
- **O_APPEND** : écriture en fin de fichier,
- **O_CREAT** : si le fichier n'existe pas il sera créé,
- **O_TRUNC** : si le fichier existe déjà il est ramené à une taille nulle,
- **O_EXCL** : provoque une erreur si l'option de création a été indiquée et si le fichier existe déjà.

Ces pseudo-constantes peuvent être combinées à l'aide de l'opérateur booléen `|`.

L'appel **creat** est équivalent à **open** avec **O_WRONLY | O_CREAT | O_TRUNC** comme mode d'accès.

L'appel à **creat**, ainsi qu'à **open** dans le cas où le mode **O_CREAT** a été indiqué, s'effectue avec un autre argument décrivant les accès `UNIX` du fichier à créer qui se combinent avec ceux définis au niveau de la commande `umask` du `SHELL`.

Ces fonctions retournent le `descripteur de fichier` ou **-1** en cas d'erreur.

La fonction **close** permet de libérer le `descripteur de fichier` passé en argument.

Exemple

```
#include <stdio.h>
#include <fcntl.h>
main()
{
  int fd1, fd2;

  if( (fd1 = open( "Fichier1",
                   O_WRONLY|O_CREAT|O_TRUNC,
                   0644 )) == -1 )
  {
    perror( "open" );
    exit(1);
  }
  if( (fd2 = creat( "Fichier2", 0644 )) == -1 )
  {
    perror( "creat" );
    exit(2);
  }
       ...
  close( fd1 );
  close( fd2 );

  return 0;
}
```

10.3 – Fonctions de lecture et d'écriture

Les fonctions **read** et **write** permettent de lire et écrire dans des fichiers par l'intermédiaire de leur `descripteur`.

Syntaxe

```
int  read(int fd, char *buffer, int NbOctets)
int write(int fd, char *buffer, int NbOctets)
```

Ces fonctions lisent ou écrivent, sur le `descripteur de fichier` **fd**, **NbOctets** à partir de l'adresse **buffer**. Elles retournent le nombre d'octets effectivement transmis.

Une valeur de retour de **read** égale à `0` signifie **fin de fichier**.

Une valeur de retour égale à **-1** correspond à la détection d'une erreur d'entrée-sortie.

Exemple

```
#include <stdio.h>
#include <fcntl.h>
main()
{
  int  fde, fds;
  char buffer[BUFSIZ];
  int  nlus;

  if( (fde = open( "entree", O_RDONLY )) == -1 )
  {
    perror( "open" );
    exit(1);
  }
  if( (fds = creat( "sortie", 0644 )) == -1 )
  {
    perror( "creat" );
    exit(2);
  }
  while( nlus = read( fde, buffer, sizeof(buffer) ) )
    write( fds, buffer, nlus );
  close( fde );
  close( fds );

  return 0;
}
```

10.4 – Accès direct

Les fonctions **tell** et **lseek** permettent de récupérer et de positionner le pointeur de position courante. Elles sont déclarées dans le fichier en-tête `unistd.h`.

Syntaxe

```
off_t lseek(int fd, off_t decalage, int origine)
int   tell(int fd)
```

Le type **off_t** est un alias du type `long` défini dans le fichier en-tête « `sys/types.h` ».

Exemple

```
#include <unistd.h>

/*
 *  fonction lisant n octets à partir
 *  de la position courante.
 */

int lire( int fd, long pos, char *buffer, int n )
{
   if( lseek( fd, pos, 0 ) > = 0 )
     return read( fd, buffer, n );
   else return -1;
}
```

10.5 – Relation entre flot et descripteur de fichier

Il est possible de changer de mode de traitement d'un fichier c'est-à-dire passer du mode `flot` (« `stream` ») au mode `descripteur` (« `bas niveau` ») ou vice-versa.

Pour ce faire, il existe :

- la pseudo-fonction **fileno** qui permet de récupérer le `descripteur` à partir d'une structure de type `FILE`,
- la fonction **fdopen** qui permet de créer une structure de type `FILE` et de l'associer à un `descripteur`.

Syntaxe

```
int   fileno(FILE *f)
FILE *fdopen(const int fd, const char *type)
```

Le mode d'ouverture indiqué au niveau de l'argument `type` de la fonction **fdopen**, doit être compatible avec celui spécifié à l'appel de la fonction **open** qui a servi à créer le descripteur.

Ces deux fonctions sont déclarées dans le fichier en-tête `stdio.h`.

Exemple

```
#include <stdio.h>
main()
{ int   fd;
  FILE *fe, *fs;
  char  buffer[BUFSIZ];

  if( (fd = creat( "copie", 0644 )) == -1 )
    perror( "creat" ), exit(1);
  write( fd, "abcdefg", 7 );
  if( (fe = fopen( "entree", "r" )) == NULL )
    perror( "fopen" ), exit(2);
  printf( "Descripteur du fichier \"entree\" : "
          "%d\n", fileno(fe) );
  if( (fs = fdopen( fd, "w" )) == NULL )
    perror( "fdopen" ), exit(3);
  fgets( buffer, sizeof buffer, fe );
  while( ! feof(fe) )
  {
    fputs( buffer, fs );
    fgets( buffer, sizeof buffer, fe );
  }
  fclose( fe ); fclose( fs );

  return 0;
}
```

11 – Annexe A

☞ Table des codes `ASCII` des caractères

11.1 – Table des codes ASCII des caractères

TAB. 4 – table des codes ASCII des caractères

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
C-@ (NUL)	0	0x00	000	espace	32	0x20	040
C-a (SOH)	1	0x01	001	!	33	0x21	041
C-b (STX)	2	0x02	002	"	34	0x22	042
C-c (ETX)	3	0x03	003	#	35	0x23	043
C-d (EOT)	4	0x04	004	$	36	0x24	044
C-e (ENQ)	5	0x05	005	%	37	0x25	045
C-f (ACK)	6	0x06	006	&	38	0x26	046
C-g (BEL)	7	0x07	007	’	39	0x27	047
C-h (BS)	8	0x08	010	(	40	0x28	050
C-i (HT)	9	0x09	011	)	41	0x29	051
C-j (LF)	10	0x0a	012	*	42	0x2a	052
C-k (VT)	11	0x0b	013	+	43	0x2b	053
C-l (FF)	12	0x0c	014	,	44	0x2c	054
C-m (CR)	13	0x0d	015	-	45	0x2d	055
C-n (SO)	14	0x0e	016	.	46	0x2e	056
C-o (SI)	15	0x0f	017	/	47	0x2f	057
C-p (DLE)	16	0x10	020	0	48	0x30	060
C-q (DC1)	17	0x11	021	1	49	0x31	061
C-r (DC2)	18	0x12	022	2	50	0x32	062
C-s (DC3)	19	0x13	023	3	51	0x33	063
C-t (DC4)	20	0x14	024	4	52	0x34	064
C-u (NAK)	21	0x15	025	5	53	0x35	065
C-v (SYN)	22	0x16	026	6	54	0x36	066
C-w (ETB)	23	0x17	027	7	55	0x37	067
C-x (CAN)	24	0x18	030	8	56	0x38	070
C-y (EM)	25	0x19	031	9	57	0x39	071
C-z (SUB)	26	0x1a	032	:	58	0x3a	072
C-[(ESC)	27	0x1b	033	;	59	0x3b	073
C-\ (FS)	28	0x1c	034	<	60	0x3c	074
C-] (GS)	29	0x1d	035	=	61	0x3d	075
C-$ (RS)	30	0x1e	036	>	62	0x3e	076
C-_ (US)	31	0x1f	037	?	63	0x3f	077

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
@	64	0x40	100	‘	96	0x60	140
A	65	0x41	101	a	97	0x61	141
B	66	0x42	102	b	98	0x62	142
C	67	0x43	103	c	99	0x63	143
D	68	0x44	104	d	100	0x64	144
E	69	0x45	105	e	101	0x65	145
F	70	0x46	106	f	102	0x66	146
G	71	0x47	107	g	103	0x67	147
H	72	0x48	110	h	104	0x68	150
I	73	0x49	111	i	105	0x69	151
J	74	0x4a	112	j	106	0x6a	152
K	75	0x4b	113	k	107	0x6b	153
L	76	0x4c	114	l	108	0x6c	154
M	77	0x4d	115	m	109	0x6d	155
N	78	0x4e	116	n	110	0x6e	156
O	79	0x4f	117	o	111	0x6f	157
P	80	0x50	120	p	112	0x70	160
Q	81	0x51	121	q	113	0x71	161
R	82	0x52	122	r	114	0x72	162
S	83	0x53	123	s	115	0x73	163
T	84	0x54	124	t	116	0x74	164
U	85	0x55	125	u	117	0x75	165
V	86	0x56	126	v	118	0x76	166
W	87	0x57	127	w	119	0x77	167
X	88	0x58	130	x	120	0x78	170
Y	89	0x59	131	y	121	0x79	171
Z	90	0x5a	132	z	122	0x7a	172
[	91	0x5b	133	{	123	0x7b	173
\	92	0x5c	134	|	124	0x7c	174
]	93	0x5d	135	}	125	0x7d	175
^	94	0x5e	136	~	126	0x7e	176
_	95	0x5f	137	C-?	127	0x7f	177

12 – Annexe B

☞ Priorité des opérateurs

12.1 – Priorité des opérateurs

Catégorie d'opérateurs	Opérateurs	Assoc.
fonction, tableau, membre de structure, pointeur sur un membre de structure	`() [] . ->`	G⇒D
opérateurs unaires	`- ++ -- ! ~ * & sizeof (type)`	D⇒G
multiplication, division, modulo	`* / %`	G⇒D
addition, soustraction	`- +`	G⇒D
opérateurs binaires de décalage	`<< >>`	G⇒D
opérateurs relationnels	`< <= > >=`	G⇒D
opérateurs de comparaison	`== !=`	G⇒D
et binaire	`&`	G⇒D
ou exclusif binaire	`^`	G⇒D
ou binaire	`|`	G⇒D
et logique	`&&`	G⇒D
ou logique	`||`	G⇒D
opérateur conditionnel	`?:`	D⇒G
opérateurs d'affectation	`= += -= *= /= %= &= ^= |= <<= >>=`	D⇒G
opérateur virgule	`,`	G⇒D

13 – Annexe C

13.1 – Énoncés des exercices

Exercice 1

Soit un programme contenant les déclarations suivantes :

```
int   i =  8;
int   j =  5;
float x =  0.005f;
float y = -0.01f;
char  c = 'c';
char  d = 'd';
```

Déterminer la valeur de chacune des expressions suivantes :

❶ `(3*i - 2*j)%(2*d - c)`

❷ `2*((i/5) + (4*(j-3))%(i + j - 2))`

❸ `i <= j`

❹ `j != 6`

❺ `c == 99`

❻ `5*(i + j) > 'c'`

❼ `(i > 0) && (j < 5)`

❽ `(i > 0) || (j < 5)`

❾ `(x > y) && (i > 0) || (j < 5)`

❿ `(x > y) && (i > 0) && (j < 5)`

Exercice 2

Soit un programme contenant les déclarations suivantes :

```
char *argv[] = {
          "Wolfgang Amadeus Mozart",
          "Ludwig van Beethoven",
          "Hector Berlioz",
          "Nicolo Paganini" };
char **p = argv;
```

Déterminer la valeur des expressions des 2 séries suivantes :

❶ `(*p++)[1]`	❶ `(*p++)[1]`
❷ `*p++[1]`	❷ `*p[1]++`
❸ `(*++p)[4]`	❸ `(*++p)[4]`
❹ `*++*p`	❹ `*++*p`

Exercice 3

Analyser les expressions contenues dans le programme suivant :

```
#include <stdio.h>
main()
{
  int a;
  int b;
  int c;

  a = 16;
  b = 2;
  c = 10;

  c += a > 0 && a <= 15 ? ++a : a/b;
  /*
   * Que dire de l'expression suivante ? :
   * -----------------------------------
   */
  a > 30 ? b = 11 : c = 100;
}
```

Exercice 4

Calculer parmi les entiers de `1` à `100` :

1. la somme des entiers **pairs**,
2. la somme des carrés des entiers **impairs**,
3. la somme des cubes de ces entiers.

Exercice 5

Écrire un programme permettant d'effectuer le produit de 2 matrices A et B. Leurs profils seront définis à l'aide de constantes symboliques. La matrice résultat C sera imprimée ligne par ligne.

Exercice 6

Écrire un programme permettant de déterminer les nombres premiers dans l'intervalle [1,n] à l'aide du crible d'Ératosthène. Il consiste à former une table avec tous les entiers naturels compris entre 2 et n et à rayer (mise à zéro), les uns après les autres, les entiers qui ne sont pas premiers de la manière suivante : dès que l'on trouve un entier qui n'a pas encore été rayé, il est déclaré premier, et on raye tous les multiples de celui-ci.

À la fin du procédé, les nombres non barrés sont des nombres premiers.

On tiendra compte du fait qu'un nombre donné peut déjà avoir été éliminé en tant que multiple de nombres précédents déjà testés.

Par ailleurs, on sait que l'on peut réduire la recherche aux nombres de 2 à sqrt(n) (si un entier non premier est strictement supérieur à sqrt(n) alors il a au moins un diviseur inférieur à sqrt(n) et aura donc déjà été rayé).

Exercice 7

Remplir un tableau de 12 lignes et 12 colonnes à l'aide des caractères `'1'`, `'2'` et `'3'` tel que :

```
1
1 2
1 2 3
1 2 3 1
1 2 3 1 2
1 2 3 1 2 3
1 2 3 1 2 3 1
1 2 3 1 2 3 1 2
1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1 2
1 2 3 1 2 3 1 2 3 1 2 3
```

Exercice 8

Écrire un programme permettant de trier les vecteurs lignes d'une matrice de nombres en ordre croissant. On s'appuiera sur l'algorithme appelé « `tri à bulle` » qui consiste à comparer 2 éléments consécutifs et les intervertir si nécessaire.

Si après avoir terminé l'exploration du vecteur au moins une interversion a été effectuée, on renouvelle l'exploration, sinon le tri est terminé.

Chaque ligne à trier sera transmise à une fonction qui effectuera le tri.

Exercice 9

Le but de cet exercice est de transformer une matrice de réels que l'on se définira. Cette transformation consiste à modifier chaque élément à l'aide d'une fonction paramétrable de la forme y = f(x).

On définira plusieurs fonctions de ce type. La valeur d'un entier défini sous la forme d'une constante symbolique indiquera la fonction à transmettre en argument de la procédure chargée d'effectuer la transformation.

Exercice 10

Écrire une fonction à laquelle on transmet des couples (entier, réel) en nombre variable, et qui retourne les sommes des entiers et des réels.

Exercice 11

Écrire un programme qui analyse les paramètres qui lui sont passés. Ce programme devra être appelé de la manière suivante :

```
exo11 -a chaine1|-b chaine2|-c chaine3 [-d -e -f] fichier
```

❶ une seule des options **-a**, **-b** ou **-c** doit être spécifiée, suivie d'une chaîne,

❷ les options **-d**, **-e**, **-f** sont facultatives. Si aucune d'entre elles n'est indiquée, le programme doit les considérer comme toutes présentes,

❸ un nom de fichier doit être spécifié.

On essaiera de rendre l'appel le plus souple possible :

1. exo11 -a chaîne -e fichier
2. exo11 -b chaîne fichier -d -f
3. exo11 -c chaîne fichier -df
   (regroupement des options **-d** et **-f**).

Exercice 12

Écrire un programme qui lit des mots sur l'entrée standard et les affiche après les avoir converti en **louchebem** (« `langage des bouchers` »).

Cette conversion consiste à :

1. reporter la 1re lettre du mot en fin de mot, suivie des lettres **'e'** et **'m'**,
2. remplacer la 1re lettre du mot par la lettre **'l'**.

Exemples

1. `vison ==> lisonvem`,
2. `vache ==> lachevem`,
3. `bonne ==> lonnebem`.

Exercice 13

Écrire une fonction **myatof** qui convertit la chaîne passée en argument, en un réel de type « `double` ».

On pourra comparer le résultat obtenu avec celui de la fonction « `atof` » de la bibliothèque standard.

Exercice 14

Écrire un programme qui lit des chaînes de caractères sur l'entrée standard.

À la rencontre de la chaîne **« la »**, il affichera la liste des chaînes déjà saisies.

À la rencontre de la chaîne **« li »**, il affichera cette liste dans l'ordre inverse.

Exercice 15

Écrire un programme dont le but est de créer, à partir du fichier « `musiciens` », deux fichiers :

❶ un fichier constitué des enregistrements du fichier « `musiciens` », mis les uns à la suite des autres en supprimant le caractère « `newline` » qui les sépare,

❷ un fichier d'index dans lequel seront rangées les positions ainsi que les longueurs des enregistrements du fichier précédent.

Exercice 16

Ce programme devra, à partir des fichiers créés par le programme de l'exercice 15, afficher :

1. la liste des enregistrements du fichier indexé des musiciens,
2. cette même liste triée par ordre alphabétique des noms des musiciens,
3. cette même liste triée par ordre chronologique des musiciens,
4. le nom du musicien mort le plus jeune, ainsi que sa durée de vie.

Exercice 17

Écrire un programme qui affichera l'enregistrement du fichier indexé des musiciens, créé par le programme de l'exercice 15, dont le rang est passé en argument. (Prévoir les cas d'erreurs).

Exercice 18

Écrire une fonction qui retourne les différentes positions d'une chaîne de caractères dans un fichier **texte** ou **binaire**.

Le nom du fichier, ainsi que la chaîne seront transmis en argument au programme.

On pourra le tester avec les arguments suivants :

1. exo18 exo18_data_bin save
2. exo18 exo18_data_bin `SAVE`
3. exo18 exo18_data_txt où-suis-je ?

Exercice 19

Écriture d'un programme intéractif de gestion d'une liste chaînée.

Ce programme affichera le menu suivant :

```
1 - AJOUTS d'éléments dans une liste chaînée.
2 - AFFICHAGE de la liste chaînée.
3 - TRI de la liste chaînée.
4 - SUPPRESSION d'éléments dans la liste.
5 - VIDER la liste.
6 - ARRÊT du programme.
```

et effectuera le traitement correspondant au choix effectué.

13.2 – Corrigés des exercices

Corrigé de l'exercice 1

```
int   i =  8;
int   j =  5;
float x =  0.005f;
float y = -0.01f;
char  c = 'c';
char  d = 'd';
```

`(3*i - 2*j)%(2*d - c)`	=	**14**
`2*((i/5) + (4*(j-3))%(i + j - 2))`	=	**18**
`i <= j`	=	**0**
`j != 6`	=	**1**
`c == 99`	=	**1**
`5*(i + j) > 'c'`	=	**0**
`(i > 0) && (j < 5)`	=	**0**
`(i > 0) || (j < 5)`	=	**1**
`(x > y) && (i > 0) || (j < 5)`	=	**1**
`(x > y) && (i > 0) && (j < 5)`	=	**0**

Corrigé de l'exercice 2

```
char *argv[] = {
          "Wolfgang Amadeus Mozart",
          "Ludwig van Beethoven",
          "Hector Berlioz",
          "Nicolo Paganini" };
char **p = argv;
```

`(*p++)[1]`	=	**'o'**	`(*p++)[1]`	=	**'o'**
`*p++[1]`	=	**'H'**	`*p[1]++`	=	**'H'**
`(*++p)[4]`	=	**'l'**	`(*++p)[4]`	=	**'r'**
`*++*p`	=	**'i'**	`*++*p`	=	**'c'**

Corrigé de l'exercice 3

```
1  #include <stdio.h>
2  int main()
3  {
4    int a;
5    int b;
6    int c;
7
8    a = 16;
9    b = 2;
10   c = 10;
11
12   /*
13    * 1) on commence par évaluer l'expression
14    *    a > 0 && a <= 15, laquelle constitue
15    *    le 1er opérande de l'opérateur ?:.
16    *    Celle-ci est fausse car les expressions
17    *    a > 0 et a <= 15 sont vraie et
18    *    fausse respectivement,
19    * 2) on évalue donc le 3eme opérande de l'opérateur
20    *    ?:, c'est-à-dire l'expression a/b,
21    * 3) et enfin on effectue l'affectation.
22    */
23   c += a > 0 && a <= 15 ? ++a : a/b;
24   printf( "c : %d\n", c );  ===>  18
```

```
25    /*
26     * Que dire de l'expression suivante? :
27     * ---------------------------------
28     */
29    a > 30 ? b = 11 : c = 100;
30
31    /*
32     * Cette expression provoque une erreur
33     * à la compilation car le troisième
34     * opérande de l'opérateur ?: est c
35     * et non pas c = 100. De ce fait,
36     * l'expression a > 30 ? b = 11 : c
37     * est d'abord évaluée. Sa valeur est ensuite utilisée
38     * comme opérande de gauche de la dernière affectation.
39     * D'où l'erreur, car cette valeur n'est pas une g-valeur.
40     *
41     * On devrait écrire :
42     */
43
44    a > 30 ? b = 11 : (c = 100);
45
46    return 0;
47  }
```

Corrigé de l'exercice 4

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int pairs, carres_impairs, cubes;
6    int i;
7
8    pairs = carres_impairs = cubes = 0;
9    /* Boucle de calcul.*/
10   for (i=1; i<=100; i++)
11   {
12     cubes += i*i*i;
13     /* "i" est-il pair ou impair?*/
14     i%2 ? carres_impairs += i*i : (pairs += i);
15   }
16   /*
17    * Impression des résultats.
18    */
19   printf( "Somme des entiers pairs entre 1 et 100 : "
20           "%d\n", pairs );
21   printf( "Somme des carrés des entiers impairs entre"
22           " 1 et 100 : %d\n", carres_impairs );
23   printf( "Somme des cubes des 100 premiers "
24           "entiers : %d\n", cubes );
25
26   printf( "\n\nFin EX04.\n" );
27
28   return 0;
29 }
```

Corrigé de l'exercice 5

```
1  #include <stdio.h>
2
3  int main()
4  {
5    const int n = 10;
6    const int m = 5;
7    const int p = 3;
8    double    a[][5] =
9        {
10         { 0.00, 0.38, 0.42, 0.91, 0.25 },
11         { 0.13, 0.52, 0.69, 0.76, 0.98 },
12         { 0.76, 0.83, 0.59, 0.26, 0.72 },
13         { 0.46, 0.03, 0.93, 0.05, 0.75 },
14         { 0.53, 0.05, 0.85, 0.74, 0.65 },
15         { 0.22, 0.53, 0.53, 0.33, 0.07 },
16         { 0.05, 0.67, 0.09, 0.63, 0.63 },
17         { 0.68, 0.01, 0.65, 0.76, 0.88 },
18         { 0.68, 0.38, 0.42, 0.99, 0.27 },
19         { 0.93, 0.07, 0.70 ,0.37, 0.44 }
20       };
21   double    b[][3] =
22       {
23         { 0.76, 0.16, 0.9047 },
24         { 0.47, 0.48, 0.5045 },
25         { 0.23, 0.89, 0.5163 },
26         { 0.27, 0.90, 0.3190 },
27         { 0.35, 0.06, 0.9866 }
28       };
29   double    c[10][3];
30   int       i,j,k;
```

```
31                    /* Produit matriciel C = A*B */
32    for( i=0; i<n; i++ )
33      for( j=0; j<p; j++ )
34      {
35        c[i][j] = 0.;
36        for( k=0; k<m; k++ )
37          c[i][j] += a[i][k] * b[k][j];
38      }
39
40                    /* Impression de la matrice C. */
41    for( i=0; i<n; i++ )
42    {
43      for( j=0; j<p; j++ )
44        printf( "%9.5f", c[i][j] );
45      printf( "\n" );
46    }
47
48    printf( "\n\nFin EXO5.\n" );
49
50    return 0;
51  }
```

Corrigé de l'exercice 6

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6    const int n = 1000;
7    int       tab_nombres[n];
8    int       imax;
9    int       i, j;
10
11   /*
12    * Remplissage du tableau "tab_nombres"
13    * à l'aide des nombres de 1 à 1000.
14    */
15   for( i=1; i<n; i++ )
16     tab_nombres[i] = i+1;
17
18   imax = (int)sqrt( (double)n );
19   for( i=1; i<imax; i++ )
20     if( tab_nombres[i] != 0 )
21       /*
22        * Suppression des multiples non déjà
23        * exclus du nombre "tab_nombres[i]".
24        */
25       for( j=i+1; j<n; j++ )
26         if( tab_nombres[j] != 0 &&
27             tab_nombres[j]%tab_nombres[i] == 0 )
28           tab_nombres[j] = 0;
```

```
29    /*
30     * Impression des nombres non exclus
31     * qui sont les nombres premiers cherchés.
32     * Impression de 10 nombres par ligne.
33     */
34    printf( "Les nombres premiers entre 1 et "
35            "%d sont :\n\n", n );
36    for( i=1; i<n; i++ )
37    {
38      static int nb_prem = 0;
39
40      if ( tab_nombres[i] != 0 )
41      {
42        if( nb_prem++%10 == 0 ) printf( "\n" );
43        printf( "%5d", tab_nombres[i] );
44      }
45    }
46
47    printf( "\n\nFin EXO6.\n" );
48
49    return 0;
50  }
```

Corrigé de l'exercice 7

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int  i, j;
6    char c, tab[12][12];
7
8    /* Boucle sur les colonnes.*/
9    for( j=0; j<12; )
10     /*
11      * On remplit par groupe de 3 colonnes avec
12      * les caractères successifs '1', '2'  et '3'.
13      */
14     for( c='1'; c<='3'; c++, j++ )
15       for( i=j; i<12; i++ )
16         tab[i][j] = c;
17   /*
18    * Impression du tableau obtenu.
19    */
20   for( i=0; i<12; i++ )
21   {
22     for( j=0; j<=i; j++ )
23       printf( "  %c", tab[i][j] );
24     printf( "\n" );
25   }
26
27   printf( "\n\nFin EXO7.\n" );
28
29   return 0;
30 }
```

Corrigé de l'exercice 8

```
1  #include <stdio.h>
2
3  #define NbElts(t) ( (sizeof(t)) / (sizeof(t[0])) )
4  #define NCOLS 4
5  typedef enum { False, True } Boolean;
6
7  int main()
8  {
9    void   tri_vec( double *t );
10   double mat[][NCOLS] =
11          {
12            {  1.56,    0.89,  10.234,  2.78  },
13            {  9.789,   2.67,  39.78,  22.34  },
14            { 99.99,  324.678,  4.56,   8.567 }
15          };
16   int i, j;
17
18   /* Tri de chaque vecteur ligne.*/
19   for( i=0; i<NbElts(mat); i++ )
20     tri_vec( mat[i] );
21   /* Impression de la matrice obtenue.*/
22   for( i=0; i<NbElts(mat); i++ )
23   {
24     for( j=0; j<NCOLS; j++ )
25       printf( "%9.3f", mat[i][j] );
26     printf( "\n" );
27   }
28   printf( "\n\nFin EXO8.\n" );
29   return 0;
30 }
```

```
31 /*
32  * Fonction effectuant le tri d'un vecteur
33  * par la méthode du tri à "bulles".
34  */
35 void tri_vec( double *t )
36 {
37   Boolean tri_termine;
38   int     i;
39
40   for( ;; )
41   {
42     tri_termine = True;
43     for( i=0; i<NCOLS-1; i++ )
44       if( t[i] > t[i+1] )
45       {
46         double temp;
47
48         temp = t[i+1];
49         t[i+1] = t[i];
50         t[i] = temp;
51         tri_termine = False;
52       }
53     if ( tri_termine ) break;
54   }
55
56   return;
57 }
```

Corrigé de l'exercice 9

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define NbElts(t) ( (sizeof(t)) / (sizeof(t[0])) )
5  #define NCOLS 4
6
7  /* Fonctions de transformations.*/
8  double identite( double x ) { return x; }
9
10 double carre   ( double x ) { return x*x; }
11
12 double cubes   ( double x ) { return x*x*x; }
13
14 int main()
15 {
16   void transform( double (*)[NCOLS],
17                   int    nb_lignes,
18                   double (*f)( double ) );
19   const int choix = 4;
20
21   double mat[][NCOLS] =
22          {
23            {  1.56,    0.89,  10.234,  2.78 },
24            {  9.789,   2.67,  39.78,  22.34 },
25            { 99.99,  324.678,  4.56,   8.567 }
26          };
27   int i, j;
```

```
28    switch( choix )
29    {
30      case 1:
31        transform( mat, NbElts(mat), identite );
32        break;
33      case 2:
34        transform( mat, NbElts(mat), carre );
35        break;
36      case 3:
37        transform( mat, NbElts(mat), cubes );
38        break;
39      case 4:
40        transform( mat, NbElts(mat), log );
41        break;
42    }
43
44    /* Impression de la matrice transformée.*/
45    for( i=0; i<NbElts(mat); i++ )
46    {
47      for( j=0; j<NCOLS; j++ )
48        printf( "%9.3f", mat[i][j] );
49      printf( "\n" );
50    }
51
52    printf( "\n\nFin EXO9.\n" );
53
54    return 0;
55  }
```

```
56 /* Fonction effectuant la transformation.*/
57 void transform( double (*p)[NCOLS],
58                  int    nb_lignes,
59                  double (*f)( double ) )
60 {
61   int i, j;
62
63   for( i=0; i<nb_lignes; i++ )
64     for( j=0; j<NCOLS; j++ )
65       p[i][j] = (*f)( p[i][j] );
66
67   return;
68 }
```

Corrigé de l'exercice 10

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  typedef struct somme
5  {
6    int    entiers;
7    double reels;
8  } Somme;
9
10 int main()
11 {
12   Somme sigma( int nb_couples, ... );
13   Somme s;
14
15   s = sigma( 4, 2, 3., 3, 10., 3, 5., 11, 132. );
16   printf( " Somme des entiers : %d\n", s.entiers );
17   printf( " Somme des réels   : %f\n", s.reels );
18   printf( "\t\t-----------------\n" );
19   s = sigma( 5, 2, 3., 3, 10., 3, 5., 11, 132., 121, 165. );
20   printf( " Somme des entiers : %d\n", s.entiers );
21   printf( " Somme des réels   : %f\n", s.reels );
22
23   printf( "\n\nFin EXO10.\n" );
24
25   return 0;
26 }
```

```
27 Somme sigma( int nb_couples, ... )
28 {
29   Somme   s;
30   int     i;
31   va_list arg;
32
33   /*
34    * Initialisation de "arg" avec l'adresse
35    * de l'argument qui suit "nb_couples".
36    * ("arg" pointe sur l'entier du 1er couple).
37    */
38   va_start( arg, nb_couples );
39   s.entiers = s.reels = 0;
40   /*
41    * Boucle de récupération des valeurs.
42    */
43   for( i=0; i<nb_couples; i++ )
44   {
45     s.entiers += va_arg( arg, int );
46     s.reels   += va_arg( arg, double );
47   }
48   va_end( arg );
49
50   return s;
51 }
```

Corrigé de l'exercice 11 : première solution

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  void traitement_option( const unsigned char option );
6  void usage( char *s );
7
8  const unsigned char  option_a = 1;
9  const unsigned char  option_b = 2;
10 const unsigned char  option_c = 4;
11 const unsigned char  option_d = 8;
12 const unsigned char  option_e = 16;
13 const unsigned char  option_f = 32;
14 unsigned char        mask_options = 0;
15 char                *valeur_option = NULL, *fichier = NULL;
16 char                *module;
17
18 void bilan_options( void );
```

```
19 int main( int argc, char **argv )
20 {
21   /* Sauvegarde du nom de l'exécutable. */
22   module = strdup( *argv );
23   /* Boucle sur les arguments. */
24   while( *++argv != NULL )
25   {
26     /*
27      * Est-ce que l'argument commence
28      * par le caractère '-' ?
29      */
30
31     if( (*argv)[0] == '-' )
32     {
33       /* On analyse les caractères qui suivent. */
34       for( (*argv)++; **argv; (*argv)++ )
35       {
36         switch( **argv )
37         {
38           case 'a':
39             traitement_option( option_a );
40             break;
41           case 'b':
42             traitement_option( option_b );
43             break;
44           case 'c':
45             traitement_option( option_c );
46             break;
47           case 'd':
48             traitement_option( option_d );
49             break;
```

```
50            case 'e':
51              traitement_option( option_e );
52              break;
53            case 'f':
54              traitement_option( option_f );
55              break;
56            default : usage( module );
57          }
58          if ( **argv == 'a' || **argv == 'b' || **argv == 'c' )
59          {
60            /*
61             * La valeur de l'option 'a', 'b' ou 'c' peut
62             * suivre immédiatement, ou bien être séparée
63             * de l'option par des blancs.
64             */
65            if( *+++argv != '\0' )
66              valeur_option = strdup( *argv );
67            /* Cas où aucune valeur ne suit. */
68            else if( (*++argv)[0] == '-' )
69              usage( module );
70            else
71              valeur_option = strdup( *argv );
72            break;
73          }
74        }
75      }
```

```
76      /*
77       * L'argument ne commence pas
78       * par le caractère '-'.
79       */
80      else if( fichier != NULL )
81        usage( module );
82      else
83        fichier = strdup( *argv );
84    }
85    bilan_options();
86
87    printf( "\n\nFin EXO11.\n" );
88
89    return 0;
90  }
91
92  void bilan_options( void )
93  {
94    /*
95     * L'option 'a', 'b', ou 'c' suivie d'une
96     * chaîne, ainsi qu'un nom de fichier
97     * doivent être spécifiés.
98     */
99
100   if( valeur_option == NULL || fichier == NULL )
101     usage( module );
```

```
102     /*
103      * Si aucune des options 'd', 'e', 'f'
104      * n'a été spécifiée, on les considère toutes.
105      */
106
107     if( ! (mask_options & option_d) &&
108         ! (mask_options & option_e) &&
109         ! (mask_options & option_f) )
110       mask_options |= option_d + option_e + option_f;
111     if( mask_options & option_a )
112       printf( "Option \"a\" fournie avec comme valeur : "
113               "%s\n", valeur_option );
114     if( mask_options & option_b )
115       printf( "Option \"b\" fournie avec comme valeur : "
116               "%s\n", valeur_option );
117     if( mask_options & option_c )
118       printf( "Option \"c\" fournie avec comme valeur : "
119               "%s\n", valeur_option );
120     printf( "Option \"d\" %s.\n",
121              mask_options & option_d ? "active" : "inactive" );
122     printf( "Option \"e\" %s.\n",
123              mask_options & option_e ? "active" : "inactive" );
124     printf( "Option \"f\" %s.\n",
125              mask_options & option_f ? "active" : "inactive" );
126
127     printf( "fichier indiqué : %s\n", fichier );
128
129     return;
130 }
```

```
131 void traitement_option( const unsigned char option )
132 {
133   /*
134    * Une seule des options "-a", "-b", "-c"
135    * doit avoir été spécifiée.
136    */
137
138   if ( option == option_a ||
139        option == option_b ||
140        option == option_c )
141     if ( valeur_option != NULL )
142       usage( module );
143
144   /*
145    * On interdit qu'une option
146    * soit indiquée 2 fois.
147    */
148
149   if ( mask_options & option )
150     usage( module );
151   else
152     mask_options |= option;
153
154   return;
155 }
156
157 void usage( char *s )
158 {
159   printf( "usage : %s -a chaine | -b chaine"
160           " | -c chaine [-d -e -f] fichier\n", s );
161   exit(1);
162 }
```

Corrigé de l'exercice 11 : deuxième solution

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  typedef enum { False, True } Boolean;
6  typedef struct options
7  {
8    char     nom[3];
9    Boolean  option_fournie;
10   Boolean  option_a_valeur;
11   char    *valeur;
12 } Option;
13 typedef enum { option_a, option_b, option_c,
14                option_d, option_e, option_f,
15                nb_options, option_invalide } option_t;
16
17 Option options[] = {
18                      { "-a", False, True,  NULL },
19                      { "-b", False, True,  NULL },
20                      { "-c", False, True,  NULL },
21                      { "-d", False, False, NULL },
22                      { "-e", False, False, NULL },
23                      { "-f", False, False, NULL }
24                    };
25
26 option_t type_option( char *option );
27 void     bilan_options( void );
28 void     usage(char *s);
29
30 char *module, *fichier = NULL;
```

```
31 int main( int argc, char **argv )
32 {
33   /* Sauvegarde du nom de l'exécutable. */
34   module = strdup( *argv );
35
36   /* Boucle sur les arguments. */
37   while( *++argv != NULL )
38   {
39     /*
40      * Est-ce que l'argument commence
41      * par le caractère '-' ?
42      */
43
44     if( (*argv)[0] == '-' )
45     {
46       option_t opt;
47
48       opt = type_option( *argv );
49       if( opt == option_invalide ) usage( module );
50       if( options[opt].option_a_valeur )
51       {
52         *argv += strlen( options[opt].nom );
53         if( argv[0][0] != '\0' )
54           options[opt].valeur = strdup( *argv );
55         /* Cas où aucune valeur ne suit. */
56         else if( (++argv)[0][0] == '-' )
57           usage( module );
58         else
59           options[opt].valeur = strdup( *argv );
60       }
61     }
```

```
62      else if( fichier != NULL )
63        usage( module );
64      else
65        fichier = strdup( *argv );
66    }
67
68    bilan_options();
69
70    printf("\n\nFin EXO11.\n");
71
72    return 0;
73  }
74
75  option_t type_option( char *option )
76  {
77    option_t rang;
78
79    for( rang=0; rang<nb_options; rang++ )
80      if ( ! strncmp( options[rang].nom, option,
81                      strlen( options[rang].nom ) ) &&
82           ! options[rang].option_fournie )
83        break;
84    if ( rang == nb_options )
85      return option_invalide;
86    if ( strcmp( options[rang].nom, option ) != 0 &&
87         ! options[rang].option_a_valeur )
88      return option_invalide;
89
90    options[rang].option_fournie = True;
91
92    return rang;
93  }
```

```
94  void bilan_options( void )
95  {
96    option_t rang;
97
98    /*
99     * Une seule des options "-a", "-b", "-c"
100    * doit avoir été spécifiée ainsi qu'un
101    * nom de fichier.
102    */
103
104   if( options[option_a].option_fournie ^
105       options[option_b].option_fournie ^
106       options[option_c].option_fournie &&
107       fichier != NULL )
108   {
109     if ( options[option_a].option_fournie &&
110          options[option_b].option_fournie &&
111          options[option_c].option_fournie ) usage( module );
112
113     /*
114      * Si aucune des options 'd', 'e', 'f'
115      * n'a été spécifiée, on les considère toutes.
116      */
117
118     if( ! options[option_d].option_fournie &&
119         ! options[option_e].option_fournie &&
120         ! options[option_f].option_fournie )
121       options[option_d].option_fournie =
122       options[option_e].option_fournie =
123       options[option_f].option_fournie = True;
```

```
124     for( rang=0; rang<nb_options; rang++ )
125       if ( options[rang].option_fournie )
126       {
127         if ( options[rang].option_a_valeur )
128           printf( "Option %s fournie avec comme valeur : "
129                   "%s\n", options[rang].nom,
130                           options[rang].valeur );
131         else
132           printf( "Option %s active.\n", options[rang].nom );
133       }
134       else if ( ! options[rang].option_a_valeur )
135         printf( "Option %s inactive.\n", options[rang].nom );
136
137     printf( "fichier indiqué : %s\n", fichier );
138   }
139   else usage( module );
140
141   return;
142 }
143
144 void usage( char *s )
145 {
146   printf( "usage : %s -a chaine | -b chaine"
147           " | -c chaine [-d -e -f] fichier\n", s );
148   exit(1);
149 }
```

Corrigé de l'exercice 12

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char  buffer[BUFSIZ];
6    char *p;
7
8    /*
9     * Boucle de lecture sur l'entrée standard
10    * avec la chaîne "--> " comme prompt.
11    */
12   fputs( "--> ", stdout );
13   gets( buffer );
14   while( ! feof(stdin) )
15   {
16     /* On se positionne à la fin du mot lu. */
17     for( p=buffer; *p; p++ );
18     /* Conversion du mot en "louchebem". */
19     p[0] = *buffer;
20     *buffer = 'l';
21     p[1] = 'e'; p[2] = 'm'; p[3] = '\0';
22     puts( buffer );
23     fputs( "--> ", stdout );
24     gets( buffer );
25   }
26
27   printf( "\n\nFin EXO12.\n" );
28
29   return 0;
30 }
```

Corrigé de l'exercice 13

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4
5  int main( int argc, char **argv )
6  {
7    void   usage ( char *s );
8    double myatof( char *s );
9
10   /* Y a-t-il un argument ? */
11   if( argc != 2 ) usage( argv[0] );
12   /*
13    * On imprime les résultats des fonctions
14    * "atof" et "myatof" pour comparaison.
15    */
16   printf( "%f\n", atof  ( argv[1] ) );
17   printf( "%f\n", myatof( argv[1] ) );
18
19   printf("\n\nFin EXO13.\n");
20
21   return 0;
22 }
23
24 double myatof( char *s )
25 {
26   long   nombre, signe;
27   double exposant;
28
29   exposant = 1.;
30   nombre   = 0;
```

```
31    /*
32     * Saut des éventuels caractères
33     * espace, tabulation et "newline"
34     * situés en tête.
35     */
36    for( ; isspace( *s ); s++ )
37        ;
38    /* Gestion du signe. */
39    signe = *s == '-' ? -1 : 1;
40    *s == '-' || *s == '+' ? s++ : s;
41    /* Gestion de la partie entière. */
42    for( ; isdigit( *s ); s++ )
43      nombre = nombre*10 + *s - '0';
44    if( *s++ != '.' )
45      return signe*nombre*exposant;
46    /* Gestion de la partie décimale. */
47    for( ; isdigit( *s ); s++ )
48    {
49      nombre = nombre*10 + *s - '0';
50      exposant /= 10.;
51    }
52
53    return signe*nombre*exposant;
54  }
55
56  void usage( char *s )
57  {
58    fprintf( stderr, "usage: %s nombre.\n", s );
59    exit( 1 );
60  }
```

Corrigé de l'exercice 14

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /* Définitions de nouveaux types. */
6  typedef enum sens {arriere, avant} Sens;
7  typedef struct cellule
8  {
9     char   *chaine;
10    struct  cellule *ptr_precedent;
11    struct  cellule *ptr_suivant;
12 } CEL;
13
14 void  liste ( CEL *p, Sens sens );
15 void  libere( CEL *p );
16 CEL  *debut = NULL;
17
18 int main()
19 {
20   CEL  *ptr_courant = NULL;
21   char  chaine[40];
22
23   /*
24    * Boucle de lecture sur l'entrée standard
25    * avec "--> " comme prompt.
26    */
27   fputs( "--> ", stdout );
28   gets( chaine );
```

```
29    while( ! feof( stdin ) )
30    {
31      /*
32       * Si "la" est la chaîne entrée,
33       * on liste les chaînes déjà saisies.
34       */
35      if( ! strcmp( chaine, "la" ) )
36        liste( debut, avant );
37      /*
38       * Si "li" est la chaîne entrée,
39       * on liste les chaînes déjà saisies
40       * dans l'ordre inverse.
41       */
42      else if( ! strcmp( chaine, "li" ) )
43        liste( ptr_courant, arriere );
44      else
45      {
46        /* C'est la 1ère chaîne. */
47        if( debut == NULL )
48        {
49          debut = malloc( sizeof(CEL) );
50          debut->ptr_precedent = NULL;
51          ptr_courant = debut;
52        }
53        else
54        {
55          /* C'est une chaîne différente de la 1ère. */
56          ptr_courant->ptr_suivant = malloc( sizeof(CEL) );
57          ptr_courant->ptr_suivant->ptr_precedent = ptr_courant;
58          ptr_courant = ptr_courant->ptr_suivant;
59        }
```

```
60        /* On valorise le nouvel élément de la liste. */
61        ptr_courant->chaine = strdup( chaine );
62        ptr_courant->ptr_suivant = NULL;
63      }
64      fputs( "--> ", stdout );
65      gets( chaine );
66    }
67    /* On libère la liste. */
68    if( debut != NULL )
69      libere( debut );
70
71    printf( "\n\nFin EXO14.\n" );
72
73    return 0;
74  }
75
76  /* Fonction récursive d'affichage de la liste. */
77  void liste( CEL *p, Sens sens )
78  {
79    if( debut == NULL )
80    {
81      printf( "Désolé! la liste est vide.\n\n" );
82      return;
83    }
84    if ( p != NULL )
85    {
86      printf( "\t%s\n", p->chaine );
87      liste( sens == avant ?
88                p->ptr_suivant : p->ptr_precedent, sens );
89    }
90
91    return;
92  }
```

```
93  /*
94   * Fonction libérant la mémoire
95   * occupée par la liste.
96   */
97
98  void libere( CEL *p )
99  {
100   if ( p->ptr_suivant != NULL )
101     libere( p->ptr_suivant );
102
103   free( p->chaine );
104   free( p );
105
106   return;
107 }
```

Corrigé de l'exercice 15

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    typedef struct index
8    {
9       unsigned int debut;
10      unsigned int longueur;
11   } INDEX;
12   FILE  *mus, *ind, *musind;
13   char   buffer[BUFSIZ];
14   INDEX  index;
15
16   /* Ouverture du fichier texte "musiciens". */
17   if( (mus = fopen( "musiciens", "r" )) == NULL )
18   {
19     perror( "fopen" );
20     exit(1);
21   }
22   /*
23    * Ouverture en écriture du fichier
24    * des musiciens indexés.
25    */
26   if( (musind = fopen( "musiciens.indexe", "w" )) == NULL )
27   {
28     perror( "fopen" );
29     exit(2);
30   }
```

```
31    /* Ouverture en écriture du fichier d'index. */
32    if( (ind = fopen( "musiciens.index", "w" )) == NULL )
33    {
34      perror( "fopen" );
35      exit(3);
36    }
37
38    index.debut = ftell( mus );
39    /* Boucle de lecture du fichier des musiciens. */
40    fgets( buffer, sizeof buffer, mus );
41    while( ! feof(mus) )
42    {
43      /* On supprime le caractère "newline". */
44      buffer[strlen( buffer )-1] = '\0';
45      fputs( buffer, musind );
46      index.longueur = strlen( buffer );
47      fwrite( &index, sizeof index, 1, ind );
48      /*
49       * Mise à jour de la position pour
50       * l'itération suivante.
51       */
52      index.debut = ftell( musind );
53      fgets( buffer, sizeof buffer, mus );
54    }
55    /* Fermeture des fichiers. */
56    fclose( ind ); fclose( musind ); fclose( mus );
57
58    printf( "\n\nFin EXO15.\n" );
59
60    return 0;
61  }
```

Corrigé de l'exercice 16

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /* Définition de types. */
6  typedef struct date
7  {
8    unsigned int date_naiss;
9    unsigned int date_mort;
10 } DATE;
11 typedef struct musicien
12 {
13   char nom[30];
14   char prenom[20];
15   DATE date;
16 } MUSICIEN;
17 typedef struct index
18 {
19   unsigned int debut;
20   unsigned int longueur;
21 } INDEX;
```

```
22 int main()
23 {
24   MUSICIEN  mort_le_plus_jeune( MUSICIEN *mus, int n );
25   void      imprime           ( MUSICIEN *mus, int n );
26   int       cmp_alpha ( const void *mus1, const void *mus2 );
27   int       cmp_chrono( const void *mus1, const void *mus2 );
28   FILE     *ind, *musind;
29   INDEX     index;
30   MUSICIEN *p_mus;
31   MUSICIEN  mus_mort_le_plus_jeune;
32   char     *buffer;
33   int       NbMus;
34
35   /* Ouverture du fichier index. */
36   if( (ind = fopen( "musiciens.index", "r" )) == NULL )
37   {
38     perror( "fopen" );
39     exit(1);
40   }
41   /* Ouverture du fichier indexé des musiciens. */
42   if( (musind = fopen( "musiciens.indexe", "r" )) == NULL )
43   {
44     perror( "fopen" );
45     exit(2);
46   }
47   NbMus = 0;
48   p_mus = NULL;
49   /*
50    * Boucle de lecture du fichier indexé des musiciens,
51    * par l'intermédiaire du fichier d'index.
52    */
```

```
53    fread( &index, sizeof(INDEX), 1, ind );
54    while( ! feof(ind) )
55    {
56      buffer = malloc( index.longueur+1 );
57      fgets( buffer, index.longueur+1, musind );
58      p_mus = realloc( p_mus, ++NbMus*sizeof(MUSICIEN) );
59      sscanf( buffer, "%s%s%d%d",
60                       p_mus[NbMus-1].prenom,
61                       p_mus[NbMus-1].nom,
62                      &p_mus[NbMus-1].date.date_naiss,
63                      &p_mus[NbMus-1].date.date_mort );
64      free( buffer );
65      fread( &index, sizeof(INDEX), 1, ind );
66    }
67    /* Fermeture des fichiers. */
68    fclose( ind ); fclose( musind );
69    /* Affichage de la liste des musiciens. */
70    printf( "\n\n\t\tListe des musiciens :\n" );
71    printf( "\t\t--------------------\n\n" );
72    imprime( p_mus, NbMus );
73    /*
74     * Tri puis affichage de la liste des musiciens
75     * triés par ordre alphabétique.
76     */
77    qsort( p_mus, NbMus, sizeof(MUSICIEN), cmp_alpha );
78    printf( "\n\n\tListe des musiciens"
79            " par ordre alphabétique :\n" );
80    printf( "\t-----------------------"
81            "--------------------\n\n" );
82    imprime( p_mus, NbMus );
```

```
83    /*
84     * Tri puis affichage de la liste des musiciens
85     * triés par ordre chronologique.
86     */
87    qsort( p_mus, NbMus, sizeof(MUSICIEN), cmp_chrono );
88    printf( "\n\n\tListe des musiciens"
89            " par ordre chronologique :\n" );
90    printf( "\t-----------------------"
91            "--------------------\n\n" );
92    imprime( p_mus, NbMus );
93
94    /* Recherche du musicien mort le plus jeune. */
95    mus_mort_le_plus_jeune =  mort_le_plus_jeune( p_mus, NbMus );
96    /*
97     * Affichage du musicien mort le plus jeune, ainsi
98     * que sa durée de vie.
99     */
100   printf( "\n\n\tLe musicien mort le plus jeune est :"
101           "\n\t---------------------------------\n\n" );
102   printf( "\t%s %s qui est mort à %d ans.\n\n",
103            mus_mort_le_plus_jeune.prenom,
104            mus_mort_le_plus_jeune.nom,
105            mus_mort_le_plus_jeune.date.date_mort -
106            mus_mort_le_plus_jeune.date.date_naiss );
107
108   printf( "\n\nFin EXO16.\n" );
109
110   return 0;
111 }
```

```
112 /*
113  * Fonction appelée par "qsort" pour trier les
114  * musiciens par ordre alphabétique.
115  */
116 int cmp_alpha( const void *mus1, const void *mus2 )
117 {
118   if( strcmp( ((MUSICIEN *)mus1)->nom,
119               ((MUSICIEN *)mus2)->nom ) > 0 )
120     return 1;
121   if( strcmp( ((MUSICIEN *)mus1)->nom,
122               ((MUSICIEN *)mus2)->nom) < 0 )
123     return -1;
124   return 0;
125 }
126
127 /*
128  * Fonction appelée par "qsort" pour trier les
129  * musiciens par ordre chronologique.
130  */
131 int cmp_chrono( const void *mus1, const void *mus2 )
132 {
133   if( ((MUSICIEN *)mus1)->date.date_naiss >
134       ((MUSICIEN *)mus2)->date.date_naiss )
135     return 1;
136   if( ((MUSICIEN *)mus1)->date.date_naiss <
137       ((MUSICIEN *)mus2)->date.date_naiss )
138     return -1;
139   return 0;
140 }
```

```
141 /* Fonction d'affichage. */
142 void imprime( MUSICIEN *mus, int n )
143 {
144   int i;
145
146   for( i=0; i<n; i++ )
147     printf( "%-20s%-25s %5d %5d\n", mus[i].prenom, mus[i].nom,
148                                     mus[i].date.date_naiss,
149                                     mus[i].date.date_mort );
150
151   return;
152 }
153
154 /*
155  * Fonction recherchant le musicien
156  * mort le plus jeune.
157  */
158 MUSICIEN mort_le_plus_jeune( MUSICIEN *mus, int n )
159 {
160   int indice = 0;
161   int m;
162
163   for( m=1; m<n; m++ )
164     if( mus[m].date.date_mort - mus[m].date.date_naiss <
165         mus[indice].date.date_mort -
166         mus[indice].date.date_naiss )
167       indice = m;
168
169   return mus[indice];
170 }
```

Corrigé de l'exercice 17

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct index
5  {
6    unsigned int debut;
7    unsigned int longueur;
8  } INDEX;
9
10 int main( int argc, char **argv )
11 {
12   void   erreur(int rc);
13   void   usage (char *s);
14   FILE  *ind, *musind;
15   int    rang_mus;
16   INDEX  index;
17   char  *buffer;
18
19   /* Le rang a-t-il été spécifié ? */
20   if( argc != 2 ) usage( argv[0] );
21   /* Conversion de l'argument en entier. */
22   rang_mus = (int)strtol( argv[1], NULL, 0 );
23   if( rang_mus <= 0 )
24     erreur( 1 );
25   /* Ouverture du fichier indexé des musiciens. */
26   if( (musind = fopen( "musiciens.indexe", "r" )) == NULL )
27     perror( "fopen" ), exit(2);
28   /* Ouverture du fichier d'index. */
29   if( (ind = fopen( "musiciens.index", "r" )) == NULL )
30     perror( "fopen" ), exit(3);
```

```
31    /*
32     * Positionnement dans le fichier d'index.
33     * Ne pas trop compter sur la valeur retournée
34     * par la fonction "fseek". Un mauvais positionnement
35     * provoquera une erreur lors de la lecture suivante.
36     */
37    fseek( ind, (rang_mus-1)*sizeof(INDEX), SEEK_SET );
38    /*
39     * Lecture de l'index contenant le positionnement et
40     * la longueur de l'enregistrement, dans le fichier
41     * indexé des musiciens, correspondant au rang spécifié.
42     */
43    if( fread( &index, sizeof index, 1, ind ) != 1 )
44      erreur( 4 );
45    /*
46     * Positionnement puis lecture
47     * de l'enregistrement désiré.
48     */
49    fseek( musind, index.debut, SEEK_SET );
50    buffer = malloc( index.longueur+1 );
51    fgets( buffer, index.longueur+1, musind );
52    /* Affichage du musicien sélectionné. */
53    printf( "\n\tmusicien de rang %d ==> %s\n\n",
54             rang_mus, buffer );
55    free( buffer );
56    /* Fermeture des fichiers. */
57    fclose( ind ); fclose( musind );
58
59    printf("\n\nFin EXO17.\n");
60
61    return 0;
62  }
```

```
63 void erreur( int rc )
64 {
65   fprintf( stderr, "rang invalide.\n" );
66   exit( rc );
67 }
68
69 void usage( char *s )
70 {
71   fprintf( stderr, "usage : %s rang\n", s );
72   exit( 6 );
73 }
```

Corrigé de l'exercice 18

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7
8  int *positions;
9  int  nb_positions;
10 int  nombre_octets_lus;
11
12 static void init( void )
13 {
14   positions = NULL;
15   nb_positions = 0;
16   nombre_octets_lus = 0;
17
18   return;
19 }
20
21 int main( int argc, char **argv )
22 {
23    void  usage(char *s);
24    void  strrech( char *buffer, int nblus,
25                   char *ChaineAchercher, size_t longueur );
26    int     fd;
27    char    buffer[BUFSIZ];
28    char   *ChaineAchercher;
29    size_t  longueur;
30    int     nblus;
```

```
31      /*
32       * Le fichier et la chaîne à chercher
33       * ont-ils été passés en argument ?
34       */
35
36      if ( argc != 3 )
37        usage( argv[0] );
38      ChaineAchercher = argv[2];
39      longueur = strlen( ChaineAchercher );
40
41      /*
42       * initialisation des variables globales.
43       */
44      init();
45
46      /* Ouverture du fichier passé en argument. */
47
48      if( (fd = open( argv[1], O_RDONLY )) == -1 )
49      {
50         perror( "open" );
51         exit( 1 );
52      }
```

```
53    /* Boucle de lecture du fichier. */
54
55    while( nblus = read( fd, buffer, sizeof buffer ) )
56    {
57      /*
58       * Récupération des positions de la chaîne
59       * dans le buffer courant.
60       */
61
62      strrech( buffer, nblus, ChaineAchercher, longueur );
63
64      /*
65       * Si BUFSIZ caractères ont été lus, on recule de
66       * (longueur-1) caractères dans le fichier,
67       * pour être sûr de n'oublier aucune position de la
68       * chaîne lors de la lecture suivante.
69       */
70
71      nombre_octets_lus += nblus;
72      if( nblus == BUFSIZ )
73      {
74        lseek( fd, -(long)(longueur - 1), SEEK_CUR );
75        nombre_octets_lus -= longueur - 1;
76      }
77    }
78    close( fd );
```

```
79
80      /* Impression des positions trouvées. */
81
82      if ( nb_positions == 0 )
83        printf( "La chaîne \"%s\" n'a pas été trouvée\n"
84                "dans le fichier \"%s\".\n",
85                 ChaineAchercher, argv[1] );
86      else
87      {
88        int pos;
89
90        printf( "Dans le fichier \"%s\", la chaîne \"%s\"\n"
91                "a été trouvée aux positions :\n\n",
92                 argv[1], ChaineAchercher );
93        for( pos=0; pos<nb_positions; )
94        {
95          printf( "%5d", positions[pos] );
96          if( ! ((++pos)%12) ) printf( "\n" );
97        }
98        printf( "\n" );
99      }
100     free( positions );
101
102     printf( "\n\nFin EXO18.\n" );
103 }
```

```
104 /*
105  * Fonction de récupération des positions
106  * de la chaîne dans le buffer courant.
107  */
108
109 void strrech( char *s, int nblus,
110               char *ChaineAchercher, size_t longueur )
111 {
112    char        *buffer, *ptr;
113    static int   n = 0;
114    int          i;
115
116    /*
117     * On prend garde de remplacer les éventuels caractères
118     * "nuls" par des blancs pour pouvoir utiliser
119     * la fonction "strstr".
120     */
121
122    buffer = malloc( nblus+1 );
123    memcpy( buffer, s, nblus );
124    for( i=0; i<nblus; i++ )
125      buffer[i] = buffer[i] ? buffer[i] : ' ';
126    buffer[nblus] = '\0';
```

```
127     /* Boucle de recherche de la chaîne. */
128
129     for( ptr=buffer; ptr=strstr( ptr, ChaineAchercher );
130          ptr+=longueur )
131     {
132       /* extension du vecteur positions. */
133       positions = (int *)realloc( positions, ++n*sizeof(int) );
134       assert( positions != NULL );
135
136       /*
137        * position de la chaîne trouvée par
138        * rapport au début du bloc lu.
139        */
140       positions[n-1] = ptr - buffer + 1;
141
142       /*
143        * position de la chaîne trouvée par
144        * rapport au début du fichier.
145        */
146       positions[n-1] += nombre_octets_lus;
147     }
148     free( buffer );
149     nb_positions = n;
150
151     return;
152 }
153
154 void usage(char *s)
155 {
156     fprintf( stderr, "usage: %s fichier"
157                      " ChaineAchercher.\n", s );
158     exit(1);
159 }
```

Corrigé de l'exercice 19

Fichier exo19.h

```
1 #define taille(t) sizeof(t) / sizeof(t[0])
2 typedef enum bool {False, True} Boolean;
```

Fichier exo19_gestion_liste.h

```
1 void ajouts     ( void );
2 void liste      ( void );
3 void tri        ( void );
4 void suppression( void );
5 void vider      ( void );
6 void arret      ( void );
```

Fichier exo19.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "exo19.h"
5  #include "exo19_gestion_liste.h"
6
7  struct menu
8  {
9    char  *texte;
10   void (*action)( void );
11 };
12
13 int main()
14 {
15   /* Définition du menu. */
16   struct menu menu[] =
17   {
18     {"  1 - AJOUTS d'éléments dans une liste chaînée.\n",
19                                                   ajouts},
20     {"  2 - AFFICHAGE de la liste chaînée.\n",
21                                                    liste},
22     {"  3 - TRI de la liste chaînée.\n",
23                                                      tri},
24     {"  4 - SUPPRESSION d'éléments dans la liste.\n",
25                                                suppression},
26     {"  5 - VIDER la liste.\n",
27                                                    vider},
28     {"  6 - ARRÊT du programme.\n",
29                                                    arret}
30   };
```

```
31    int SelectionMenu( struct menu menu[], int NbChoix );
32
33    /* Boucle infinie sur les choix effectués. */
34    for (;;)
35      menu[SelectionMenu( menu, taille(menu) )].action();
36  }
```

```
37 /* Fonction renvoyant le choix effectué. */
38
39 int SelectionMenu( struct menu menu[], int NbChoix )
40 {
41   int   choix, m;
42   char  entree[10];
43   char *endp;
44
45   do
46   {
47     printf( "\n\nListe des choix :\n" );
48     for( m=0; m<NbChoix; m++ )
49       printf( "%s", menu[m].texte );
50
51     /*
52      * Lecture du choix.
53      * Attention : si "scanf", lire le "newline"
54      * car, par la suite, les lectures s'effectueront
55      * à l'aide de la fonction "gets".
56      *  scanf("%d%*c", &choix);
57      */
58
59     gets( entree );
60     choix = (int)strtol( entree, &endp, 0 );
61     if( *endp != '\0' ||
62         choix < 1 || choix > NbChoix )
63       printf( "\nERREUR - choix invalide.\n" );
64   } while( *endp != '\0' ||
65            choix < 1 || choix > NbChoix );
66   printf("\n");
67
68   return --choix;
69 }
```

Fichier exo19_gestion_liste.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "exo19.h"
5  #include "exo19_gestion_liste.h"
6
7  #define LISTE_VIDE "La liste est vide.\n"
8
9  static const char * const prompt_ajout       =
10     "Élément à ajouter[CtrlD pour terminer] --> ";
11 static const char * const prompt_suppression =
12     "Élément à supprimer[CtrlD pour terminer] --> ";
13 static const char *prompt;
14
15 typedef struct cellule
16 {
17   char           *capitale;
18   struct cellule *ptr_precedent;
19   struct cellule *ptr_suivant;
20 } CEL;
21
22 static CEL  *debut   = NULL;
23 static CEL  *curseur = NULL;
24
25 static Boolean  liste_vide( void );
26 static void     ajout_cellule( char *chaine );
27 static void     suppression_cellule( void );
28 static Boolean  recherche_cellule( char *chaine );
29 static char    *lire_chaine( void );
30 static void     affichage_liste( CEL *p );
```

```
31 static Boolean liste_vide( void )
32 {
33   return debut == NULL ? True : False;
34 }
35
36 static void ajout_cellule( char *chaine )
37 {
38   CEL *p;
39
40   /*
41    * Allocation, valorisation,
42    * insertion du nouvel élément.
43    */
44
45   p = malloc( sizeof(CEL) );
46   p->capitale = chaine;
47
48   if ( liste_vide() )
49     p->ptr_suivant = p->ptr_precedent = NULL;
50   else
51   {
52     if ( curseur != debut )
53       curseur->ptr_precedent->ptr_suivant = p;
54     p->ptr_precedent = curseur->ptr_precedent;
55     curseur->ptr_precedent = p;
56     p->ptr_suivant = curseur;
57   }
58   curseur = p;
59   if( curseur->ptr_precedent == NULL )
60     debut = curseur;
61
62   return;
63 }
```

```
64 static void suppression_cellule( void )
65 {
66   if( curseur == debut )
67   {
68     /*
69      * L'élément à supprimer est le 1er de la liste.
70      */
71
72     debut = curseur->ptr_suivant;
73     if( ! liste_vide() )
74       debut->ptr_precedent = NULL;
75   }
76   else
77   {
78     /*
79      * L'élément à supprimer n'est pas le 1er de la liste.
80      */
81
82     curseur->ptr_precedent->ptr_suivant =
83                             curseur->ptr_suivant;
84     if( curseur->ptr_suivant != NULL )
85       /*
86        * L'élément à supprimer n'est
87        * pas le dernier de la liste.
88        */
89       curseur->ptr_suivant->ptr_precedent =
90                             curseur->ptr_precedent;
91   }
```

```
92    {
93      CEL *p = curseur;
94
95      free( p->capitale ); free( p );
96      if ( curseur->ptr_suivant != NULL )
97        curseur = curseur->ptr_suivant;
98      else
99        curseur = debut;
100   }
101
102   return;
103 }
104
105 static Boolean recherche_cellule( char *chaine )
106 {
107   CEL *p;
108
109   for( p=debut; p; p=p->ptr_suivant )
110     if ( ! strcmp( p->capitale, chaine ) )
111       break;
112
113   if( p != NULL )
114   {
115     curseur = p;
116     return True;
117   }
118
119   return False;
120 }
```

```
121 static char *lire_chaine( void )
122 {
123   char buffer[BUFSIZ];
124
125   /*
126    * Lecture de l'élément à ajouter.
127    */
128
129   fputs( prompt, stdout );
130   gets( buffer );
131
132   /*
133    * Si Control-D, annuler le bit indicateur
134    * de fin de fichier, pour les prochaines saisies.
135    */
136
137   if( feof( stdin ) )
138   {
139     clearerr( stdin );
140     return NULL;
141   }
142
143   return strdup( buffer );
144 }
```

```
145 /*
146  * Fonction rattachée au choix 1.
147  * (AJOUTS d'éléments dans la liste chaînée).
148  */
149
150 void ajouts( void )
151 {
152   char *chaine;
153
154   /*
155    * Boucle de lecture des chaînes.
156    */
157
158   prompt = prompt_ajout;
159
160   while( (chaine = lire_chaine()) != NULL )
161     ajout_cellule( chaine );
162
163   return;
164 }
```

```
165 /*
166  * Fonction rattachée au choix 3.
167  * (TRI de la liste chaînée).
168  */
169
170 void tri( void )
171 {
172   Boolean  tri_terminee;
173   CEL     *ptr;
174
175   /*
176    * La liste doit exister.
177    */
178
179   if ( liste_vide() )
180     fprintf( stderr, LISTE_VIDE );
181   else
182   {
183     /*
184      * Boucle de tri.
185      */
```

```
186     do
187     {
188       tri_terminee = True;
189       for( ptr=debut; ptr->ptr_suivant;
190            ptr = ptr->ptr_suivant )
191         if( strcmp( ptr->capitale,
192                     ptr->ptr_suivant->capitale ) > 0 )
193         {
194           /*
195            * On effectue une interversion.
196            */
197
198           curseur = ptr;
199           ajout_cellule(
200             strdup( curseur->ptr_suivant->capitale ) );
201           curseur = ptr->ptr_suivant;
202           suppression_cellule();
203           tri_terminee = False;
204           if ( ptr->ptr_suivant == NULL )
205             break;
206         }
207     } while( ! tri_terminee );
208   }
209
210   return;
211 }
```

```
212 /*
213  * Fonction rattachée au choix 4.
214  * (SUPPRESSION d'éléments dans la liste).
215  */
216 void suppression( void )
217 {
218   char *chaine;
219
220   /*
221    * Boucle de lecture des chaînes.
222    */
223   prompt = prompt_suppression;
224
225   while( ! liste_vide() && (chaine = lire_chaine()) != NULL )
226   {
227     if( ! recherche_cellule( chaine ) )
228     {
229       fprintf( stderr, "L'élément \"%s\" est"
230                        " inexistant!\n\n", chaine );
231       continue;
232     }
233     suppression_cellule();
234     printf( "L'élément \"%s\" a été supprimé"
235             " de la liste.\n\n", chaine );
236   }
237
238   /*
239    * La liste est-elle vide ?
240    */
241   if ( liste_vide() ) fprintf( stderr, LISTE_VIDE );
242
243   return;
244 }
```

```
245 /*
246  * Fonction rattachée au choix 5.
247  * (VIDER la liste ).
248  */
249 void vider( void )
250 {
251   if ( liste_vide() )
252     fprintf( stderr, LISTE_VIDE );
253   else
254   {
255     curseur = debut;
256     while ( ! liste_vide() )
257       suppression_cellule();
258   }
259
260   return;
261 }
262 /*
263  * Fonction rattachée au choix 6.
264  * (ARRET du programme).
265  */
266 void arret( void )
267 {
268   /*
269    * Si la liste n'est pas vide, on libère
270    * la mémoire qu'elle occupe.
271    */
272   if( ! liste_vide() ) vider();
273
274   printf( "\n\nFin EXO19.\n" );
275
276   exit( 0 );
277 }
```

Index

– Symboles –

– A –

– B –

– C –

– D –

– E –

– F –

– G –

– I –

– K –

– L –

– M –

– N –

– O –

– P –

– Q –

– R –

– S –

– T –

– U –

– V –

– W –

– X –