Langage C

Patrick Corde

Patrick.Corde@idris.fr

4 juillet 2012

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Table des matières I

1. Présentation du langage C
   - Historique
   - Intérêts du langage
   - Qualités attendues d'un programme
2. Généralités
   - Jeu de caractères utilisés
   - Identificateurs et mots-clés
   - Structure d'un programme C
   - Compilation et édition des liens
3. Les déclarations
   - Les types de base
   - Les énumérations de constantes
   - Les pointeurs
   - Forme générale d'une déclaration
   - Constructeurs homogènes
   - Constructeurs hétérogènes
   - Définitions de types
4. Expressions et opérateurs
   - Constantes littérales
     - Constantes entières
     - Constantes réelles
     - Constantes caractères
     - Constantes chaîne de caractères
     - Constantes agrégats
     - Constantes symboliques

Table des matières II

Opérateurs arithmétiques
Opérateurs logiques
Opérateurs de taille
Opérateurs d'adressage et d'indirection
Opérateurs de forçage de type
Opérateurs de manipulations de bits
- Opérateurs arithmétiques "bit à bit"
- Opérateurs de décalage

Opérateurs d'affectation
- Opérateurs à effet de bord
- Opérateurs d'incrémentation et de décrémentation

Opérateurs conditionnel
Opérateurs séquentiel
Opérateurs d'indexation
Opérateurs d'appel de fonction
Opérateurs de sélection de champ

5 Portée et visibilité
Niveau d'une variable
Durée de vie d'une variable
Classes de mémorisation
Portée des variables internes
Portée des variables externes
- Programme monofichier
- Programme multifichiers

Table des matières III

Initialisation des variables
Visibilité des fonctions

6 Instructions
Instructions élémentaires
Structures de contrôle
- Les tests
- les boucles tant-que
- la boucle for
- L'aiguillage

Instructions d'échappement

7 Préprocesseur
Introduction
Pseudo-constantes
Pseudo-fonctions
Inclusion de fichiers
Compilation conditionnelle

8 Les fonctions
Passage arguments-paramètres
Fonction retournant un pointeur
Passage d'un vecteur comme argument
Passage d'une structure comme argument
Passage d'une fonction comme argument
fonction inline
Passage d'arguments à la fonction main

Table des matières IV

Fonction avec un nombre variable d'arguments

9 La bibliothèque standard
- Notion de pointeur générique
- Entrées-sorties de haut niveau
- Fonctions d'ouverture et de fermeture
- Lecture et écriture de caractères
- Lecture et écriture de mots
- Lecture et écriture de chaînes de caractères
- Lecture et écriture de blocs
- Accès direct
- Entrées-sorties formatées
- Autres fonctions
- Manipulation de caractères
- Fonctions de conversions
- Manipulation de chaînes de caractères
- Manipulation de tableaux d'octets
- Allocation dynamique de mémoire
- Date et heure courantes
- Accès à l'environnement
- Sauvegarde et restauration du contexte
- Aide à la mise au point de programme
- Récupération des erreurs
- Fonctions mathématiques

Table des matières V

Fonctions de recherche et de tri

10 les entrées-sorties de bas niveau
- Notion de descripteur de fichier
- Fonctions d'ouverture et de fermeture de fichier
- Fonctions de lecture et d'écriture
- Accès direct
- Relation entre flot et descripteur de fichier

1 Présentation du langage C
- Historique
- Intérêts du langage
- Qualités attendues d'un programme

2 Généralités

3 Les déclarations

4 Expressions et opérateurs

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

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 1989 avec la parution par la suite du manuel : « **The C programming Language - 2ème édition** ».

L'année suivante, la norme américaine est adoptée par l'ISO (*International Organization for Standardization*) et l'IEC (*International Electrotechnical Commission*). Elle devient donc norme internationale sous l'appellation ISO/IEC-9899.

Une seconde version de la norme du langage C paraît en 1999 venant combler un certain nombre de lacunes, avec notamment l'adjonction de nouveaux types, l'extension du mécanisme de déclarations ...
Cette nouvelle norme porte le nom d'ISO/IEC-9899:1999.

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 ! ! !

Qualités attendues d'un programme

- Clarté ;
- Simplicité ;
- Modularité ;
- Extensibilité.

1 Présentation du langage C

2 Généralités
- Jeu de caractères utilisés
- Identificateurs et mots-clés
- Structure d'un programme C
- Compilation et édition des liens

3 Les déclarations

4 Expressions et opérateurs

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

Jeu de caractères

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

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

- séquences d'échappement telles que :
  - passage à la ligne ⇒ « `\n` » ;
  - tabulation ⇒ « `\t` » ;
  - backspace ⇒ « `\b` ».

Identificateurs et mots-clés

Les diverses composantes d'un programme telles que les variables et les fonctions se définissent au moyen d'identificateurs lesquels respectent l'écriture suivante :

- sont formés de lettres et de chiffres ainsi que du caractère « _ » permettant une plus grande lisibilité. Le 1[er]caractère doit obligatoirement être une lettre ou bien le caractère « _ » ;
- peuvent contenir jusqu'à 31 caractères minuscules et majuscules ;
- il est d'usage de réserver les identificateurs entièrement en majuscules aux variables du préprocesseur.

- 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-clés du langage

Les **mots-clés** définis par le langage sont réservés : ils ne peuvent en aucun cas servir à nommer un identificateur.
En voici la liste :

- 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	

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 le caractère « **;** » ;
- commentaire :
  - encadré par les délimiteurs « /* » et « */ » ;
  - ligne précédée des caractères « // » ;
- Instruction préprocesseur ⇒ commence par le caractère « **#** » en première colonne.

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);
}
```


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 :

- appel au pré-processeur (cpp) ;
- appel au compilateur ;
- 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` ».

1 Présentation du langage C

2 Généralités

3 Les déclarations
- Les types de base
- Les énumérations de constantes
- Les pointeurs
- Forme générale d'une déclaration
- Constructeurs homogènes
- Constructeurs hétérogènes
- Définitions de types

4 Expressions et opérateurs

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Les types de base

Le langage contient des types de base qui sont :

- les `entiers` ;
- les `réels` simple et double précision ;
- les `complexes` simple et double précision ;
- les `logiques` ;
- les `caractères` ;
- l'`ensemble vide`.

Ces différents types de base sont identifiés à l'aide des mots-clés `int`, `float`, `double`, `_Complex float`, `_Complex double`, `_Bool`, `char` et `void`.

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

Table 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
`long long int`	entier étendu
`float`	réel simple précision
`double`	réel double précision
`long double`	réel précision étendue
`_Complex float`	complexe simple précision
`_Complex double`	complexe double précision
`_Complex long double`	complexe précision étendue
`_Bool`	logique

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 2^{32}-1, et `long` une quantité comprise entre -2^{31} et 2^{31}-1.

Exemple

```
#include <stdio.h>
unsigned char mask;
long          val;
int main()
{
  unsigned int indice;
  float        x;
  double       y;
  char         c;
  _Bool        flag;
        ...
  return 0;
}
double f(double x)
{
  unsigned short  taille;
  int             i;
  unsigned long   temps;
  double          y;
  _Complex double c;
       ...
  return y;
}
```

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.

Exemple

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

Les valeurs générées par le compilateur seront :

1. noir ⇒ 0, bleu ⇒ 1, vert ⇒ 2, rouge ⇒ 3, blanc ⇒ 4, jaune ⇒ 5
2. noir ⇒ -1, bleu ⇒ 0, vert ⇒ 1, rouge ⇒ 5, blanc ⇒ 6, jaune ⇒ 7

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 !

Forme générale d'une déclaration

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

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

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**.

Table 2: *Symboles associés aux constructeurs homogènes*

Symbole	Objet construit
*	pointeur
[ ]	vecteur
( )	fonction

Exemple

```
char     lignes[100]; // vecteur de 100 caractères
int     *p_entier;    // pointeur d'entier
double   fonc();      // 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.

On a vu que le constructeur « [] » permet de se définir des vecteurs. Le nombre d'éléments de celui-ci étant précisé entre les crochets.
Avant la norme C99 ce nombre devait être une constante littérale. Depuis cette nouvelle norme, on peut se définir des VLA (*Variable Length Array*) dont le nombre d'éléments peut être précisé à l'aide d'une variable comme le montre l'exemple suivant :

Exemple

```
int main()
{
  int n, p, q;

  // Valorisation des variables n, p, q
     ...
  // Déclaration des VLA A, B et C :
  int A[n][p], B[p][q], C[n][q];

  return 0;
}
```

Constructeurs hétérogènes : les structures

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

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

Les **structures** permettent de regrouper des objets dont les types peuvent être différents. La syntaxe générale est la suivante :

```
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.

Constructeurs hétérogènes : les structures

- 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;

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;
```

Constructeurs hétérogènes : les champ de bits

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. Ils doivent être du type `unsigned int` et indiquer le nombre de bits de chaque champ :

```
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;
```

Remarques :

- 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.

Constructeurs hétérogènes : les unions

Le constructeur `union` permet de définir des données de type différent ayant la même adresse mémoire :

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

Remarques :

- à 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.

Exemple

```
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;
```

Exemple

```
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;
```

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 :

`typedef < déclaration >`

Exemple

```
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.

Exemple

```
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]
```

1 Présentation du langage C

2 Généralités

3 Les déclarations

4 Expressions et opérateurs
- Constantes littérales
  - Constantes entières
  - Constantes réelles
  - Constantes caractères
  - Constantes chaîne de caractères
  - Constantes agrégats
  - Constantes symboliques
- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateurs de taille
- Opérateurs d'adressage et d'indirection
- Opérateurs de forçage de type
- Opérateurs de manipulations de bits
  - Opérateurs arithmétiques "bit à bit"
  - Opérateurs de décalage
- Opérateurs d'affectation
  - Opérateurs à effet de bord
  - Opérateurs d'incrémentation et de décrémentation

- Opérateurs conditionnel
- Opérateurs séquentiel
- Opérateurs d'indexation
- Opérateurs d'appel de fonction
- Opérateurs de sélection de champ

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

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` ».
Depuis la norme C99 qui a défini le type `long long`, le suffixe « `ll` » ou « `LL` » permet l'écriture d'une constante entière de ce nouveau type.

Exemple

- en base 10 :
```
  22    56    1789    32765
 22u   56L   29ULL       1L
```
- en base 8 :
```
  0643     0177     0644     0755
 0177L   0222UL    0777u   0766ul
```
- en 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` ».

Exemple

```
          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.

Séquences d'échappement

Table 3: *Séquences d'échappement*

Type	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

Constantes caractères

Exemples de caractères avec valeurs entières associées

```
'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'`).

Sa valeur 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 le caractère « \ » ;
- soit la découper en plusieurs constantes chaîne de caractères, le compilateur effectuera automatiquement la concaténation.

Exemple

```
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";
```

Constantes agrégats

On peut construire des objets anonymes de type tableau ou structure. Ces constructions se caractérisent par une liste d'initialisations préfixée par un type entre parenthèses.

Exemple

```
#include <stdio.h>

struct data
{
  int    i;
  double r;
};
void f( struct data d ) { ... }
void g( struct data *p ) { ... }
int main()
{
  int *p = (int []){ 1, 2 };
  (int []){ 18, 100 } = ... // Erreur
  // passage d'une constante agrégat à une fonction :
  f( (struct data){ 1756, 3.1415926536 } );
  // passage de l'adresse d'une telle constante :
  g( &(struct data){ 1756, 2.7182818285 } );
  g( &(struct data){ .r=2.7182818285, .i=1756 } );
}
```


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`.

Constantes symboliques

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.

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

Exemple

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

Opérateurs logiques

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**. Afin de permettre une programmation plus rigoureuse, la norme `C99` a définit les types `_Bool` et `bool`.

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**.

Opérateurs logiques

Exemple

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

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

Expression	Valeur
`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)

Opérateurs de taille

L'opérateur sizeof renvoie la taille en octets de son opérande. L'opérande est soit une expression soit une expression de type :

```
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** : `p + i ≡ p + i*sizeof(t)`

Exemple

```
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.
```


Opérateurs d'adressage et d'indirection

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.

Exemple

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

Exemple (suite)

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

Opérateurs de forçage de type

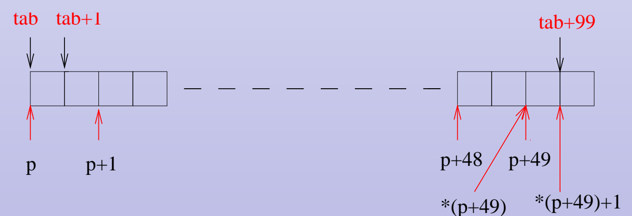
Il est possible d'effectuer des **conversions explicites** à l'aide de l'opérateur de **forçage de type** ou *cast* :

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

Exemple

```c
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;
```

Figure 1: *Schéma d'adressage*



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

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Opérateurs arithmétiques "bit à bit"

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

- « **non** : ~ », « **et** : & », « **ou** : | », « **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

Exemple

```c
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, oct, ent, 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;
```


Opérateurs d'affectation

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

On a l'équivalence suivante : `e1 op= e2` ≡ `e1 = e1 op e2`

Exemple

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

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;
```

Exemple (suite)

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

Opérateurs conditionnel

L'opérateur conditionnel « ?: » est un opérateur **ternaire**. Ses opérandes sont des expressions :

```
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;
```

Opérateurs séquentiel

L'opérateur séquentiel « , » permet de regrouper des sous-expressions sous forme de liste.

- ces sous-expressions sont évaluées en séquence ;
- cette liste d'expressions a pour valeur celle de la dernière sous-expression.

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);
```

Opérateurs 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];
```

Exemple (suite)

```
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];
```

Opérateurs 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);
// un appel valide :
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 1er opérande doit être une **structure** ou une **union** et le 2e 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 par l'intermédiaire d'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;
```

Exemple (suite)

```
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;
```

1. Présentation du langage C
2. Généralités
3. Les déclarations
4. Expressions et opérateurs
5. Portée et visibilité
   - Niveau d'une variable
   - Durée de vie d'une variable
   - Classes de mémorisation
   - Portée des variables internes
   - Portée des variables externes
     - Programme monofichier
     - Programme multifichiers
   - Initialisation des variables
   - Visibilité des fonctions
6. Instructions
7. Préprocesseur

8. Les fonctions
9. La bibliothèque standard
10. les entrées-sorties de bas niveau

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
int main()
{
    int   i, j;           // niveau 1
    char  c;              // niveau 1
    {
        Complexe c1, *c2; // niveau 2
        int       i;      // niveau 2
        if( ... )
        {
          char car;       // niveau 3
            ...
        }
    }
      ...
}
```

Exemple (suite)

```
int ent;    // niveau 0
void f(void)
{
    long i; // niveau 1
      ...
}
```

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.

Remarque :

- Un **VLA** (*Variable Length Array*), par définition, ne peut être une variable statique, mais sera forcément une variable temporaire.

Classes de mémorisation

Il existe quatre **classes de mémorisation** :

- `static` ;
- `extern` ;
- `auto` ;
- `register`.

Remarques :

- 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.

Exemple

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

Portée des variables internes. Attribut `register`

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.
Dans une boucle de type `for`, il est possible de déclarer la ou les variables intervenant au niveau de l'expression dite d'initialisation. De ce fait, celles-ci ont une portée limitée à la boucle : `for (int i=0; i<n; i++)`.

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++;
  }
}
```

Portée des variables externes : `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 ici.
  {
     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.
       ...
  }
    ...
}
```

Portée des variables externes : programme monofichier

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 1

```
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 2

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

Portée des variables externes : 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**.

Portée des variables externes : programme multifichiers

Exemple : source1.c

```c
int main()
{
    ...
}

// déclaration de la
// variable externe i.
extern int i;

void f(int a)
{
    ...
}
```

Exemple : source2.c

```c
float h(void)
{
    ...
}

// déclaration de la
// variable externe i.
extern int i;

void func(void)
{
    ...
}
```

Exemple : source3.c

```c
// définition de la variable externe i.
int i = 11;

int g(void)
{
    ...
}
```

Portée des variables externes : programme multifichiers

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

```c
int x = 10;
extern int y;
int z;
int a;
int b = 20;
int c;

main()
{
    ...
}
```

Exemple : sourceB

```c
int x;
extern int y;
extern int z;

int b = 21;
int c = 30;

int calcul(void)
{
    ...
}
```

Exemple : sourceC

```c
extern int x;
extern int y;
extern int z;
int a;


int somme(void)
{
    ...
}
```

Portée des variables externes : `programme multifichiers`

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

```c
static float r = 2.154;
double dble = 17.89;

int main()
{
   ...
}
float f1(void)
{
   ...
}
```

Exemple : sourceB

```c
void f2(void)
{
   ...
}

extern double dble;

int f3(int i)
{
   ...
}

static int n = 10;

void f4(float r)
{
   ...
}
```

Initialisation des variables

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

```
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

```c
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.

Exemple 1

```
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 }}
                 };
```


Exemple 2

```
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}
                    }
                  };
```

Il est possible de spécifier la valeur initiale de certains éléments d'un vecteur ou d'une structure en utilisant des **désignateurs** prenant la forme d'un indice dans le cas d'un vecteur et d'un champ dans le cas d'une structure :

```
[expression-constante]=...,
.nom_de_champ=...
```

Exemple

```c
#include <stdio.h>
struct data {
  int a; int b; double c[100];
};
struct cel {
  ...; struct data d; ...;
};
union U {
  int i; float r;
};
int f(void);
int main()
{
  struct data donnee_1   = {.b=1791, .a=1756};
  float       vec[100]   = {[20]=3.14, [33]=2.718};
  struct data donnee_2   = {.b=1791, .a=1756, .c[11]=3.1415926536, .c[96]=2.7182818285};
  struct data donnees[]  = {[5].a=13, [7].c[99]=2.7182818285};
  struct cel  c          = {.d.c[47]=3.1415926536};
  struct data donnee_3   = {.b=f()};
  union U     u          = {.r=3.14f};
       ...
}
```


Visibilité des fonctions

La **définition** d'une fonction comprend un en-tête (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.

Si le compilateur n'a pas cette **visibilité**, il suppose que la valeur retournée par la fonction est de type `int`. Dans un tel cas, la norme C99 impose l'émission d'un message d'avertissement.

Visibilité des fonctions

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();
    ...
}
```

Exemple : sourceB

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


1. Présentation du langage C
2. Généralités
3. Les déclarations
4. Expressions et opérateurs
5. Portée et visibilité
6. Instructions
   - Instructions élémentaires
   - Structures de contrôle
     - Les tests
     - les boucles tant-que
     - la boucle for
     - L'aiguillage
   - Instructions d'échappement
7. Préprocesseur
8. Les fonctions

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

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>
int 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 );
  }
}
```

Structures de contrôle : les tests

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

- 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 **vraie** (c-a-d non nulle). Sinon, si le test comporte une **partie-sinon** c'est celle-ci qui sera exécutée.

Exemple

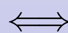
```
char buffer[2048];
void f(void)
{
  static char *p = (char *)0;

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

Structures de contrôle : les tests

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 )
   ecrire( "positif" );
else if( x < 0 )
   ecrire( "négatif" );
else
   ecrire( "nul" );
```



Exemple

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

Structures de contrôle : les boucles « tant-que »

- Les boucles « tant-que » :

```
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>
int main()
{
  int chiffre = 0;

  printf("Boucle \"while\"\n\n");
  while( chiffre )
  {
    printf( " %d", chiffre++ );
    if(! (chiffre%5)) printf("\n");
  }
```

Exemple (suite)

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


Structures de contrôle : la boucle « pour »

- La boucle « pour » :

```
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

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

  for( int 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( int i=0; i<4; i++ )
    for( int j=0; j<3; j++ )
      t[i][j] = i + j;
}
```

Structures de contrôle : L'aiguillage

- 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 :

```
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.

Structures de contrôle : L'aiguillage

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>
int main()
{
  char *buffer = "\nCeci est une chaîne\n de caractères\t"
                 "sur\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 );
}
```

Instructions d'échappement : instruction continue;

Les instructions d'échappement permettent de rompre le déroulement séquentiel d'une suite d'instructions.
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>
int main()
{
  char *buffer = "\nCeci est une chaîne\n de caractères\t"
                 "sur\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 );
}
```


Instructions d'échappement : instruction break;

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

Exemple

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

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

Instructions d'échappement : instruction return;

```
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

```c
#include <stdio.h>
int main()
{
  char c;
  char majus( char c );
     ...
  printf( "%c\n", majus(c) );
  return 0;
}
char majus( char c )
{
  return c >= 'a' && c <= 'z' ? c + 'A' - 'a' : c;
}
```

Instructions d'échappement : instruction go to;

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

```
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

```c
#include <stdio.h>
int 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 );
}
```

Instructions d'échappement : fonction exit

Un programme peut être interrompu au moyen de la fonction `exit` :

```
exit(expression);
```

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

Exemple

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

  for( int i=0; i<sizeof tab / sizeof tab[0]; i++ )
    for( int j=0; j<4; j++ )
      if( tab[i][j] == 10 )
      {
        printf( "L'élément tab[%d][%d] est égal à 10.\n", i, j );
        return 0;
      }
  fprintf( stderr, "Elément non trouvé.\n" );
  exit(1);
}
```


1. Présentation du langage C
2. Généralités
3. Les déclarations
4. Expressions et opérateurs
5. Portée et visibilité
6. Instructions
7. Préprocesseur
   - Introduction
   - Pseudo-constantes
   - Pseudo-fonctions
   - Inclusion de fichiers
   - Compilation conditionnelle
8. Les fonctions
9. La bibliothèque standard

10 les entrées-sorties de bas niveau

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 coder sur plusieurs lignes, chaque ligne à compléter étant terminée par les caractères « \ » suivi de *return*.

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 « _ » étant 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)
int main()
{
  int tab[TAILLE];

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

  return 0;
}
```

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 ;
- `__func__` : nom de la fonction courante ;
- `__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.

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>

int main()
{
  int    tab[][2] = { 1,  2,  3,  9, 10, 11, 13, 16};
  double r = -acos(-1.);

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

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

  return 0;
}
```

Pseudo-fonctions : 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 1

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

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

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

  return 0;
}
```

Exemple 2

```
#define CARRE(x) (x)*(x)
#define MAX(a,b) \
        ( (a) > (b) ? (a) : (b) )
int main()
{
  float x = 3.1, y = 4.15;
  printf("%f\n", CARRE(x+1));
  printf("%f\n", MAX(x+10., y));

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

  y = CARRE(x++);

  return 0;
}
```

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 ».

```
#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 ».

Inclusion de fichiers

Exemple : def.h

```
#include <stdbool.h>
#define  NbElements(t)  sizeof t / sizeof t[0]
#define  TAILLE 256
typedef struct cellule
{
 int               tab[TAILLE];
 struct cellule *ptr;
} Cel;
extern void init( int t[], bool imp );
Cel c;
```

Exemple : sourceA

```
#include <stdbool.h>
#include "def.h"
int main()
{
  int t[TAILLE] = {1, 2, 9, 10};
  bool imp = true;

  init( t, imp );
  return 0;
}
```

Exemple : sourceB

```
#include <stdbool.h>
#include <stdio.h>
#include "def.h"
void init( int t[], bool imp )
{
  for(int 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;
  return 0;
}
```

Inclusion de fichiers

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) ;
- `stdbool.h` (accès au type `bool` ainsi qu'aux constantes `true` et `false`) ;
- `complex.h` (fonctions mathématiques à arguments complexes) ;
- `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).

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** :

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

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

Compilation conditionnelle : test d'existence d'une pseudo-constante

Exemple 1

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

Exemple 2 : def.h

```
#ifdef DEBUG
#    define trace(s) \
             printf s
#else
#    define trace(s)
#endif/* DEBUG */
```

Exemple 2 (suite)

```
#define DEBUG
#include "def.h"
#include <stdio.h>
int main()
{
  int f(float x);
  int i;
  float r;

  i = f(r);
  trace(("%d\n", i));
  return 0;
}
```

Compilation conditionnelle : test d'existence d'une pseudo-constante

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` ».

```
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
```

Compilation conditionnelle : 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`.

au moyen de la structure suivante :

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

Remarques :

- 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 */`.

Préprocesseur | Compilation conditionnelle

Compilation conditionnelle : évaluation de pseudo-expressions

Exemple

```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];

int main()
{
  printf( "Taille du vecteur : %d caractères\n", sizeof buffer );

  return 0;
}
```

1. `cc -DTAILLE=255 source.c`
2. `cc -DTAILLE=257 source.c`
3. `cc source.c`

1 Présentation du langage C

2 Généralités

3 Les déclarations

4 Expressions et opérateurs

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions
- Passage arguments-paramètres
- Fonction retournant un pointeur
- Passage d'un vecteur comme argument
- Passage d'une structure comme argument
- Passage d'une fonction comme argument
- fonction inline
- Passage d'arguments à la fonction main
- Fonction avec un nombre variable d'arguments

9 La bibliothèque standard

10 les entrées-sorties de bas niveau

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

- par **référence** ;
- par **adresse** ;
- par **valeur**.

Un langage comme **Fortran** a choisi la 1re solution, tandis qu'un langage comme **Pascal** offre les deux dernières possibilités au programmeur. Le langage **C** a choisi la dernière 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>
int 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 0;
}
void somme(int a, int b, int *c)
{
  *c = a + b;
  return;
}
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

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>
int main()
{
  char *p;
  char *ini_car( void );
  p = ini_car();
  printf( "%c\n", *p );

  return 0;
}

char *ini_car( void )
{
  char c;
  c = '#';

  return(&c); // ERREUR
}
```

Passage d'un vecteur comme argument

Un vecteur est une constante symbolique dont la valeur est l'adresse de son 1er élément. Lorsqu'un vecteur est passé en argument, c'est donc l'adresse de son 1er élément qui est transmise par valeur.

Exemple 1

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

Passage d'un vecteur comme argument

Exemple 2

```
#define NbElements(t) sizeof t / sizeof t[0]
#include <stdio.h>
int 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)));

  return 0;
}

int somme(int (*t)[5], int n)
{
  int som = 0;
  int (*p)[5] = t;

  for( ; t-p<n; t++)
    for (int i=0; i<5; i++)
      som += (*t)[i];
  return som;
}
```

Passage d'un vecteur comme argument

Exemple 2 : même exemple avec tab transmis sous forme de VLA

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

  printf("Somme des éléments de tab : %d\n",
             somme(NbElements(tab), M, tab));

  return 0;
}
int somme(int n, int m, int t[n][m])
{
  int som;

  som = 0;
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
      som += t[i][j];

  return som;
}
```


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];
} Data;
int main()
{
  Data data = { {1.34f, 8.78f, 10.f, 4.f, 22.12f, 3.145f} }, inv;
  Data inverse( Data data, int n );
  int  n = NbElts(data.v);

  inv = inverse( data, n );
  for (int i=0; i<n; i++) printf( "inv.v[%d] : %f\n", i, inv.v[i] );
  return 0;
}
Data inverse( Data data, int n )
{
  Data inv_data;

  for (int i=0; i<n; i++) inv_data.v[i] = data.v[i] ? 1./data.v[i] : 0.f;
  return inv_data;
}
```

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;
  h = (b_sup - b_inf)/pas;
  for (int i=0; i<pas; i++)
    surface += h*(*f)( b_inf+i*h );
  return surface;
}
double carre( double x ) {return x*x;}
```


fonction inline

Le mécanisme de fonction `inline` déjà existant en `C++` a été introduit dans la norme `C99`. C'est par l'intermédiaire du mot-clé `inline`, précisé lors de l'écriture du prototype de la fonction (déclaration, définition), que l'on active ce mécanisme.

Exemple

```
#include <stdio.h>
int main()
{
   int a, b, c;
   inline 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 0;
}
inline void somme( int a, int b, int *c )
{
   *c = a + b;

   return;
}
```

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 :

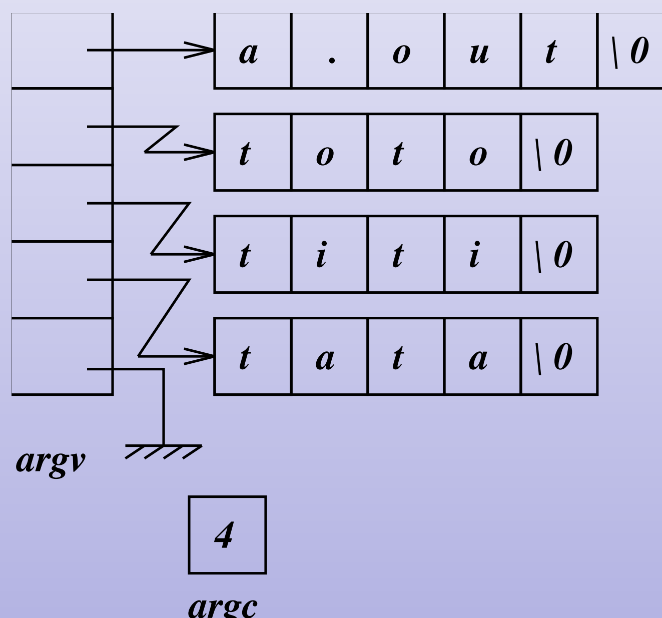
- `argc` désigne le nombre d'arguments transmis au moment du lancement de l'exécutable ;
- `argv` désigne le vecteur contenant les différents arguments ;
- `envp` désigne le vecteur contenant les informations sur l'environnement.

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

Passage d'arguments à la fonction main

Une commande telle que « `a.out toto titi tata` » génère la structure de données suivante :

Figure 2: *Passage d'arguments à la fonction main*



Passage d'arguments à la fonction main

Exemple

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

int 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 );

  return 0;
}

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

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

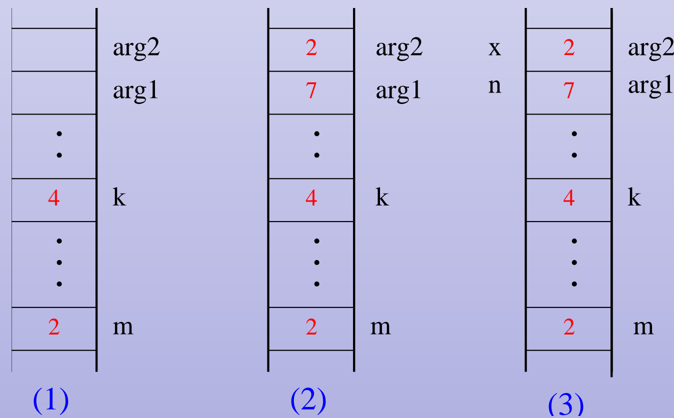
```
int puissance(int n, int x)
{
  int p = 1;

  while(n--) p *= x;

  return p;
}

int main()
{
  int m, k, r;

  k = 4; m = 2;
  r = puissance(k+3, m);

  return 0;
}
```

Fonction avec un nombre variable d'arguments

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).

Figure 3: *Processus de passage d'arguments*



Fonction avec un nombre variable 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) );
}
```

Fonction avec un nombre variable d'arguments

- 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.

Fonction avec un nombre variable d'arguments : 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 les caractères « `...` » dans le **prototype** de la fonction :

```c
#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;
  va_list arg;
  va_start(arg, nombre);
  for( int i=0; i<nombre; i++)
    somme += va_arg(arg, int);
  va_end(arg);
  return (float)somme/nombre;
}
```

Fonction avec un nombre variable d'arguments : 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 1er argument 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.

Fonction avec un nombre variable d'arguments : version Unix System V

Exemple : transmission d'entiers

```c
#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 (float)somme/nombre;
}
```

Exemple : transmission de réels

```c
#include <stdio.h>
#include <varargs.h>
int 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++)
    // double : surtout pas float !
    somme += va_arg(arg, double);
  va_end(arg);
  return somme/nombre;
}
```

1 Présentation du langage C

2 Généralités

3 Les déclarations

4 Expressions et opérateurs

5 Portée et visibilité

6 Instructions

7 Préprocesseur

8 Les fonctions

9 La bibliothèque standard
- Notion de pointeur générique
- Entrées-sorties de haut niveau
- Fonctions d'ouverture et de fermeture
- Lecture et écriture de caractères
- Lecture et écriture de mots

- Lecture et écriture de chaînes de caractères
- Lecture et écriture de blocs
- Accès direct
- Entrées-sorties formatées
- Autres fonctions
- Manipulation de caractères
- Fonctions de conversions
- Manipulation de chaînes de caractères
- Manipulation de tableaux d'octets
- Allocation dynamique de mémoire
- Date et heure courantes
- Accès à l'environnement
- Sauvegarde et restauration du contexte
- Aide à la mise au point de programme
- Récupération des erreurs
- Fonctions mathématiques
- Fonctions de recherche et de tri

10 les entrées-sorties de bas niveau

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 , d'auto-incrémentation et d'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.

Exemple

```
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
```

Notion de pointeur générique

Exemple

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

   r           = *(void **)p;
  *(void **)p = *(void **)q;
  *(void **)q =  r;
}

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

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, ...

Entrées-sorties de haut niveau

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/SP6

```
#define _NIOBRW          16
extern FILE      _iob[_NIOBRW];

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

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 libérer :

```
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 ;
- le 2e argument de la fonction `fopen` indique le mode d'ouverture du fichier ;
- 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.

Mode d'ouverture d'un 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

Fonctions d'ouverture et de fermeture

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

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

Lecture et écriture de caractères

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** :.

```
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).

Lecture et écriture par caractère

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 erroné

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

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

Exemple correct

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

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

Exemple avec ferror et feof

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

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

Lecture et écriture de mots

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

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

Exemple

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

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

Lecture et écriture de chaînes de caractères

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

```
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.

Lecture et écriture de chaînes de caractères

- puts écrit sur le flot stdout la chaîne passée en argument suivie d'un *retour chariot* ;
- fgets lit sur le *flot* fourni en 3e 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 1er argument sur le flot spécifié en 2e 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.

Lecture et écriture de chaînes de caractères

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int 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." );
}
```

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** :

```
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.

Lecture et écriture de blocs

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#define NbElt(t) ( sizeof t / sizeof t[0] )
int 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);
}
```


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).

Accès direct

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int 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 de mise à jour de fichier

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>
#define NbElements(t) sizeof t / sizeof t[0]

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

Exemple de mise à jour de fichier (suite)

```c
  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( int n=0; n<NbElements(mus); n++ )
    fputs( mus[n], f );
  fclose( f );
}
void conversion( char *nom, char *mode )
{
  int (*conv)(int);

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


Entrées-sorties formatées

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

```c
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 ( const char *buffer, const char *format [, ...] )
int snprintf( const char *buffer, size_t n, const char *format [, ...] )
```

Entrées-sorties formatées : fonctions printf, fprintf, sprintf, snprintf

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, snprintf.

La fonction snprintf admet un argument supplémentaire indiquant le nombre maximum de caractères à transférer permettant ainsi d'éviter les débordements en mémoire. Les **spécifications de conversions** portent successivement sur les arguments passés à la suite du paramètre **format**.

La bibliothèque standard Entrées-sorties formatées

Entrées-sorties formatées : fonctions printf, fprintf, sprintf, snprintf

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.

Entrées-sorties formatées : fonctions `printf`, `fprintf`, `sprintf`, `snprintf`

- 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.

  Pour un entier, s'il s'agit d'un `long` ou d'un « `long long` » on précisera « `l` », « `ll` » comme préfixe respectivement (« `lu` », « `lld` » . . .).

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|
```

Exemples

```
printf( "|%f|\n",     1.234567890123456789e5 );     |123456.789012|
printf( "|%.4f|\n",   1.234567890123456789e5 );     |123456.7890|
printf( "|%.15f|\n",  1.234567890123456789e5 );     |123456.789012345670000|
printf( "|%15.4f|\n",1.234567890123456789e5 );      |    123456.7890|
printf( "|%e|\n",     1.234567890123456789e5 );     |1.234568e+05|
printf( "|%.4e|\n",   1.234567890123456789e5 );     |1.2346e+05|
printf( "|%.18e|\n",  1.234567890123456789e5 );     |1.234567890123456700e+05|
printf( "|%18.4e|\n",1.234567890123456789e5 );      |        1.2346e+05|
printf( "|%.4g|\n",   1.234567890123456789e-5 );    |1.235e-05|
printf( "|%.4g|\n",   1.234567890123456789e+5 );    |1.235e+05|
printf( "|%.4g|\n",   1.234567890123456789e-3 );    |0.001235|
printf( "|%.8g|\n",   1.234567890123456789e5 );     |123456.79|
```

Exemples

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

  printf( "|%s|\n",        chaine );        |Wolfgang Amadeus Mozart|
  printf( "|%.16s|\n",     chaine );        |Wolfgang Amadeus|
  printf( "|%-23.16s|\n",  chaine );        |Wolfgang Amadeus       |
  printf( "|%23.16s|\n",   chaine );        |       Wolfgang Amadeus|
}
```

Exemple d'utilisation de snprintf

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
  char buffer[1024];
  int  n, taille_dispo;

  taille_dispo = sizeof buffer;
  n = snprintf( buffer, taille_dispo, "Commande lancée ==> " );
  for( int i=0; i<argc; i++ )
  {
    taille_dispo = sizeof buffer - n;
    n += snprintf( buffer+n, taille_dispo, " %s", argv[i] );
  }
  printf( "%s\n", buffer );

  return 0;
}
```

Entrées-sorties formatées : 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.

Entrées-sorties formatées : fonctions scanf, fscanf, sscanf

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.

Entrées-sorties formatées : fonctions scanf, fscanf, sscanf

- [...] 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.

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, l ou ll pour référencer respectivement un « short * », un « long * » ou bien un « long long * ». De même les spécifications e, f, g peuvent être précédées de la lettre l pour référencer un « double * ».

Entrées-sorties formatées : fonctions scanf, fscanf, sscanf

Exemple

```
#include <stdio.h>
int 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 );
}
```


Entrées-sorties formatées : fonctions scanf, fscanf, sscanf

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int 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);
}
```

Autres fonctions : freopen, fflush

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 :

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

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int 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 dans"
          " le fichier dont le nom est" );
  printf( " passé en argument.\n" );
}
```

Exemple (suite)

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

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.

Manipulation de caractères

Exemple

```
#include <stdio.h>
#include <ctype.h>
int main()
{
  int c;
  int NbMaj, NbMin, NbNum, NbAutres;

  NbMaj = NbMin = NbNum = 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 );
}
```


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`.

Manipulation de caractères

Exemple

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

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`.

Manipulation de chaînes de caractères

- 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.

Manipulation de chaînes de caractères

Exemple 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int 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);
}
```

Manipulation de chaînes de caractères

Exemple 2

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


Manipulation de tableaux d'octets

Il existe d'autres fonctions qui agissent sur des tableaux d'octets 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`.

Manipulation de tableaux d'octets

Exemple

```
#include <stdio.h>
#include <string.h>
int 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 );
}
```


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** :

```
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)
```

Allocation dynamique de mémoire

Exemple 1

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


Allocation dynamique de mémoire

Exemple 2

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
  char **ptr = (char **)NULL;
  char   buffer[BUFSIZ];
  for( int nb=0;; )
  {
    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( int i=0; i<nb; i++ )
  {
    printf( "%s\n", ptr[i] );
    free( ptr[i] );
  }
  free(ptr);
}
```

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))`.

Date et heure courantes

Remarques :

- 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.

Date et heure courantes

Exemple

```
#include <stdio.h>
#include <time.h>
int 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
}
```

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 :

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

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#define RACINE        "RACINE"
#define DEFAUT_RACINE "."
int main()
{
  char *racine;

  if( (racine = getenv( RACINE )) == NULL )
    racine = DEFAUT_RACINE;
  printf( "Répertoire de travail: \"%s\"\n", racine );

  return 0;
}
```

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**.

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

La bibliothèque standard | Sauvegarde et restauration du contexte

Sauvegarde et restauration du contexte

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 2e argument, à la fonction `longjmp`.

Exemple

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf cntx;
int main()
{
  int appel_boucle( void );
  printf( "Terminaison avec \"%c\"\n", appel_boucle() );
}
int appel_boucle( void )
{
  void boucle( void );
  int  retour = setjmp( cntx );
  printf( "setjmp retourne %d\n", retour );
  if( retour == 0 ) boucle();
  return retour;
}
```

Sauvegarde et restauration du contexte

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;
}
```


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, le nom de la fonction 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
int main()
{
  char tab[DIM];
  void f( char *p, int n );

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

Exemple (suite)

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

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

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.

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

Récupération des erreurs

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

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

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

- `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** :

```
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>
int main()
{
  double x, y;

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

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;
}
```


Fonctions de recherche

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

```
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 1er est un pointeur sur l'élément à rechercher ;
- le 2e un pointeur sur un élément du vecteur ;
- cette fonction doit retourner un entier négatif, nul ou positif suivant que son 1er argument est inférieur, égal ou supérieur à son 2e argument (en terme de rang dans le vecteur).

Fonctions de recherche

Exemple

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int 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;
}
```


Fonctions de tri

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)` :

```
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`.

Fonctions de tri

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int 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;
}
```


1 Présentation du langage C
2 Généralités
3 Les déclarations
4 Expressions et opérateurs
5 Portée et visibilité
6 Instructions
7 Préprocesseur
8 Les fonctions
9 La bibliothèque standard
10 les entrées-sorties de bas niveau
Notion de descripteur de fichier
Fonctions d'ouverture et de fermeture de fichier
Fonctions de lecture et d'écriture

Accès direct
Relation entre flot et descripteur de fichier

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** ».

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 :

```
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.

les entrées-sorties de bas niveau | Fonctions d'ouverture et de fermeture de fichier

Fonctions d'ouverture et de fermeture de fichier

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 <stdlib.h>
#include <fcntl.h>
int 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;
}
```

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` :

```
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.

Fonctions de lecture et d'écriture

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
int 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;
}
```

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** :

```
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;
}
```

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**.

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

12 Annexe B : priorité des opérateurs

13 Annexe C : exercices

Table 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

Table 5: *table des codes ASCII des caractères (suite)*

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
ˆ	94	0x5e	136	∼	126	0x7e	176
_	95	0x5f	137	C-?	127	0x7f	177

11 Annexe A : table des codes ASCII des caractères

12 Annexe B : priorité des opérateurs

13 Annexe C : exercices

Table 6: *table de priorité des opérateurs*

Catégorie d'opérateurs	Opérateurs	Assoc.
`fonction, tableau,`	`() [] . ->`	G⇒D
`membre de structure,`		
`pointeur sur un`		
`membre de structure`		
`opérateurs unaires`	`- ++ -- ! ~`	D⇒G
	`* & sizeof (type)`	
`multiplication, division,`	`* / %`	G⇒D
`modulo`		
`addition, soustraction`	`- +`	G⇒D
`opérateurs binaires`	`<< >>`	G⇒D
`de décalage`		
`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

11 Annexe A : table des codes ASCII des caractères

12 Annexe B : priorité des opérateurs

13 Annexe C : exercices
Énoncés
Corrigés

Énoncés

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 :

1. `(3*i - 2*j)%(2*d - c)`
2. `2*((i/5) + (4*(j-3))%(i + j - 2))`
3. `i <=j`
4. `j != 6`
5. `c == 99`
6. `5*(i + j) > 'c'`
7. `(i > 0) && (j < 5)`
8. `(i > 0) || (j < 5)`
9. `(x > y) && (i > 0) || (j < 5)`
10. `(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 :

1. à créer une table de n logiques valorisés à **true** (C'est la valeur de l'indice dans cette table qui référence les différents entiers) ;
2. à rayer (mise à **false**) les uns après les autres, les éléments dont l'indice n'est pas premier de la manière suivante : dès que l'on trouve un élément qui n'a pas encore été rayé, son indice est déclaré premier, et on raye tous les éléments d'indice multiple de celui-ci.

À la fin du procédé, les indices du tableau qui correspondent aux éléments **true** sont les nombres premiers recherchés.
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`

- une seule des options `-a`, `-b` ou `-c` doit être spécifiée, suivie d'une chaîne ;
- les options `-d`, `-e`, `-f` sont facultatives. Si aucune d'entre elles n'est indiquée, le programme doit les considérer comme toutes présentes ;
- 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 :

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

- la liste des enregistrements du fichier indexé des musiciens ;
- cette même liste triée par ordre alphabétique des noms des musiciens ;
- cette même liste triée par ordre chronologique des musiciens ;
- 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.

Corrigé de l'exercice 1

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

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

Corrigé de l'exercice 2

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

printf( "%c\n", (*p++)[1] ); // 'o'
printf( "%c\n", *p++[1]   ); // 'H'
printf( "%c\n", (*++p)[4] ); // 'l'
printf( "%c\n", *++*p     ); // 'i'

printf( "%c\n", (*p++)[1] ); // 'o'
printf( "%c\n", *p[1]++   ); // 'H'
printf( "%c\n", (*++p)[4] ); // 'r'
printf( "%c\n", *++*p     ); // 'c'
```

Corrigé de l'exercice 3

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

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

  // 1) on commence par évaluer l'expression  « a > 0 && a <= 15 »,
  //    laquelle constitue le 1er opérande de l'opérateur « ?: ».
  //    Celle-ci est fausse car les expressions « a > 0 » et « a <= 15 »
  //    sont vraie et fausse respectivement ;
  // 2) on évalue donc le 3e opérande de l'opérateur « ?: »,
  //    c'est-à-dire l'expression « a/b » ;
  // 3) et enfin on effectue l'affectation.

  c += a > 0 && a <= 15 ? ++a : a/b;
  printf( "c = %d\n", c );            // c = 18

  // Que dire de l'expression suivante ?
  a > 30 ? b = 11 : c = 100;

  // Cette expression provoque une erreur à la compilation car le
  // 3e opérande de l'opérateur « ?: » est « c » et non pas « c = 100 ».
  // De ce fait, l'expression « a > 30 ? b = 11 : c » est d'abord évaluée.
  // Sa valeur est ensuite utilisée comme opérande de gauche de la dernière
  // affectation : d'où l'erreur, car ce n'est pas une « g-valeur ».
  // On devrait écrire :
  a > 30 ? b = 11 : (c = 100);

  return 0;
}
```


Corrigé de l'exercice 4

```c
#include <stdio.h>

int main()
{
  int pairs, carres_impairs, cubes;

  pairs = carres_impairs = cubes = 0;
  // Boucle de calcul.
  for( int i=1; i<=100; i++ )
  {
    cubes += i*i*i;
    // « i » est-il pair ou impair ?
    i%2 ? carres_impairs += i*i : (pairs += i);
  }
  // Impression des résultats.
  printf( "Somme des entiers pairs entre 1 et 100 : %d\n", pairs );
  printf( "Somme des carrés des entiers impairs entre"
          " 1 et 100 : %d\n", carres_impairs );
  printf( "Somme des cubes des 100 premiers entiers : %d\n", cubes );

  printf( "\n\nFin EX04.\n" );

  return 0;
}
```

Corrigé de l'exercice 5

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

// « drand48 » est un générateur de nombres pseudo-aléatoires uniformément distribués.
int main()
{
  const int n = 10, m = 5, p = 3;
  double    a[n][m], b[m][p], c[n][p];

  // Produit matriciel « C = A*B »
  for( int i=0; i<n; i++ )
    for( int j=0; j<p; j++ )
    {
      c[i][j] = 0.;
      for( int k=0; k<m; k++ )
      {
        a[i][k] = drand48(); b[k][j] = drand48();
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  // Impression de la matrice « C ».
  for( int i=0; i<n; i++ )
  {
    for( int j=0; j<p; j++ )
      printf( "%9.5f", c[i][j] );
    printf( "\n" );
  }

  printf( "\n\nFin EX05.\n" );

  return 0;
}
```


Corrigé de l'exercice 6

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

int main()
{
  const int n = 1000;
  bool      tab_nombres[n];
  int       imax;

  // Remplissage du tableau « tab_nombres »
  for( int i=1; i<n; i++ ) tab_nombres[i] = true;

  imax = (int)sqrt( (double)n );
  for( int i=2; i<imax; i++ )
    if( tab_nombres[i] )
      // Suppression des multiples non déjà exclus du nombre « i »
      for( int j=i+1; j<n; j++ )
        if( tab_nombres[j] && j%i == 0 )
          tab_nombres[j] = false;
  // Impression des nombres non exclus qui sont les nombres
  // premiers cherchés à raison de 10 nombres par ligne.
  printf( "Les nombres premiers entre 1 et %d sont :\n\n", n );
  for( int i=1; i<n; i++ ) {
    static int nb_prem = 0;
    if ( tab_nombres[i] ) {
      if( nb_prem++%10 == 0 ) printf( "\n" );
      printf( "%5d", i );
    }
  }

  printf( "\n\nFin EX06.\n" );

  return 0;
}
```

Corrigé de l'exercice 7

```
#include <stdio.h>

int main()
{
  char tab[12][12];

  // Boucle sur les colonnes.
  for( int j=0; j<12; )
    // On remplit par groupe de 3 colonnes avec les
    // caractères successifs '1', '2'  et '3'.
    for( char c='1'; c<='3'; c++, j++ )
      for( int i=j; i<12; i++ )
        tab[i][j] = c;

  // Impression du tableau obtenu.
  for( int i=0; i<12; i++ )
  {
    for( int j=0; j<=i; j++ )
      printf( "  %c", tab[i][j] );
    printf( "\n" );
  }

  printf( "\n\nFin EXO7.\n" );

  return 0;
}
```

Corrigé de l'exercice 8

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

#define NLIGNES 3
#define NCOLS   4

int main()
{
  void      tri_vec( double *t );
  const int n = NLIGNES, m = NCOLS;
  double    mat[n][m];

  // Tri de chaque vecteur ligne.
  for( int i=0; i<n; i++ )
  {
    for ( int j=0; j<m; j++ )
      mat[i][j] = drand48();
    tri_vec( mat[i] );
  }
  // Impression de la matrice obtenue.
  for( int i=0; i<n; i++ )
  {
    for( int j=0; j<m; j++ )
      printf( "%9.3f", mat[i][j] );
    printf( "\n" );
  }
  printf( "\n\nFin EXO8.\n" );
  return 0;
}
```

Corrigé de l'exercice 8 (suite)

```
// Fonction effectuant le tri d'un
// vecteur par la méthode du tri
// à « bulles ».
void tri_vec( double *t )
{
  bool tri_termine;

  for( ;; )
  {
    tri_termine = true;
    for( int i=0; i<NCOLS-1; i++ )
      if( t[i] > t[i+1] )
      {
        double temp;

        temp = t[i+1];
        t[i+1] = t[i];
        t[i] = temp;
        tri_termine = false;
      }
    if ( tri_termine ) break;
  }

  return;
}
```

Corrigé de l'exercice 9

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

// Fonctions de transformations.
double identite(double x) { return x; }
double carre   (double x) { return x*x; }
double cubes   (double x) { return x*x*x; }

int main()
{
  void init      ( int n, int m,
                   double mat[n][m] );
  void transform( int n, int m,
                   double mat[n][m],
                   double (*f)( double ) );
  const int n = 3, m = 4;
  double    mat[n][m];
  const int choix = 4;

  init( n, m, mat );
  switch( choix )
  {
    case 1:
      transform( n, m, mat, identite );
      break;
    case 2:
      transform( n, m, mat, carre );
      break;
    case 3:
      transform( n, m, mat, cubes );
      break;
    case 4:
      transform( n, m, mat, log );
      break;
  }
```

Corrigé de l'exercice 9 (suite)

```c
  // Impression de la matrice transformée.
  for( int i=0; i<n; i++ )
  {
    for( int j=0; j<m; j++ )
      printf( "%9.3f", mat[i][j] );
    printf( "\n" );
  }

  printf( "\n\nFin EX09.\n" );

  return 0;
}
// Fonction d'inialisation.
void init( int n, int m, double mat[n][m] )
{
  for( int i=0; i<n; i++ )
    for( int j=0; j<m; j++ )
      mat[i][j] = drand48();

  return;
}
// Fonction effectuant la transformation.
void transform( int n, int m,
                double mat[n][m],
                double (*f)( double ) )
{
  for( int i=0; i<n; i++ )
    for( int j=0; j<m; j++ )
      mat[i][j] = (*f)( mat[i][j] );

  return;
}
```


Corrigé de l'exercice 10

```c
#include <stdio.h>
#include <stdarg.h>

typedef struct somme
{
  int    entiers;
  double reels;
} Somme;

int main()
{
  Somme sigma( int nb_couples, ... );
  Somme s;

  s = sigma( 4, 2, 3., 3, 10., 3, 5., 11, 132. );
  printf( " Somme des entiers : %d\n", s.entiers );
  printf( " Somme des réels   : %f\n", s.reels );
  printf( "\t\t------------------\n" );

  s = sigma( 5, 2, 3., 3, 10., 3, 5., 11, 132., 121, 165. );
  printf( " Somme des entiers : %d\n", s.entiers );
  printf( " Somme des réels   : %f\n", s.reels );

  printf( "\n\nFin EX010.\n" );

  return 0;
}
```

Corrigé de l'exercice 10 (suite)

```
Somme sigma( int nb_couples, ... )
{
  Somme   s;
  va_list arg;

  // Initialisation de « arg » avec l'adresse de l'argument qui
  // suit « nb_couples ». (« arg » pointe sur l'entier du 1er couple).
  va_start( arg, nb_couples );
  s.entiers = s.reels = 0;

  // Boucle de récupération des valeurs.
  for( int i=0; i<nb_couples; i++ )
  {
    s.entiers += va_arg( arg, int );
    s.reels   += va_arg( arg, double );
  }
  va_end( arg );

  return s;
}
```

Corrigé de l'exercice 11 : 1er solution

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

void traitement_option( const unsigned char option );
void usage( char *s );

const unsigned char  option_a = 1;
const unsigned char  option_b = 2;
const unsigned char  option_c = 4;
const unsigned char  option_d = 8;
const unsigned char  option_e = 16;
const unsigned char  option_f = 32;
unsigned char        mask_options = 0;
char                *valeur_option = NULL, *fichier = NULL;
char                *module;

void bilan_options( void );
```

Corrigé de l'exercice 11 : 1er solution (suite)

```
int main( int argc, char **argv )
{
  // Sauvegarde du nom de l'exécutable.
  module = strdup( *argv );
  // Boucle sur les arguments.
  while( *++argv != NULL )
  {
    // Est-ce que l'argument commence par le caractère '-' ?
    if( (*argv)[0] == '-' )
    {
      // On analyse les caractères qui suivent.
      for( (*argv)++; **argv; (*argv)++ )
      {
        switch( **argv )
        {
          case 'a':
            traitement_option( option_a );
            break;
          case 'b':
            traitement_option( option_b );
            break;
          case 'c':
            traitement_option( option_c );
            break;
          case 'd':
            traitement_option( option_d );
            break;
          case 'e':
            traitement_option( option_e );
            break;
          case 'f':
            traitement_option( option_f );
            break;
          default : usage( module );
        }
```


Annexe C : exercices Corrigés

Corrigé de l'exercice 11 : 1er solution (suite)

```
        if ( **argv == 'a' || **argv == 'b' || **argv == 'c' )
        {
          // La valeur de l'option 'a', 'b' ou 'c' peut suivre
          // immédiatement, ou bien être séparée de l'option
          // par des blancs.
          if( *++*argv != '\0' )
            valeur_option = strdup( *argv );
          // Cas où aucune valeur ne suit.
          else if( (*++argv)[0] == '-' )
            usage( module );
          else
            valeur_option = strdup( *argv );
          break;
        }
      }
    }
    // L'argument ne commence pas par le caractère '-'.
    else if( fichier != NULL )
      usage( module );
    else
      fichier = strdup( *argv );
  }
  bilan_options();

  printf( "\n\nFin EXO11.\n" );

  return 0;
}
```

Corrigé de l'exercice 11 : 1er solution (suite)

```
void bilan_options( void )
{
  // L'option 'a', 'b', ou 'c' suivie d'une chaîne, ainsi
  // qu'un nom de fichier doivent être spécifiés.

  if( valeur_option == NULL || fichier == NULL )
    usage( module );

  // Si aucune des options 'd', 'e', 'f' n'a
  // été spécifiée on les considère toutes.

  if( ! (mask_options & option_d) && ! (mask_options & option_e) &&
      ! (mask_options & option_f) )
    mask_options |= option_d + option_e + option_f;
  if( mask_options & option_a )
    printf( "Option \"a\" fournie avec comme valeur : %s\n", valeur_option );
  if( mask_options & option_b )
    printf( "Option \"b\" fournie avec comme valeur : %s\n", valeur_option );
  if( mask_options & option_c )
    printf( "Option \"c\" fournie avec comme valeur : %s\n", valeur_option );

  printf( "Option \"d\" %s.\n", mask_options & option_d ? "active" : "inactive" );
  printf( "Option \"e\" %s.\n", mask_options & option_e ? "active" : "inactive" );
  printf( "Option \"f\" %s.\n", mask_options & option_f ? "active" : "inactive" );

  printf( "fichier indiqué : %s\n", fichier );

  return;
}
```

Corrigé de l'exercice 11 : 1er solution (suite)

```
void traitement_option( const unsigned char option )
{
  // Une seule des options "-a", "-b", "-c" doit avoir été spécifiée.
  if ( option == option_a || option == option_b || option == option_c )
    if ( valeur_option != NULL )
      usage( module );

  // On interdit qu'une option soit indiquée 2 fois.
  if ( mask_options & option )
    usage( module );
  else
    mask_options |= option;

  return;
}

void usage( char *s )
{
  const char * const message =
        "usage: %s -a chaine | -b chaine | -c chaine [-d -e -f] fichier\n";

  fprintf(stderr, message, s);

  exit(1);
}
```

Corrigé de l'exercice 11 : 2e solution

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

typedef struct options
{
  char  nom[3];
  bool  option_fournie, option_a_valeur;
  char *valeur;
} Option;
typedef enum { option_a, option_b, option_c, option_d, option_e, option_f,
               nb_options, option_invalide } option_t;

Option options[] = {
                     { "-a", false, true,  NULL },
                     { "-b", false, true,  NULL },
                     { "-c", false, true,  NULL },
                     { "-d", false, false, NULL },
                     { "-e", false, false, NULL },
                     { "-f", false, false, NULL }
                   };

option_t type_option   ( char *option );
void     bilan_options ( void );
void     usage         ( char *s );

char *module, *fichier = NULL;
```


Annexe C : exercices Corrigés

Corrigé de l'exercice 11 : 2e solution (suite)

```
int main( int argc, char **argv )
{
  // Sauvegarde du nom de l'exécutable.
  module = strdup( *argv );

  // Boucle sur les arguments.
  while( *++argv != NULL )
  {
    // Est-ce que l'argument commence par le caractère '-' ?
    if( (*argv)[0] == '-' )
    {
      option_t opt;

      opt = type_option( *argv );
      if( opt == option_invalide ) usage( module );
      if( options[opt].option_a_valeur )
      {
        *argv += strlen( options[opt].nom );
        if( argv[0][0] != '\0' )
          options[opt].valeur = strdup( *argv );
        // Cas où aucune valeur ne suit.
        else if( (++argv)[0][0] == '-' )
          usage( module );
        else
          options[opt].valeur = strdup( *argv );
      }
    }
```

Corrigé de l'exercice 11 : 2e solution (suite)

```
    else if( fichier != NULL )
      usage( module );
    else
      fichier = strdup( *argv );
  }
  bilan_options();

  printf( "\n\nFin EXO11.\n" );
  return 0;
}
option_t type_option( char *option )
{
  option_t rang;

  for( rang=0; rang<nb_options; rang++ )
    if ( ! strncmp(options[rang].nom, option, strlen( options[rang].nom )) &&
         ! options[rang].option_fournie )
      break;
  if ( rang == nb_options )
    return option_invalide;
  if ( strcmp(options[rang].nom, option) != 0 && ! options[rang].option_a_valeur )
    return option_invalide;

  options[rang].option_fournie = true;

  return rang;
}
void usage( char *s )
{
  const char * const message =
       "usage: %s -a chaine | -b chaine | -c chaine [-d -e -f] fichier\n";

  fprintf(stderr, message, s);
  exit(1);
}
```


Corrigé de l'exercice 11 : 2e solution (suite)

```
void bilan_options( void )
{
  // Une seule des options "-a", "-b", "-c" doit avoir
  // été spécifiée ainsi qu'un nom de fichier.
  if( options[option_a].option_fournie ^ options[option_b].option_fournie ^
      options[option_c].option_fournie && fichier != NULL )
  {
    if ( options[option_a].option_fournie && options[option_b].option_fournie &&
         options[option_c].option_fournie ) usage( module );
    // Si aucune des options 'd', 'e', 'f' n'a
    // été spécifiée on les considère toutes.
    if( ! options[option_d].option_fournie && ! options[option_e].option_fournie &&
        ! options[option_f].option_fournie )
      options[option_d].option_fournie = options[option_e].option_fournie
                                       = options[option_f].option_fournie = true;
    for( option_t rang=0; rang<nb_options; rang++ )
      if ( options[rang].option_fournie )
      {
        if ( options[rang].option_a_valeur )
          printf( "Option %s fournie avec comme valeur : %s\n",
                  options[rang].nom, options[rang].valeur );
        else
          printf( "Option %s active.\n", options[rang].nom );
      }
      else if ( ! options[rang].option_a_valeur )
        printf( "Option %s inactive.\n", options[rang].nom );

    printf( "fichier indiqué : %s\n", fichier );
  }
  else
    usage( module );

  return;
}
```

Corrigé de l'exercice 12

```
#include <stdio.h>

int main()
{
  char  buffer[BUFSIZ];
  char *p;

  // Boucle de lecture sur l'entrée standard
  // avec la chaîne « --> » comme prompt.
  fputs( "--> ", stdout );
  gets( buffer );
  while( ! feof(stdin) )
  {
    // On se positionne à la fin du mot lu.
    for( p=buffer; *p; p++ ); // corps de boucle vide.
    // Conversion du mot en « louchebem »
    p[0] = *buffer;
    *buffer = 'l';
    p[1] = 'e'; p[2] = 'm'; p[3] = '\0';
    puts( buffer );
    fputs( "--> ", stdout );
    gets( buffer );
  }

  printf( "\n\nFin EXO12.\n" );

  return 0;
}
```


Corrigé de l'exercice 13

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

int main( int argc, char **argv )
{
  void   usage (char *s);
  double myatof(char *s);

  // Y a-t-il un argument ?
  if( argc != 2 ) usage( argv[0] );
  // On imprime les résultats des fonctions
  // « atof » et « myatof » pour comparaison.
  printf("%f\n", atof  ( argv[1] ));
  printf("%f\n", myatof( argv[1] ));

  printf("\n\nFin EXO13.\n");

  return 0;
}
```

Corrigé de l'exercice 13 (suite)

```
double myatof( char *s )
{
  long   nombre, signe;
  double exposant;

  exposant = 1.;
  nombre   = 0;
  // Saut des éventuels caractères « espace », « tabulation »
  // et « newline » situés en tête.
  for( ; isspace( *s ); s++ )
    ;
  // Gestion du signe.
  signe = *s == '-' ? -1 : 1;
  *s == '-' || *s == '+' ? s++ : s;
  // Gestion de la partie entière.
  for( ; isdigit( *s ); s++ )
    nombre = nombre*10 + *s - '0';
  if( *s++ != '.' ) return signe*nombre*exposant;
  // Gestion de la partie décimale.
  for( ; isdigit( *s ); s++ )
  {
    nombre = nombre*10 + *s - '0';
    exposant /= 10.;
  }
  return signe*nombre*exposant;
}
void usage( char *s )
{
  fprintf( stderr, "usage: %s nombre.\n", s );
  exit( 1 );
}
```


Corrigé de l'exercice 14

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

// Définitions de nouveaux types.
typedef enum sens {arriere, avant} Sens;
typedef struct cellule
{
   char   *chaine;
   struct  cellule *ptr_precedent;
   struct  cellule *ptr_suivant;
} CEL;

void  liste ( CEL *p, Sens sens );
void  libere( CEL *p );
CEL   *debut = NULL;
```

Corrigé de l'exercice 14 (suite)

```
int main()
{
  CEL   *ptr_courant = NULL;
  char   chaine[40];

  // Boucle de lecture sur l'entrée standard avec « --> » comme prompt.
  fputs( "--> ", stdout );
  gets( chaine );
  while( ! feof( stdin ) )
  {
    // Si « la » est la chaîne entrée, on liste les chaînes déjà saisies.
    if( ! strcmp( chaine, "la" ) )
      liste( debut, avant );
    // Si « li » est la chaîne entrée, on liste les chaînes déjà saisies
    // dans l'ordre inverse.
    else if( ! strcmp( chaine, "li" ) )
      liste( ptr_courant, arriere );
    else
    {
      // C'est la 1re chaîne.
      if( debut == NULL )
      {
        debut = malloc( sizeof(CEL) );
        debut->ptr_precedent = NULL; ptr_courant = debut;
      }
      else
      {
        // C'est une chaîne différente de la 1re.
        ptr_courant->ptr_suivant = malloc( sizeof(CEL) );
        ptr_courant->ptr_suivant->ptr_precedent = ptr_courant;
        ptr_courant = ptr_courant->ptr_suivant;
      }
```


Corrigé de l'exercice 14 (suite)

```
      // On valorise le nouvel élément de la liste.
      ptr_courant->chaine = strdup( chaine );
      ptr_courant->ptr_suivant = NULL;
    }
    fputs( "--> ", stdout );
    gets( chaine );
  }
  // On libère la liste.
  if( debut != NULL ) libere( debut );

  printf( "\n\nFin EXO14.\n" );
  return 0;
}
// Fonction récursive d'affichage de la liste.
void liste( CEL *p, Sens sens )
{
  if( debut == NULL )
  {
    printf( "Désolé! la liste est vide.\n\n" );
    return;
  }
  if ( p != NULL )
  {
    printf( "\t%s\n", p->chaine );
    liste( sens == avant ? p->ptr_suivant : p->ptr_precedent, sens );
  }
  return;
}
// Fonction libérant la mémoire occupée par la liste.
void libere( CEL *p )
{
  if ( p->ptr_suivant != NULL ) libere( p->ptr_suivant );
  free( p->chaine ); free( p );
  return;
}
```

Corrigé de l'exercice 15

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

int main()
{
  typedef struct index
  {
     unsigned int debut;
     unsigned int longueur;
  } INDEX;
  FILE  *mus, *ind, *musind;
  char   buffer[BUFSIZ];
  INDEX  index;

  // Ouverture du fichier texte « musiciens ».
  if( (mus = fopen( "musiciens", "r" )) == NULL )
  {
    perror( "fopen" );
    exit(1);
  }
  // Ouverture en écriture du fichier des musiciens indexés.
  if( (musind = fopen( "musiciens.indexe", "w" )) == NULL )
  {
    perror( "fopen" );
    exit(2);
  }
```


Corrigé de l'exercice 15 (suite)

```
  // Ouverture en écriture du fichier d'index.
  if( (ind = fopen( "musiciens.index", "w" )) == NULL )
  {
    perror( "fopen" );
    exit(3);
  }

  index.debut = ftell( mus );
  // Boucle de lecture du fichier des musiciens.
  fgets( buffer, sizeof buffer, mus );
  while( ! feof(mus) )
  {
    // On supprime le caractère « newline ».
    buffer[strlen( buffer )-1] = '\0';
    fputs( buffer, musind );
    index.longueur = strlen( buffer );
    fwrite( &index, sizeof index, 1, ind );
    // Mise à jour de la position pour l'itération suivante.
    index.debut = ftell( musind );
    fgets( buffer, sizeof buffer, mus );
  }
  // Fermeture des fichiers.
  fclose( ind ); fclose( musind ); fclose( mus );

  printf( "\n\nFin EXO15.\n" );

  return 0;
}
```

Corrigé de l'exercice 16

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

// Définition de types.
typedef struct date
{
  unsigned int date_naiss;
  unsigned int date_mort;
} DATE;

typedef struct musicien
{
  char nom[30];
  char prenom[20];
  DATE date;
} MUSICIEN;

typedef struct index
{
  unsigned int debut;
  unsigned int longueur;
} INDEX;
```

Corrigé de l'exercice 16 (suite)

```c
int main()
{
  MUSICIEN  mort_le_plus_jeune( MUSICIEN *mus, int n );
  void      imprime           ( MUSICIEN *mus, int n );
  int       cmp_alpha ( const void *mus1, const void *mus2 );
  int       cmp_chrono( const void *mus1, const void *mus2 );
  FILE     *ind, *musind;
  INDEX     index;
  MUSICIEN *p_mus;
  MUSICIEN  mus_mort_le_plus_jeune;
  char     *buffer;
  int       NbMus;

  // Ouverture du fichier index.
  if( (ind = fopen( "musiciens.index", "r" )) == NULL )
  {
    perror( "fopen" );
    exit(1);
  }
  // Ouverture du fichier indexé des musiciens.
  if( (musind = fopen( "musiciens.indexe", "r" )) == NULL )
  {
    perror( "fopen" );
    exit(2);
  }
  NbMus = 0;
  p_mus = NULL;
```

Corrigé de l'exercice 16 (suite)

```c
  // Boucle de lecture du fichier indexé des musiciens,
  // par l'intermédiaire du fichier d'index.
  fread( &index, sizeof(INDEX), 1, ind );
  while( ! feof(ind) )
  {
    buffer = malloc( index.longueur+1 );
    fgets( buffer, index.longueur+1, musind );
    p_mus = realloc( p_mus, ++NbMus*sizeof(MUSICIEN) );
    sscanf( buffer, "%s%s%d%d", p_mus[NbMus-1].prenom,
                                p_mus[NbMus-1].nom,
                                &p_mus[NbMus-1].date.date_naiss,
                                &p_mus[NbMus-1].date.date_mort );
    free( buffer );
    fread( &index, sizeof(INDEX), 1, ind );
  }
  // Fermeture des fichiers.
  fclose( ind ); fclose( musind );
  // Affichage de la liste des musiciens.
  printf( "\n\n\t\tListe des musiciens :\n" );
  printf( "\t\t-------------------\n\n" );
  imprime( p_mus, NbMus );
  // Tri puis affichage de la liste des musiciens triés par ordre alphabétique.
  qsort( p_mus, NbMus, sizeof(MUSICIEN), cmp_alpha );
  printf( "\n\n\tListe des musiciens par ordre alphabétique :\n" );
  printf( "\t-----------------------------------------\n\n" );
  imprime( p_mus, NbMus );
  // Tri puis affichage de la liste des musiciens triés par ordre chronologique.
  qsort( p_mus, NbMus, sizeof(MUSICIEN), cmp_chrono );
  printf( "\n\n\tListe des musiciens par ordre chronologique :\n" );
  printf( "\t-------------------------------------------\n\n" );
  imprime( p_mus, NbMus );
```


Annexe C : exercices Corrigés

Corrigé de l'exercice 16 (suite)

```c
  // Recherche du musicien mort le plus jeune.
  mus_mort_le_plus_jeune =  mort_le_plus_jeune( p_mus, NbMus );
  // Affichage du musicien mort le plus jeune, ainsi que sa durée de vie.
  printf( "\n\n\tLe musicien mort le plus jeune est :"
          "\n\t----------------------------------\n\n" );
  printf( "\t%s %s qui est mort à %d ans.\n\n",
                     mus_mort_le_plus_jeune.prenom,
                     mus_mort_le_plus_jeune.nom,
                     mus_mort_le_plus_jeune.date.date_mort -
                     mus_mort_le_plus_jeune.date.date_naiss );

  printf( "\n\nFin EXO16.\n" );

  return 0;
}

// Fonction appelée par « qsort » pour trier les  musiciens par ordre alphabétique.
int cmp_alpha( const void *mus1, const void *mus2 )
{
  return strcmp( ((MUSICIEN *)mus1)->nom, ((MUSICIEN *)mus2)->nom );
}

// Fonction appelée par « qsort » pour trier les musiciens par ordre chronologique.
int cmp_chrono( const void *mus1, const void *mus2 )
{
  if( ((MUSICIEN *)mus1)->date.date_naiss > ((MUSICIEN *)mus2)->date.date_naiss )
    return 1;
  if( ((MUSICIEN *)mus1)->date.date_naiss < ((MUSICIEN *)mus2)->date.date_naiss )
    return -1;
  return 0;
}
```

Corrigé de l'exercice 16 (suite)

```c
// Fonction d'affichage.
void imprime( MUSICIEN *mus, int n )
{
  for( int i=0; i<n; i++ )
    printf( "%-20s%-25s %5d %5d\n", mus[i].prenom, mus[i].nom,
                                    mus[i].date.date_naiss,
                                    mus[i].date.date_mort );

  return;
}

// Fonction recherchant le musicien mort le plus jeune.
MUSICIEN mort_le_plus_jeune( MUSICIEN *mus, int n )
{
  int indice = 0;

  for( int m=1; m<n; m++ )
    if( mus[m].date.date_mort - mus[m].date.date_naiss <
        mus[indice].date.date_mort - mus[indice].date.date_naiss )
      indice = m;

  return mus[indice];
}
```

Corrigé de l'exercice 17

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

typedef struct index
{
  unsigned int debut, longueur;
} INDEX;

int main( int argc, char **argv )
{
  void   erreur( int rc );
  void   usage ( char *s );
  FILE  *ind, *musind;
  int    rang_mus;
  INDEX  index;
  char  *buffer;

  // Le rang a-t-il été spécifié ?
  if( argc != 2 ) usage( argv[0] );
  // Conversion de l'argument en entier.
  rang_mus = (int)strtol( argv[1], NULL, 0 );
  if( rang_mus <= 0 ) erreur( 1 );
  // Ouverture du fichier indexé des musiciens.
  if( (musind = fopen( "musiciens.indexe", "r" )) == NULL )
    perror( "fopen" ), exit(2);
  // Ouverture du fichier d'index.
  if( (ind = fopen( "musiciens.index", "r" )) == NULL )
    perror( "fopen" ), exit(3);
```

Corrigé de l'exercice 17 (suite)

```
  // Positionnement dans le fichier d'index.
  // Ne pas trop compter sur la valeur retournée par la fonction « fseek ».
  // Un mauvais positionnement provoquera une erreur lors de la lecture suivante.
  fseek( ind, (rang_mus-1)*sizeof(INDEX), SEEK_SET );
  // Lecture de l'index contenant le positionnement et la longueur de l'enregistrement
  // dans le fichier indexé des musiciens correspondant au rang spécifié.
  if( fread( &index, sizeof index, 1, ind ) != 1 ) erreur( 4 );
  // Positionnement puis lecture de l'enregistrement désiré.
  fseek( musind, index.debut, SEEK_SET );
  buffer = malloc( index.longueur+1 );
  fgets( buffer, index.longueur+1, musind );
  // Affichage du musicien sélectionné.
  printf( "\n\tmusicien de rang %d ==> %s\n\n", rang_mus, buffer );
  free( buffer );
  // Fermeture des fichiers.
  fclose( ind ); fclose( musind );

  printf( "\n\nFin EXO17.\n" );

  return 0;
}
void erreur( int rc )
{
  fprintf( stderr, "rang invalide.\n" );
  exit( rc );
}

void usage( char *s )
{
  fprintf( stderr, "usage : %s rang\n", s );
  exit( 6 );
}
```


Corrigé de l'exercice 17 : solution avec setjmp/longjmp

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

// Les pseudos-variables suivantes doivent surtout
// correspondre à des valeurs non nulles car celles-ci
// seront utilisées lors des appels à la fonction « longjmp »
#define USAGE           1
#define RANG_INVALIDE   2
#define ERROR_OPEN_INDEX 3
#define ERROR_OPEN_MUS  4
#define ERROR_LEC_INDEX 5

typedef struct index
{
   unsigned int debut;
   unsigned int longueur;
} INDEX;

jmp_buf cntx;

static int  traitement_arg(int argc, char **argv);
static void recherche_mus (int rang_mus);
```

Corrigé de l'exercice 17 : solution avec setjmp/longjmp (suite)

```
int main(int argc, char **argv)
{
  int retour;
  int rang_mus;

  if ( (retour = setjmp( cntx )) == 0 )
  {
    rang_mus = traitement_arg(argc, argv);
    recherche_mus(rang_mus);
  }
  else
    exit(retour);

  printf( "\n\nFin EXO17.\n");

  return 0;
}
```

Corrigé de l'exercice 17 : solution avec setjmp/longjmp (suite)

```
static int traitement_arg(int argc, char **argv)
{
  int rang_mus;

  // Le rang a-t-il été spécifié ?
  if( argc != 2 )
  {
    fprintf( stderr, "usage : %s rang\n", argv[0] );
    longjmp( cntx, USAGE );
  }
  // Conversion de l'argument en entier.
  rang_mus = (int)strtol( argv[1], NULL, 0 );
  if( rang_mus <= 0 )
  {
    fprintf( stderr, "rang invalide.\n" );
    longjmp( cntx, RANG_INVALIDE );
  }

  return rang_mus;
}
```

Corrigé de l'exercice 17 : solution avec setjmp/longjmp (suite)

```c
static void recherche_mus(int rang_mus)
{
  FILE  *ind, *musind;
  INDEX  index;
  char  *buffer;

  // Ouverture du fichier indexé des musiciens.
  if( (musind = fopen( "musiciens.indexe", "r" )) == NULL ) {
    perror( "fopen(mus)" ); longjmp(cntx, ERROR_OPEN_INDEX);
  }
  // Ouverture du fichier d'index.
  if( (ind = fopen( "musiciens.index", "r" )) == NULL ) {
    perror( "fopen(index)" ); longjmp(cntx, ERROR_OPEN_MUS);
  }
  // Positionnement dans le fichier d'index.
  // Ne pas trop compter sur la valeur retournée par la fonction « fseek ».
  // Un mauvais positionnement provoquera une erreur lors de la lecture suivante.
  fseek( ind, (rang_mus-1)*sizeof(INDEX), SEEK_SET );
  // Lecture de l'index contenant le positionnement et la longueur de l'enregistrement
  // dans le fichier indexé des musiciens correspondant au rang spécifié.
  if( fread( &index, sizeof index, 1, ind ) != 1 ) {
    fprintf( stderr, "rang invalide.\n" ); longjmp(cntx, ERROR_LEC_INDEX);
  }
  // Positionnement puis lecture de l'enregistrement désiré.
  fseek( musind, index.debut, SEEK_SET );
  buffer = malloc( index.longueur+1 );
  fgets( buffer, index.longueur+1, musind );
  // Affichage du musicien sélectionné.
  printf( "\n\tmusicien de rang %d ==> %s\n\n", rang_mus, buffer );
  free( buffer );
  // Fermeture des fichiers.
  fclose( ind ); fclose( musind );

  return;
}
```


Corrigé de l'exercice 18

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <fcntl.h>

int *positions;
int  nb_positions;
int  nombre_octets_lus;

static void init( void )
{
  positions = NULL;
  nb_positions = 0;
  nombre_octets_lus = 0;

  return;
}

int main( int argc, char **argv )
{
  void    usage  (char *s);
  void    strrech(char *buffer, int nblus, char *ChaineAchercher, size_t longueur);
  int     fd;
  char    buffer[BUFSIZ];
  char   *ChaineAchercher;
  size_t  longueur;
  int     nblus;
```

Corrigé de l'exercice 18 (suite)

```
  // Le fichier et la chaîne à chercher ont-ils été passés en argument ?
  if ( argc != 3 ) usage( argv[0] );
  ChaineAchercher = argv[2];
  longueur = strlen( ChaineAchercher );

  // initialisation des variables globales.
  init();
  // Ouverture du fichier passé en argument.
  if( (fd = open( argv[1], O_RDONLY )) == -1 )
  {
     perror( "open" );
     exit( 1 );
  }
  // Boucle de lecture du fichier.
  while( nblus = read( fd, buffer, sizeof buffer ) )
  {
    // Récupération des positions de la chaîne dans le buffer courant.
    strrech( buffer, nblus, ChaineAchercher, longueur );

    // Si BUFSIZ caractères ont été lus, on recule de (longueur-1)
    // caractères dans le fichier pour être sûr de n'oublier aucune
    // position de la chaîne lors de la lecture suivante.
    nombre_octets_lus += nblus;
    if( nblus == BUFSIZ )
    {
      lseek( fd, -(long)(longueur - 1), SEEK_CUR );
      nombre_octets_lus -= longueur - 1;
    }
  }
  close( fd );
```


Corrigé de l'exercice 18 (suite)

```
  // Impression des positions trouvées.
  if ( nb_positions == 0 )
    printf( "La chaîne \"%s\" n'a pas été trouvée\ndans le fichier \"%s\".\n",
                  ChaineAchercher, argv[1] );
  else
  {
    printf( "Dans le fichier \"%s\", la chaîne \"%s\"\n"
            "a été trouvée aux positions :\n\n", argv[1], ChaineAchercher );
    for( int pos=0; pos<nb_positions; )
    {
      printf( "%5d", positions[pos] );
      if( ! ((++pos)%12) ) printf( "\n" );
    }
    printf( "\n" );
  }
  free( positions );

  printf( "\n\nFin EXO18.\n" );
}
// Fonction de récupération des positions de la chaîne dans le buffer courant.
void strrech( char *s, int nblus, char *ChaineAchercher, size_t longueur )
{
  char       *buffer;
  static int  n = 0;

  // On prend garde de remplacer les éventuels caractères « nuls »
  // par des blancs pour pouvoir utiliser la fonction « strstr ».
  buffer = malloc( nblus+1 );
  memcpy( buffer, s, nblus );
  for( int i=0; i<nblus; i++ )
    buffer[i] = buffer[i] ? buffer[i] : ' ';
  buffer[nblus] = '\0';
```

Corrigé de l'exercice 18 (suite)

```
  // Boucle de recherche de la chaîne.
  for( char *ptr=buffer; ptr=strstr( ptr, ChaineAchercher );
       ptr+=longueur )
  {
    // Extension du vecteur positions.
    positions = (int *)realloc( positions, ++n*sizeof(int) );
    assert( positions != NULL );
    // Position de la chaîne trouvée par rapport au début du bloc lu.
    positions[n-1] = ptr - buffer + 1;
    // Position de la chaîne trouvée par rapport au début du fichier.
    positions[n-1] += nombre_octets_lus;
  }
  free( buffer );
  nb_positions = n;

  return;
}

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

Corrigé de l'exercice 19 : exo19.h

```
#define taille(t) sizeof(t) / sizeof(t[0])
```

Corrigé de l'exercice 19 : exo19_gestion_liste.h

```
void ajouts      ( void );
void liste       ( void );
void tri         ( void );
void suppression( void );
void vider       ( void );
void arret       ( void );
```

Corrigé de l'exercice 19 : exo19.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "exo19.h"
#include "exo19_gestion_liste.h"

struct menu
{
  char  *texte;
  void (*action)( void );
};

int main()
{
  // Définition du menu.
  struct menu menu[] =
  {
    {"  1 - AJOUTS d'éléments dans une liste chaînée.\n", ajouts},
    {"  2 - AFFICHAGE de la liste chaînée.\n",            liste},
    {"  3 - TRI de la liste chaînée.\n",                  tri},
    {"  4 - SUPPRESSION d'éléments dans la liste.\n",     suppression},
    {"  5 - VIDER la liste.\n",                           vider},
    {"  6 - ARRÊT du programme.\n",                       arret}
  };
  int SelectionMenu( struct menu menu[], int NbChoix );

  // Boucle infinie sur les choix effectués.
  for( ;; )
    menu[SelectionMenu( menu, taille(menu) )].action();
}
```


Annexe C : exercices Corrigés

Corrigé de l'exercice 19 : exo19.c

```
// Fonction renvoyant le choix effectué.
int SelectionMenu( struct menu menu[], int NbChoix )
{
  int   choix;
  char  entree[10];
  char *endp;

  do
  {
    printf( "\n\nListe des choix :\n" );
    for( int m=0; m<NbChoix; m++ )
      printf( "%s", menu[m].texte );

    // Lecture du choix.
    // Attention : si « scanf », lire le « newline » car par la suite
    // les lectures s'effectueront à l'aide de la fonction « gets ».
    //  scanf("%d%*c", &choix);
    gets( entree );
    choix = (int)strtol( entree, &endp, 0 );
    if( *endp != '\0' || choix < 1 || choix > NbChoix )
      printf( "\nERREUR - choix invalide.\n" );
  } while( *endp != '\0' || choix < 1 || choix > NbChoix );
  printf( "\n" );

  return --choix;
}
```

Corrigé de l'exercice 19 : exo19_gestion_liste.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "exo19.h"
#include "exo19_gestion_liste.h"

#define LISTE_VIDE "La liste est vide.\n"

static const char * const prompt_ajout      =
                  "Élément à ajouter[CtrlD pour terminer] --> ";
static const char * const prompt_suppression =
                  "Élément à supprimer[CtrlD pour terminer] --> ";
static const char *prompt;

typedef struct cellule
{
  char           *capitale;
  struct cellule *ptr_precedent;
  struct cellule *ptr_suivant;
} CEL;

static CEL  *debut = NULL, *curseur = NULL;

static bool  liste_vide         ( void );
static void  ajout_cellule      ( char *chaine );
static void  suppression_cellule( void );
static bool  recherche_cellule  ( char *chaine );
static char *lire_chaine        ( void );
static void  affichage_liste    ( CEL *p );
```


Annexe C : exercices Corrigés

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```c
static bool liste_vide( void )
{
  return debut == NULL ? true : false;
}

static void ajout_cellule( char *chaine )
{
  CEL *p;

  // Allocation, valorisation, insertion du nouvel élément.
  p = malloc( sizeof(CEL) );
  p->capitale = chaine;

  if ( liste_vide() )
    p->ptr_suivant = p->ptr_precedent = NULL;
  else
  {
    if ( curseur != debut ) curseur->ptr_precedent->ptr_suivant = p;
    p->ptr_precedent = curseur->ptr_precedent;
    curseur->ptr_precedent = p;
    p->ptr_suivant = curseur;
  }
  curseur = p;
  if( curseur->ptr_precedent == NULL ) debut = curseur;

  return;
}
```

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
static void suppression_cellule( void )
{
  if( curseur == debut )
  {
    // L'élément à supprimer est le 1er de la liste.
    debut = curseur->ptr_suivant;
    if( ! liste_vide() ) debut->ptr_precedent = NULL;
  }
  else
  {
    // L'élément à supprimer n'est pas le 1er de la liste.
    curseur->ptr_precedent->ptr_suivant = curseur->ptr_suivant;
    if( curseur->ptr_suivant != NULL )
      // L'élément à supprimer n'est pas le dernier de la liste.
      curseur->ptr_suivant->ptr_precedent = curseur->ptr_precedent;
  }
  {
    CEL *p = curseur;

    free( p->capitale ); free( p );
    if ( curseur->ptr_suivant != NULL )
      curseur = curseur->ptr_suivant;
    else
      curseur = debut;
  }
  return;
}
```


Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
static bool recherche_cellule( char *chaine )
{
  CEL *p;

  for( p=debut; p; p=p->ptr_suivant )
    if ( ! strcmp( p->capitale, chaine ) )
      break;

  if( p != NULL )
  {
    curseur = p;
    return true;
  }

  return false;
}
static char *lire_chaine( void )
{
  char buffer[BUFSIZ];

  // Lecture de l'élément à ajouter.
  fputs( prompt, stdout ); gets( buffer );
  // Si « Control-D », annuler le bit indicateur de fin de fichier
  // pour les prochaines saisies.
  if( feof( stdin ) )
  {
    clearerr( stdin );
    return NULL;
  }
  return strdup( buffer );
}
```

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
// Fonction rattachée au choix 1.(AJOUTS d'éléments dans la liste chaînée).
void ajouts( void )
{
  char *chaine;

  // Boucle de lecture des chaînes.
  prompt = prompt_ajout;
  while( (chaine = lire_chaine()) != NULL )
    ajout_cellule( chaine );
  return;
}

// Fonction rattachée au choix 2. (AFFICHAGE de la liste chaînée).
void liste( void )
{
  if ( liste_vide() )
  {
    fprintf( stderr, LISTE_VIDE );
    return;
  }
  affichage_liste( debut );
  return;
}
static void affichage_liste( CEL *p )
{
  if( p != NULL )
  {
    printf( "\t%s\n", p->capitale );
    affichage_liste( p->ptr_suivant );
  }
  return;
}
```


www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
// Fonction rattachée au choix 3. (TRI de la liste chaînée).
void tri( void )
{
  bool  tri_terminee;
  CEL  *ptr;

  // La liste doit exister.
  if ( liste_vide() ) fprintf( stderr, LISTE_VIDE );
  else
  {
    // Boucle de tri.
    do
    {
      tri_terminee = true;
      for( ptr=debut; ptr->ptr_suivant; ptr = ptr->ptr_suivant )
        if( strcmp( ptr->capitale, ptr->ptr_suivant->capitale ) > 0 )
        {
          // On effectue une interversion.
          curseur = ptr;
          ajout_cellule( strdup( curseur->ptr_suivant->capitale ) );
          curseur = ptr->ptr_suivant;
          suppression_cellule();
          tri_terminee = false;
          if ( ptr->ptr_suivant == NULL ) break;
        }
    } while( ! tri_terminee );
  }

  return;
}
```

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
// Fonction rattachée au choix 4. (SUPPRESSION d'éléments dans la liste).
void suppression( void )
{
  char *chaine;

  // Boucle de lecture des chaînes.
  prompt = prompt_suppression;
  while( ! liste_vide() && (chaine = lire_chaine()) != NULL )
  {
    if( ! recherche_cellule( chaine ) )
    {
      fprintf( stderr, "L'élément \"%s\" est inexistant!\n\n", chaine );
      continue;
    }
    suppression_cellule();
    printf( "L'élément \"%s\" a été supprimé de la liste.\n\n", chaine );
  }
  // La liste est-elle vide ?
  if ( liste_vide() ) fprintf( stderr, LISTE_VIDE );

  return;
}
```

Corrigé de l'exercice 19 : exo19_gestion_liste.c (suite)

```
// Fonction rattachée au choix 5. (VIDER la liste ).
void vider( void )
{
  if ( liste_vide() ) fprintf( stderr, LISTE_VIDE );
  else
  {
    curseur = debut;
    while ( ! liste_vide() )
      suppression_cellule();
  }

  return;
}
// Fonction rattachée au choix 6. (ARRET du programme).
void arret( void )
{
  // Si la liste n'est pas vide, on libère la mémoire qu'elle occupe.
  if( ! liste_vide() ) vider();

  printf( "\n\nFin EXO19.\n" );

  exit( 0 );
}
```

– Symboles –

#define 107
#ifdef 113, 115
#ifndef 113, 115
#include 111
#undef 107
_Bool 23
_Complex double 23
_Complex float 23
_iob 143

– A –

accès à l'environnement 189
accès direct 157, 211
acos 197
aiguillage 97
allocation dynamique 185
ANSI 9
argument variable-fonction 131
argument-fonction 127
argument-structure 125
argument-vecteur 123
arguments-main 129
argv 129
ASCII
    table 215
asctime 187
asin 197
assert 193
assert.h 193
atan 197
atof 175
atoi 175
atol 175
auto 17, 75

– B –

Bell 9
bloc 91
boucle pour 95

Annexe C : exercices | Corrigés

boucle tant-que 95
break 17, 97, 99
bsearch 199, 201

– C –

calloc 185
caractères d'échappement 47
case 17, 97
cast 61
ceil 197
champ de bits 37
champs de bits 35, 37
char 17, 23, 25
classes de mémorisation 75
clock 187
clock_t 187
close 207
commentaire 17
compilation 19
compilation conditionnelle 113, 115
composantes
    structure 35
compteur ordinal 191
const 17, 51
constante agrégat 51
constantes caractères 47
constantes chaînes de car. 49
constantes entières 45
constantes réelles 45
constantes symboliques 51
constructeur homogène 31
constructeurs hétérogènes 35
constructeurs homogènes 29, 31
continue 17, 99
conversions 55
cos 197
cosh 197
creat 207
ctime 187
ctype.h 113, 173

Patrick Corde (*Patrick.Corde@idris.fr*) | Langage C | 4 juillet 2012 | 282 / 282

– D –

déclaration 29
déclaration d'une fonction 87, 89
définition d'une fonction 87, 89
définition de type 41
default 17
defined 117
descripteur de fichier 207, 211
directives-préprocesseur 105
do 17
do-while 95
double 17, 23
durée de vie d'une var. 73

– E –

edition des liens
    édition des liens 19
else 17
enum 17
enumérateurs
    énumérateurs 27
enumérations
    énumérations 27
environnement du shell 129
envp 129
EOF 149, 151
errno 195, 197
errno.h 113, 195
etiquette de cas
    étiquette de cas 97
exercices
    corrigés
        exercice 1 231
        exercice 10 241
        exercice 11 : 1er solution 241
        exercice 11 : 2e solution 247
        exercice 12 251
        exercice 13 251
        exercice 14 253
        exercice 2 231
        exercice 3 233

Annexe C : exercices | Corrigés

        exercice 4 233
        exercice 5 235
        exercice 6 235
        exercice 7 237
        exercice 8 237
        exercice 9 239
exit 103
exp 197
expression de type 41
expressions de type 41
extern 17, 75, 79, 89

– F –

fabs 197
fclose 145
fcntl.h 113, 207
fdopen 211
feof 149, 153, 155
fermeture de fichier 205
ferror 149, 153
fflush 173
fgetc 147
fgets 151, 153
fichier
    fermeture 145
    ouverture 145
fichier en-tête 111
FILE 143, 211
fileno 211
fin de fichier 149, 151, 209
float 17, 23
floor 197
flot 143, 211
fonction 31
fonction d'écriture 209
fonction de lecture 209
fonction de recherche 199
fonction de tri 201
fonctions 17
fonctions de conversions 175
fopen 145

Patrick Corde (Patrick.Corde@idris.fr) | Langage C | 4 juillet 2012 | 282 / 282

for 17, 95
format 159, 161, 167
fprintf 159, 161
fputc 147
fputs 151, 153
fread 153, 155
free 185
freopen 173
fscanf 159, 167
fseek 155, 157
ftell 155, 157
fwrite 153, 155

– G –

getc 147
getchar 147
getenv 189
gets 151
getw 149
go to 101
goto 17

– I –

identificateurs 15
identificateurs de fonctions 51, 53
identificateurs de vecteurs 51, 53
if 17, 93
inclusion de fichiers 111
indirection 29
initialisation
    structure 85
    union 85
    vecteur 85
initialisation des variables 83
inline 127
instruction élémentaire 91
instruction composée 17, 87, 91
instruction d'échappement 97
instruction préprocesseur 17
instruction simple 17

Annexe C : exercices | Corrigés

instructions d'échappement 99
int 17, 23
isalnum 173
isalpha 173
iscntrl 173
isdigit 173
isgraph 173
islower 173
isprint 173
ispunct 173
isspace 173
isupper 173
isxdigit 173

– K –

Kernighan 9

– L –

localtime 187
log 197
log10 197
long 17, 23, 25
long double 23
long int 23, 25
longjmp 191
lseek 211

– M –

mémoire tampon 143, 149, 173
macros 109
main 17
malloc 185
math.h 113, 197
matherr 197
membres
    structure 35
memchr 181
memcmp 181
memcpy 181

Patrick Corde (Patrick.Corde@idris.fr) | Langage C | 4 juillet 2012 282 / 282

memmove 181
memset 181
mktime 187
mode_t 207
mots réservés 17

– N –

Niveau d'une variable 73
NULL 145, 151, 153

– O –

O_APPEND 207
O_CREAT 207
O_EXCL 207
O_RDONLY 207
O_RDWR 207
O_TRUNC 207
O_WRONLY 207
off_t 211
opérateur conditionnel 65
opérateur d'adressage 59
opérateur d'ap. de fonction 69
opérateur d'incrémentation 65
opérateur d'indexation 67
opérateur d'indirection 59
opérateur de décrémentation 65
opérateur de forçage de type 61
opérateur de taille 59
opérateur séquentiel 67
opérateurs à effet de bord 63
opérateurs arithmétiques 53
opérateurs d'affectation 63
opérateurs de décalage 63
opérateurs de manip. de bits 61
opérateurs de sél. de champ 69
opérateurs logiques 57
open ..... 205, 20[illegible]
ouverture de fichier 205

Annexe C : exercices Corrigés

– P –

passage arguments 131, 133
perror 195
pile 75, 131, 133, 191
pointeur 29, 31, 33, 55
pointeur générique 141
pointeurs 31
portée d'une variable 83
portée des var. externes 77
portée des var. internes 75
pow 197
préprocesseur 105
printf 159, 161
programme multifichiers 79
prototype d'une fonction 87
pseudo-constantes 107
    prédéfinies 107
pseudo-expressions 117
pseudo-fonctions 109
putc 147
putchar 147
puts 151, 153
putw 149

– Q –

qsort 201

– R –

read 209
realloc 185
register 17, 75, 77
restauration du contexte 191
return 17, 101
Richards 9
Ritchie 9

– S –

séquences d'échappement ..... 4[illegible]
sauvegarde du contexte 191

Patrick Corde (Patrick.Corde@idris.fr) Langage C 4 juillet 2012 282 / 282

scanf 159, 167
SEEK_CUR 155
SEEK_END 155
SEEK_SET 155
setjmp 191
setjmp.h 113, 191
short 17, 23, 25
short int 23, 25
signal.h 113
signed 17, 25
signed char 25
sin 197
sinh 197
size_t 179
sizeof 17, 35, 59
snprintf 159, 161, 167
spécification de conversion 169
spécifications de conversions 161, 167
sprintf 159, 161
sqrt 197
sscanf 159, 167
stack 75
static 17, 75, 83
stderr 143, 173, 195
stdin 143, 147, 151, 173
stdio.h 113, 143, 195, 211
stdlib.h 175, 185, 199, 201
stdout 143, 147, 151, 173
strcat 177
strchr 179
strcmp 179
strcpy 177
strdup 177
stream 143, 211
strerror 195
string.h 113, 177, 195
strlen 179
strncat 177
strncmp 179
strncpy 177
strrchr 179

Annexe C : exercices Corrigés

strstr 179
strtod 175
strtol 175
strtoul 175
struct 17, 41
struct exception 197
struct tm 187
structure 35
structures 35, 153
structures de contrôle 93
switch 17, 97
sys/types.h 207, 211

– T –

table ASCII 215
tableaux 153
tan 197
tanh 197
tell 211
test 93
Thompson 9
time 187
time.h 113, 187
time_t 187
tolower 173
toupper 173
typedef 17, 41
types de base 23

– U –

umask 207
union 17, 39, 41
unions 35
unistd.h 211
unsigned 17, 25
unsigned char 25
unsigned long 25

Patrick Corde (*Patrick.Corde@idris.fr*) Langage C 4 juillet 2012 282 / 282

– V –

va_alist 137
va_arg 135, 137
va_dcl 137
va_end 135, 137
va_list 135, 137
va_start 135, 137
varargs.h 113
variable permanente 73
variable temporaire 73, 121
vecteur 31, 33
visibilité d'une variable 75
visibilité des fonctions 87
void 17, 23
void * 141
volatile 17

– W –

while 17, 95
write 209

– X –

X3J11 9

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com