Langage C : référence (très) rapide

Pascal AMSILI

IRIT

UNIVERSITE PARIS 7

AP 1993-1997

Table des matières

1 Généralités

1.1 Historique

- Né en 1972, aux « Bell Labs » (Dennis Ritchie).
- But initial : langage de haut niveau permettant de faire de la programmation système.
- Utilisé en 1973 pour ré-écrire le système UNIX (Bell Labs, 1969, Brian Kernighan et Ken Thompson).
- Premier livre de référence : « *The C programming language* », par Kernighan et Ritchie (1978). Variante dite « traditionnelle » ou K&R.
- Normalisation de l'ANSI en 1988 ⇒ co-existence de deux « dialectes » : K&R et ANSI.

1.2 Exemple : extrait d'un programme C « réaliste »

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    @(#)liens.c        1.2  93/06/11
    Nom          : liens.c
    Auteur  : PA
    Syntaxe : K&R 8bits
    Contient     : Etablit les liens entre deux abreviations/sigles.
                  Hypothèse : on lit les sigles dans le fichier
                         d'entrée sous la forme :  N/Forme Développée
    Algo :
                       Pour Fi forme développée du sigle n i
                         Pour chaque mot mi de Fi
                           Pour chaque forme Fi != Fj
                             si mi \in Fj alors l(i,j)++
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include <stdio.h>
#include <string.h>

/*******************************************************************\
         Fonctions utilisées
\*******************************************************************/
void initliens() ;
...
void lecture() ;
void affichage() ;

/*******************************************************************\
         CONSTANTES
\*******************************************************************/
#define M 90            /* Taille max d'une forme développée */
#define T 500           /* Nombre max d'entrées traitées */

/*******************************************************************\
         VARIABLES GLOBALES
\*******************************************************************/
struct fd {             /* Enregistrement "forme développée" : */
  char f[M] ;           /* Forme elle-même */
  int  n;               /* No du sigle correspondant */
} ;
typedef struct fd fd;
fd tab[T] ;             /* Tableau des formes développées */
int nbf = 0;            /* Nombre de valeurs dans le tableau */
                        /* précédent */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*\
    Fonction   : lecture()
    entrée     : /
    sortie     : /
    modifie    : tab[], nbf
    Rôle       : lit sur l'entrée standard les enregistrements au
          format : <n de sigle>/<forme développée>(<commentaire>)
  L'opération de lecture se fait avec une ``ss-fonction'', sousLect()
\*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
int sousLect(no)
int no ;
{
  char buf[BUFSIZ] ;
  int i;

  i=scanf("%d/%[^(\n]%*[(]%*[^)]%*[)]\n",&(tab[no].n),tab[no].f,buf,buf) ;
  return i;
}
void lecture()
{
  int chp_lus;

  while((chp_lus=sousLect(nbf))!=EOF && chp_lus !=0)
    nbf++ ;

  if (chp_lus==0) {
    fprintf(stderr,"ERREUR DE SYNTAXE, enregistrement n %d\n",nbf) ;
    fprintf(stderr,"Reste du fichier ignoré.\n") ;
  }
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*\
    Fonction   : appartient()
    entree     : char * mot
                 char * chaine
    sortie     : booléen (int)
    modifie    :
    Role       : détermine si le mot est dans la chaîne (sous forme
                 de mot : utilise strtok()).
\*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
int appartient(mot, chaine)
char * mot, *chaine ;
{
  char *pc, buf[M] ;

  strcpy(buf,chaine) ;
  for (pc=strtok(buf,delim); pc!=NULL ; pc=strtok(NULL,delim))
    if (strcmp(pc,mot)==0) return 1;
  return 0;
}

/*******************************************************************\
          PROGRAMME PRINCIPAL
\*******************************************************************/
main()
{
  lecture();
  liens();
  affichage();
}
```

1.3 Objet de base : la fonction

Un seul type de sous-programme : la *fonction*. Valeur de retour : type scalaire quelconque. Par défaut : *int* ; les « procédures » renvoient le type *void*.

Une fonction étant définie dans un fichier, son utilisation dans d'autres fichiers nécessite qu'elle y soit *déclarée* pour que le compilateur sache la reconnaître. La déclaration consiste à donner le *prototype* de la fonction.

Définition	Prototype
type nom (type a_1 *[, type* a_2*...])* *{* *déclarations de variables* *instructions* *}*	*type nom (type a1 [, type a2...]) ;*

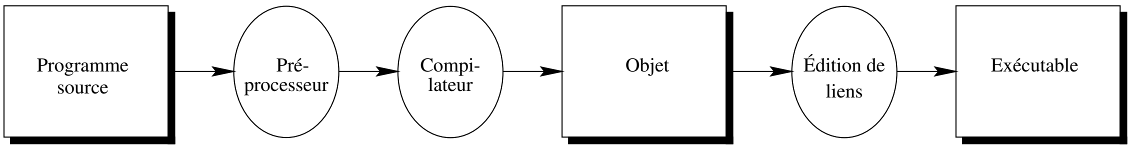
La syntaxe traditionnelle (K&R) est quelque peu différente :

Définition	Prototype
type nom () *type* a_1 *[, type* a_2*...]* *{* *déclarations de variables* *instructions* *}*	*type nom () ;*

À titre d'exemple, voici les prototypes en syntaxe ANSI des fonctions qui apparaissent dans le listing de la section 1.2 :

```
int sousLect(int no)
void lecture(void)
int appartient(char * mot, *chaine)
int main(void)
```

1.4 Compilation



Cf. figure 1

1.5 Le préprocesseur

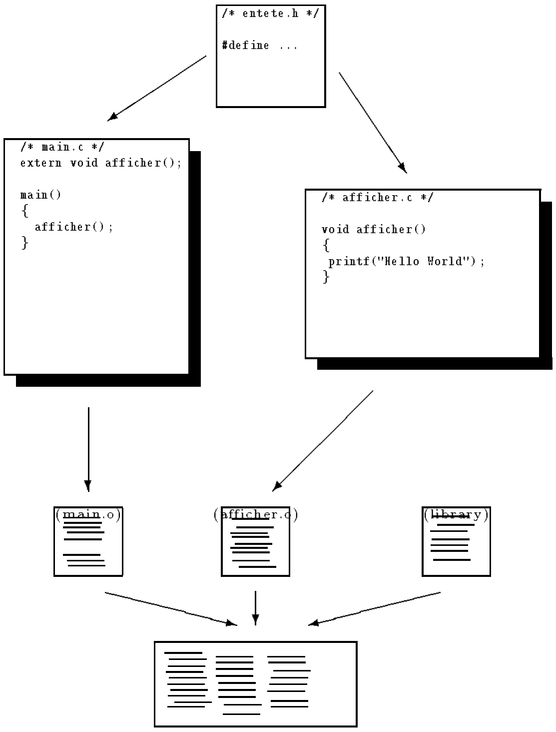
1.5.1 Pseudo-constantes

`#define` ⟨*identificateur*⟩ ⟨*espace ou tab*⟩ ⟨*chaîne de substitution*⟩

1.5.2 Pseudo-fonctions

`#define` ⟨*identificateur*⟩(*lpf*) ⟨*espace ou tab*⟩ ⟨*chaîne de substitution*⟩

FIG. 1 - *Compilation « séparée »*



1.5.3 Directives d'inclusion

`#include<`*fichier*`>`	Fichier se trouvant dans les répertoires d'inclusion par défaut du système.
`#include"`*fichier*`"`	Fichier se trouvant dans le répertoire courant.

2 Types et variables

2.1 Types élémentaires

- Types de base
- Types énumérés
- Pointeurs

2.1.1 Types de base

	void
[unsigned]	*char*
[unsigned]	short [int]
[unsigned]	*int*
[unsigned]	long [int]
	float
	double
	long double

2.1.2 Types énumérés

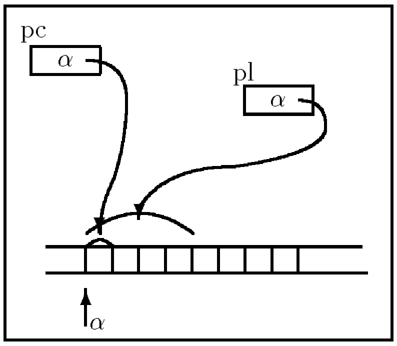
```
enum Feu {vert,orange,rouge,cligno,en_panne};
⋮
enum Feu a, b ;
```

Syntaxe générale (Idem pour `struct`) :

`enum X {..};`	Définition du type `enum X`.
`enum X {..} y;`	Définition du type `enum X`, et d'une variable `y` de ce type.
`enum X y;`	Déclaration d'une variable `y` de type `enum X` préalablement défini.
`enum {..} y;`	Déclaration d'une variable `y` de type « énumération de constantes » sans que son type soit nommé.

2.1.3 Pointeurs

```
char * pc ;
long * pl ;
```



2.2 Définitions et déclarations de variables

Syntaxe d'une déclaration (types simples) :

type identificateur [, *identificateur...*] ;

Constructeurs de type :

Constructeur	Type construit	Syntaxe
`*`	Pointeur	*type* `*` *pointeur*
`[ ]`	Vecteur (homogène)	*type vecteur*`[`*taille*`]`
`( )`	Fonction	⟨*type de retour*⟩ *fonction*`(`*lpf*`)`
`struct`	Agrégat (hétérogène)	`struct` *nom* `{`*champs* `}`
`union`	Union disjointe (hétérogène)	*idem*

2.3 Tableaux

Forme générale :

type nom`[`*dimension*`]` ;

Exemples :

```
int tab[100] ;
char Etat[NBEtats] ;
float * f[2] ;
float matrice[100][100] ;
int premier[5]={1,2,3,5,7};
float liste[]={1.314,0.,2.67};
char ch[]="Chaine" ;
```

2.4 Initialisation et affectation de variables

Les deux choses se font avec le signe `=`. Presque toutes les variables peuvent être initialisées lors de leur déclaration :

```
int i=1, j=256, k ;
char c='\n', car='+' ;
```

2.5 Implémentation et visibilité

Mots-clés pour modifier les classes de stockage par défaut : `auto`, `extern`, `static`, `register`.

2.5.1 Implémentation des variables

Variables de niveau 0 : statiques.
Variables de niveau supérieur : dynamiques (automatiques).
Le mot-clé `auto` correspond exactement à ce comportement par défaut, il est donc redondant.

On ne peut pas rendre dynamique les variables de niveau 0.
Les variables peuvent être rendues statiques grâce au mot clé `static` devant la déclaration.

Le mot-clé `register` permet d'indiquer au compilateur que l'on désire, si c'est possible, que la variable soit placée dans un registre. Ce n'est valable que pour les types élémentaires.

2.5.2 Visibilité des variables

Variables de niveau 0 : globales au fichier, accessibles à l'extérieur (sous réserve d'une déclaration `extern`).

L'utilisation du mot-clé `static` avec une variable de niveau 0 la rend locale au fichier (c'est-à-dire invisible à l'extérieur).

Les variables de niveau n sont visibles dans le bloc où elles sont déclarées et dans tous les sous-blocs.

2.5.3 Visibilité des fonctions

Par défaut, les fonctions sont globales à toute l'application, à condition, comme pour les variables globales, qu'elles soient déclarées dans chaque fichier où elles sont utilisées. Le mot clé `extern` est facultatif, il permet cependant d'augmenter la lisibilité.

Le mot-clé `static` permet – c'est rarement utilisé – de rendre une fonction locale à un fichier.

3 Expressions

3.1 Les constantes

3.1.1 Constantes littérales

Constantes littérales du C :

Type		Syntaxe	Exemples
Entiers	(décimal)	Que des chiffres	`10240`
	(octal)	Précédé de `0`	`032` (=26)
	(hexadécimal)	Précédé de `0x` ou `0X`	`0xFF` (=255)
	(long)	Suivi de `l/L`	`235L`
Réels		Caractère «`.`» ou «`e/E`»	`37.2`, `0.`, `1E-10`
		Suivi de `f/F`	`12F`
Caractères		Lettre entre apostrophes	`'A'`, `'+'`
		Code ascii en octal	`'\033'` (=ESC), `'\012'`
		Code spécial (cf table)	`'\n'`
Chaîne de caractères		Entre guillemets (")	`"Hello World\n"`

Certains caractères « non affichables », et particulièrement utiles, peuvent être codés avec une séquence `'\c'` où `c` est un caractère :

`\a`	BEL	[ANSI] (Sonnerie)
`\b`	BS	(*Backspace*)
`\f`	FF	(Saut de page)
`\n`	NL	(Nouvelle ligne)
`\r`	CR	(Retour chariot)
`\t`	HT	(Tabulation horizontale)
`\v`	VT	(Tabulation verticale)
`\\`	`\`	
`\'`	`'`	
`\"`	`"`	
`\?`	`?`	[ANSI]

3.1.2 Constantes symboliques

- Constantes énumérées
- Identificateurs de tableaux
- Identificateurs de fonctions

3.2 Les opérateurs

Table des opérateurs par ordre de priorité décroissante :

<table>
<tr><th>Opérateur</th><th>Ass.</th><th></th></tr>
<tr><td><code>() []</code>
<code>-> .</code></td><td>→</td><td>fonction et tableau
sélection de champ</td></tr>
<tr><td><code>! ~ + -</code>
<code>++ --</code>
<code>* &</code>
<code>(</code>type<code>) sizeof</code></td><td>←</td><td>négations et signes (unaires)
auto-in/décrémentation
indirections
« transtypage » et taille</td></tr>
<tr><td><code>* / %</code></td><td>→</td><td>opérateurs multiplicatifs</td></tr>
<tr><td><code>+ -</code></td><td>→</td><td>opérateurs additifs</td></tr>
<tr><td><code><< >></code></td><td>→</td><td>décalages</td></tr>
<tr><td><code>< <= > >=</code></td><td>→</td><td>comparaisons</td></tr>
<tr><td><code>== !=</code></td><td>→</td><td>égalité et différence</td></tr>
<tr><td><code>&</code></td><td>→</td><td>ET bit à bit</td></tr>
<tr><td><code>^</code></td><td>→</td><td>XOR bit à bit</td></tr>
<tr><td><code>|</code></td><td>→</td><td>OU bit à bit</td></tr>
<tr><td><code>&&</code></td><td>→</td><td>ET booléen</td></tr>
<tr><td><code>||</code></td><td>→</td><td>OU booléen</td></tr>
<tr><td><code>?:</code></td><td>←</td><td>opérateur conditionnel ternaire</td></tr>
<tr><td><code>=</code>
<code>+= -= *= /= %=</code>
<code>>>= <<= &= ^= |=</code></td><td>←</td><td>affectation (simple)
affectations
combinées</td></tr>
<tr><td><code>,</code></td><td>→</td><td>séquence</td></tr>
</table>

4 Les instructions

4.1 Instructions élémentaires

4.1.1 Expressions

Toute expression suivie d'un point-virgule est une instruction.

4.1.2 Structure de bloc

Un bloc se compose d'un ensemble de déclarations suivies d'un ensemble d'instructions, le tout entre accolades (`{ }`).

4.2 Instructions composées

4.2.1 Test

Forme générale :

```
if ( expression )
    instruction
[else
    instruction]
```

4.2.2 Aiguillage

Forme générale :

```
switch ( expression ) {
  case constante1: instruction
  case constante2: instruction
  ⋮
 [default: instruction]
}
```

Si on veut obtenir le comportement d'un `case` Pascal, on ajoute l'instruction `break` qui fait directement sortir du `switch`.

4.2.3 Boucles

La boucle « tant que », `while`, a deux syntaxes qui correspondent approximativement à la distinction entre boucle « tant que » et boucle « jusqu'à ».

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

Boucle `for` : syntaxe particulière et spécialement puissante pour cette instruction classique :

```
for ( e1 ; e2 ; e3 )
  instruction
```

L'instruction est équivalente à :

```
e1 ;
while ( e2 ) {
  instruction
  e3 ;
}
```

4.3 Instructions d'échappement

L'instruction `continue` permet de « sauter » à la fin de la boucle, pour faire le tour suivant sans exécuter le reste des instructions du corps de la boucle.

L'instruction `break` permet de « sauter » à la première instruction qui suit la boucle.

L'instruction `goto` existe, mais on n'en dira pas plus ici.

Enfin, l'instruction `return` *expression* permet de sortir d'une fonction et de renvoyer une valeur de retour.

Récapitulation des instructions :

Instructions élémentaires :	
Expressions	*expression* ;

Bloc	{
	[*déclarations*]
	[*instructions*]
	}

Instructions structurées :	

Test	if (*expression*)
	    instruction
	[else
	    instruction]

Aiguillage	switch (*expression*) {
	  case $constante_1$: *instruction*
	        [break ;]
	  case $constante_2$: *instruction*
	        [break ;]
	  ⋮
	 [default: *instruction*]
	}

Boucle while	while (*expression*) *instruction*
Boucle do	do *instruction* while (*expression*) ;
Boucle pour	for (e_1 ; e_2 ; e_3) *instruction*
Instructions d'échappement :	
Retour de fonction Sortie de boucle Tour suivant Branchement	`return` *expression* ; `break` ; `continue` ; `goto` *étiquette*

5 Chaînes de caractères

Implémentation : tableau de caractères terminé par le caractère de code ASCII 0 (`\0`).

Fonctions classiques de manipulation :

```
double atof(char * chaîne)
int atoi(char * chaîne)
long atol(char * chaîne)
```

Convertissent la chaîne de caractères *chaîne* en valeur réelle (`f`), entière (`i`) ou entière longue ('`a`' pour ASCII).

```
char * strcpy(char * destination, char * source)
char * strncpy(char * destination, char * source, int i)
```

Fait une copie (caractère `'\0'` compris) de la chaîne *source* dans *destination*. *source* doit être une chaîne, c'est-à-dire se terminer par un `'\0'`, et *destination* doit correspondre à une zone de taille suffisante. La seconde fait la même chose, mais s'arrête à `i` caractères (ou à `'\0'` si celui-ci est rencontré avant).

```
char * strcat(char * destination, char * source)
char * strncat(char * destination, char * source, int i)
```

Concaténation : fait une copie (caractère `'\0'` compris) de la chaîne *source* au bout de la chaîne *destination*. *source* doit être une chaîne, c'est-à-dire se terminer par un `'\0'`, et *destination* doit correspondre à une zone de taille suffisante.

Les quatre fonctions précédentes renvoient un pointeur sur la zone destinataire.

```
int strcmp(char * chaîne1, char * chaîne2)
int strncmp(char * chaîne1, char * chaîne2, int i)
```

Compare les deux chaînes. Le résultat est : $\left.\begin{array}{l} < 0 \text{ si } \textit{chaîne1} < \textit{chaîne2} \\ = 0 \text{ si } \textit{chaîne1} = \textit{chaîne2} \\ > 0 \text{ si } \textit{chaîne1} > \textit{chaîne2} \end{array}\right\}$ ordre lexicographique

```
int strlen(char * chaîne)
```

Renvoie la longueur ('`\0`' non compté) de la chaîne *chaîne*.

```
char * strchr(char * chaîne, char caractère)
```

Renvoie un pointeur sur la première occurence du *caractère* dans la *chaîne*.

6 Les entrées/sorties standards

Toutes les fonctions données ici travaillent sur un « flot », manipulé par l'intermédiaire d'un pointeur sur le type `FILE` défini dans `stdio.h`.

6.1 Fonctions formatées sur écran/clavier

6.1.1 `printf` & `scanf`

Ecriture formatée sur la sortie standard (l'écran) :

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

Lecture formatée sur l'entrée standard (en général clavier) :

```
int scanf(char * format, (void *) tampon, ...)
```

Cette fonction lit sur le clavier (entrée standard) des données selon le format donné comme premier argument, et les stocke dans les variables dont les adresses sont passées en paramètre.

La fonction renvoie le nombre de champs qu'elle a décodé.

6.1.2 Format

La forme générale d'un format est :

`%`[*préfixe*][*largeur*][`.`*précision*][*modificateur*]*type*

Les *types* possibles sont :

`d` ou `i` (`u`)	Entier décimal signé (non signé)
`o`	Entier octal
`x` (`X`)	Entier hexadécimal (majuscules)
`f`	Réel en notation décimale
`e` (`E`)	Réel en notation « scientifique »
`g` (`G`)	Format f ou e (E) suivant la précision
`c`	Caractère
`s`	Chaîne de caractères
`p`	Pointeur

Les *préfixes* possibles sont :

rien	Cadré à droite et complété par des blancs
`-`	Cadré à gauche et complété par des blancs
`+`	Affichage systématique du signe
espace	Valeurs positives préfixées par un blanc (au lieu du signe `+`)
`#`	Formaté selon les conventions des constantes C.

Largeur et *précision* sont des entiers. On peut aussi donner `*`, ce qui signifie que la valeur est l'argument suivant (qui doit donc être de type entier).

```
float x= 678.78945;
printf("|%10.*f|",4,x);
↪           | 678.7894|
```

Les valeurs possibles du *modificateur* sont : `h` (entier court), `l` (entier long), et `L` (réel long double). N.B. : afficher un entier long (par exemple) sans donner le bon modificateur donne des résultats incohérents.

6.2 Fonctions d'ouverture et de fermeture

Tout programme C a par défaut au moins 3 « fichiers » ouverts au lancement : l'entrée standard (en général le clavier) `stdin`, la sortie standard (en général l'écran) `stdout`, et la sortie erreur (en général l'écran) `stderr`.

Les fonctions sont :

```
FILE * fopen(char * fichier, char * mode d'accès)
int fclose(FILE * flot)
```

La fonction `fopen()` renvoie `NULL` s'il se produit une erreur. `fclose()` renvoie 0 dans le cas normal.

Modes d'accès :

Mode d'accès	Paramètre	Position	Comportement si le fichier existe	n'existe pas
Lecture	`r`	Début	-	erreur
Ecriture	`w`	Début	vidé	créé
	`a`	Fin	-	créé
Lecture & Ecriture	`r+`	Début	-	erreur
	`w+`	Début	vidé	créé
	`a+`	Fin	-	créé

Focntion de redirection :

```
FILE * freopen(char * fichier, char * mode d'accès, FILE * flot)
```

```
freopen("/tmp/trace","w",stdout);
```

6.3 Fonctions formatées de lecture et écriture

```
int fprintf(FILE * fichier, char * format, variables...)
int fscanf(FILE * fichier, char * format, pointeurs...)
```

Formats: cf p 11.

Retour : nombre de caractères écrits (`fprintf`), ou nombre de champs décodés (`fscanf`).

`printf(...)` `scanf(...)`	est une abréviation de	`fprintf(stdout,...)` `fscanf(stdin,...)`

6.4 Accès séquentiel et accès direct

```
int fseek(FILE * fichier, long décalage, int position)
long int ftell(FILE * fichier)
```

`ftell()` renvoie la position en octets du « curseur » dans le fichier.

`fseek()` permet de déplacer le « curseur » dans le fichier, d'un certain *décalage* (en octets) à partir : du début (*position* à `0`) ; de la position courante (*position* à `1`) ; de la fin (*position* à `2`). Renvoie 0 en cas de réussite.

```
fseek(f,16*sizeof(char),0); /* Saut des 16 premiers caractères*/
fscanf(f,"%c",&car);
```

6.5 Contrôle de la mémorisation

Les fonctions vues ici sont des entrées/sorties « mémorisées » (*buffered*).

Le tampon (*buffer*) associé à un flot de sortie est envoyé quand :

- il est plein
- le flot est fermé
- la fonction `fflush()` est appelée

```
int fflush(FILE * fichier)
```

Comportements spécifiques additionnels selon les systèmes d'exploitation.

6.6 Fonctions non formatées

Remarque : `int` $\rightsquigarrow$ `char`.

```
char [f]getc(FILE * fichier)
[f]putc(char caractère, FILE * fichier)
```

Renvoie le caractère lu/écrit ou `EOF`.

```
char * fgets(char * tampon, int max, FILE * fichier)
```

Lit *max*-1 caractères ou jusqu'au *newline* et les met dans le tampon ('`\n`' devient '`\0`'). Renvoie le pointeur sur *tampon* ou `NULL`.

```
int fputs(char * tampon, FILE * fichier)
```

Ecrit la chaîne, caractère '`\0`' non compris.

```
#include <stdio.h>

main(int argc, char *argv[])
{
   char buffer[BUFSIZ];
   FILE * f;
   f = fopen(argv[1],"r");

   while (fgets(buffer,BUFSIZ,f)!= NULL)
     fputs(buffer,stdout);

   fclose(f);
}
```

Abréviations pour les E/S standard :

`getchar()`	`getc(stdin)`
`putchar(c)`	`putc(c,stdout)`
`gets(s)`	`fgets(s,BUFSIZ,stdin)`
`puts(s)`	`fputs(s,stdout)` (+ `fputs("\n")`)

7 Bibliographie sommaire

Jean-Pierre Braquelaire (1991). *Méthodologie de la programmation en langage C*. Masson. *Excellente description complète du langage, référence incontournable pour ce qui concerne la programmation en C sous* UNIX.

Jean-Marie Riflet (1986). *La programmation sous* UNIX. McGraw Hill. *Uniquement pour les aspects liés à la programmation en C sous* UNIX. *Une référence en la matière. Ré-éditions plus récentes.*

Brian Kernighan et Dennis Ritchie (1978). *Le langage C*. Masson. *La référence. Maintenant mis à jour selon la norme ANSI.*

Robert Sedgewick (1991). *Algorithmes en langage C*. InterEditions - Addison-Wesley Europe. *Plutôt orienté algorithmique et structures de données, mais on y trouve tout.*