partie 5 : La bibliothèque standard d'entrées/sorties

MOOC Langage C INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

La bibliothèque `stdio` (`standard input/output`) permet d'accéder aux fichiers et de les manipuler à l'aide d'un ensemble de fonctions d'emploi aisé. Tout programme qui utilise cette bibliothèque doit inclure le fichier d'en-tête `stdio.h` avec la directive :

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

A chaque fichier ouvert est associé un tampon dit d'entrée/sortie ; cette association est appelée un flux ou flot. Elle permet essentiellement de diminuer le nombre d'appels système, et, partant, d'améliorer le trafic entre processeur et mémoire de masse. Les fonctions de la bibliothèque nous autorisent à accéder aux fichiers :

- par caractères,
- par mots,
- par chaînes de caractères,
- par lignes,
- par blocs,
- par données formatées.

1 Ouverture d'un fichier

1.1 Fonction `fopen`

Avant d'accéder à un fichier en lecture ou en écriture, il est nécessaire de l'ouvrir. Pour cela, on doit utiliser la fonction `fopen()` dont voici le prototype :

```
FILE * fopen ( char * nomfic, char * mode);
```

Cette fonction renvoie l'adresse d'une structure de type `FILE`. En réalité, `fopen()` a créé un flux, dont toutes les caractéristiques sont mentionnées dans la structure de type `FILE`. L'utilisateur doit conserver dans un pointeur approprié l'adresse de cette structure car toutes les opérations ultérieures sur le flux `y` feront référence :

```
FILE *pFic;
pFic = fopen ( NomFichier, ModeOuverture);
```

1.2 Paramètres d'appel

Voyons les paramètres d'appel à partir de l'exemple précédent : `NomFichier` est l'adresse de la chaîne de caractères contenant le nom du fichier (incluant si nécessaire le chemin d'accès - mais cela peut nuire à la portabilité), tandis que `ModeOuverture` est l'adresse d'une autre chaîne déterminant la manière dont le fichier va être employé. Voici les modes d'ouverture possibles :

- `"r"` en lecture
- `"w"` en écriture
- `"a"` pour ajout en fin de fichier,
- `"r+"` en lecture plus écriture : ainsi le fichier peut être lu, modifié et agrandi,
- `"w+"` ouvert en écriture plus relecture.

Si on ouvre en mode écriture (ou ajout) un fichier qui n'existe pas, le système d'exploitation le crée. Si on tente d'ouvrir en lecture un fichier qui n'existe pas, il y a évidemment erreur. La fonction `fopen()` renvoie alors la constante `NULL` (définie dans le fichier à inclure `stdio.h`, et dont la valeur est 0) qui indique l'absence de flux. Aussi est-il habituel d'ouvrir ainsi un fichier :

```
if ( (pFic = fopen (NomFic, Mode)) == NULL )
{
  printf ("\nOn ne peut pas ouvrir %s\n", NomFic);
  exit (0);            /* Sortie brutale, non structuree */
}
/* ... Suite logique, tout va bien */
```

Cependant, si vous voulez rendre convivial votre programme, adoptez une solution structurée avec possibilité à l'utilisateur de se repentir ou de se reprendre. Notez que dans ce programme, le message d'erreur est affiché dans le flux de sortie standard. Nous verrons dans le paragraphe 7 comment, selon l'usage, le diriger vers le flux erreur dont nous parlerons dans le paragraphe 3.

2 Fermeture d'un fichier

La fonction :

```
int fclose(FILE * pFic);
```

vide le tampon, puis libère la mémoire qui lui était allouée, et enfin referme le fichier du flux référencé ici par `pFic`. Ce pointeur se trouve alors périmé, et peut être réutilisé avec un autre flux. `fclose()` renvoie 0 si la fermeture du fichier s'est réalisée correctement, sinon elle retourne -1. Le nombre de fichiers simultanément ouverts étant limité pour un utilisateur donné, il convient de refermer ceux dont on ne se sert plus. La fonction `fclose()` est d'ailleurs appelée automatiquement à la fin du traitement pour fermer les fichiers encore ouverts.

3 Accès caractère par caractère

Considérons maintenant le cas le plus simple, celui de l'accès caractère par caractère. Pour cela, on emploie les macro-instructions `getc()` et `putc()`. Elles permettent, respectivement, de lire ou d'écrire un caractère dans le flux, donc, finalement, dans le fichier. A chaque appel, elles incrémentent automatiquement le pointeur courant du tampon associé au fichier.

```
int Caractere;

Caractere = getc (pFic);
```

La macro-instruction `getc()` lit un caractère dans le flux référencé par le pointeur `pFic`, et renvoie la valeur de ce caractère. Le caractère lu est celui pointé par le pointeur courant du tampon du flux, qui est post-incrémenté. Quand on lit le dernier caractère d'un fichier, la valeur retournée est celle d'une constante particulière, `EOF` (end of file), définie dans `stdio.h`. Remarquez que bien que la valeur lue soit un caractère, la valeur retournée est un entier pour garder la compatibilité avec d'autres fonctions de la bibliothèque. Cet entier est signé, car la valeur de `EOF` est en générale négative. La macro `getc()` fait donc une extension (sur 32 bits pour nos machines) du signe dans la valeur du caractère lu avant de la renvoyer. Voici un exemple classique d'emploi de `getc()` :

```
while ( (Caractere = getc (pFic)) != EOF )
{
/* ... Utiliser le caractere lu */
}
```

La macro-instruction `putc()` effectue l'opération inverse en plaçant un caractère dans le tampon du flux référencé par `pFic` :

```
int putc ( int Caractere, FILE *pFic);
```

Ici aussi, `putc()` renvoie une valeur entière, le caractère lui-même. Si cette valeur est `EOF`, cela signifie que l'écriture n'a pu se réaliser. En général, on dédaigne cette valeur retournée.

4 Flux d'entrée, flux de sortie, flux d'erreur

Considérons un programme écrit en C et tournant dans un environnement UNIX. Les entrées et les sorties sont alors considérées comme des fichiers. Lorsque l'utilisateur donne l'ordre d'exécuter son programme, le processus shell lance ce programme dans un processus fils, où il a créé trois flux :

- l'entrée standard (par défaut le clavier),
- la sortie standard (par défaut l'écran),
- la sortie erreur (l'écran aussi).

La référence de chacun de ces flux (de type `FILE *`) est une constante dénommée respectivement `stdin`, `stdout`, `stderr`, avec les valeurs `(FILE *) 0`, `(FILE *) 1` et `(FILE *) 2`. Soulignons bien que les fichiers standards n'ont pas besoin d'être ouverts par `fopen()`. Bien que les fichiers standards soient normalement des terminaux, ils peuvent être redirigés vers d'autres fichiers ou pipelines. Ainsi, voici les instructions d'entrée au clavier et d'affichage à l'écran :

```
Caractere = getc (stdin);
putc (Caractere, stdout);
```

On peut aussi employer `getchar()` et `putchar()` :

```
Caractere = getchar ();
putchar (Caractere);
```

qui sont deux macro-instructions standards (cf. annexe **??**) ainsi définies :

```
#define  getchar() getc(stdin)
#define  putchar(x)  putc(x, stdout);
```

5 Entrées/sorties de mots, de lignes

5.1 Manipulation de mots

La bibliothèque `stdio` offre la possibilité de manipuler des mots mémoire, en les considérant comme des entiers conservés en binaire naturel. Les fonctions de lecture et d'écriture sont respectivement `getw` et `putw`. Leur emploi est analogue à `getc()` et `putc()`. Ainsi, `getc()` prélève l'information dans le flux 8 bits par 8 bits, alors que `getw()` effectue la même chose 32 bits par 32 bits.

5.2 Manipulation de lignes

1. La fonction `fgets()`, dont voici le prototype :

   ```
   char * fgets ( char * Ligne, int MAX, FILE * pFic);
   ```

   lit des caractères dans le fichier référencé par `pFic` et les range en mémoire à partir de l'adresse `Ligne` qui doit évidemment correspondre à un tableau de taille suffisante. La fonction s'arrête de lire lorsque le caractère interligne `'\n'` et rencontré ou que `MAX` -1 caractères ont été lus. `fgets()` ajoute ensuite le caractère nul `('\0')` dans `Ligne`. Cette fonction renvoie l'adresse de la chaîne saisie, c'est-à-dire `Ligne`, ou la constante `NULL` si un problème est survenu ou que la fin du fichier a été atteinte :

   ```
   char chaine_lue[100];

   if (fgets(chaine_lue,100,pFic)!=NULL) {
    ...
   }
   ```

2. La fonction `fputs()` :

   ```
   int fputs ( char * Ligne, FILE * pFic);
   ```

   La fonction `fputs()` écrit la chaîne `Ligne` (qui doit être terminée par un caractère nul) dans le fichier référencé par `pFic`. Le caractère nul n'est pas recopié dans le fichier.
   Comme précédemment, il existe des fonctions travaillant directement avec les fichiers standards :

   ```
   char * gets (char * Ligne);
   int  puts (char * Ligne);
   ```

6 Entrées/sorties par blocs

L'accès par bloc (autrement dit, en vrac) est normalement la méthode la plus rapide pour traiter les fichiers, pourvu que les blocs soient suffisamment gros. Voici les deux fonctions d'entrée/sortie par bloc d'octets :

```
int fread (void * Tampon, int Taille_Bloc, int Nombre,
          FILE * pFic);
int fwrite (void * Tampon, int Taille_Bloc, int Nombre,
           FILE * pFic);
```

dont les arguments sont les suivants :

— `Tampon` est l'adresse en mémoire d'un tableau d'octets origine ou destination du transfert. Une fois de plus, il est de l'entière responsabilité du programmeur de réserver la place suffisante en mémoire, s'il s'agit d'un tableau destination.

— `Taille_Bloc` est la taille d'un bloc, en octets.

— `Nombre` est le nombre de blocs à lire.

— `pFic` est l'adresse du flux concerné.

Ces deux fonctions retournent le nombre de blocs transférés. Remarquez que les blocs représentent des données brutes (considérées comme des caractères), si bien que ces fonctions exigent souvent l'utilisation de forçage de type comme par exemple si l'on veut écrire dans un fichier la valeur d'une structure `Client` :

```
fwrite (&Client, sizeof Client, 1, pFic);
```

7 Le tampon du flux

7.1 Déplacement du pointeur dans le tampon

La fonction :

```
int fseek ( FILE * pFic, int Deplac, int Origine);
```

déplace le pointeur dans le tampon de `Deplac` octets par rapport à `Origine`, selon la convention suivante :

— si `Origine` vaut 0, le déplacement est compté par rapport au début du fichier (déplacement absolu),

— si `Origine` vaut 1, le déplacement est compté par rapport à la position courante du pointeur (déplacement relatif),

— si `Origine` vaut 2, le déplacement est effectué en partant de la fin du fichier.

`fseek()` renvoie 0 s'il n'y a pas d'erreur, sinon -1.

La fonction :

```
int ftell (FILE * pFic);
```

donne la valeur en octets de la position du pointeur dans le tampon, par rapport au début du fichier. Par exemple, les quelques instructions suivantes fournissent la taille d'un fichier :

```
fseek (pFic, 0, 2);         /* Va a la fin du fichier */
Taille = ftell (pFic);        /* Taille du fichier */
fseek (pFic, 0, 0);     /* Ramene au debut du fichier */
```

7.2 Taille du tampon

La fonction :

```
void setbuf (FILE * pFic, char * Tampon);
```

permet d'allouer au flux un tampon de la taille que l'on veut, et de nom explicite `Tampon`. Celui-ci remplace le tampon alloué automatiquement. Ceci est intéressant pour le programmeur qui a donc accès à ce tampon sans avoir recours aux fonctions déjà vues. `setbuf()` est normalement utilisée juste après l'ouverture du fichier concerné. Il est aussi possible de supprimer le tampon (accès non tamponné), ce qui présente un intérêt dans certaines applications de périphériques. Il suffit pour cela de fournir une adresse de tampon nulle. Ainsi :

```
void setbuf ( pFic, NULL);
```

7.3 Pour vider un tampon

La fonction :

```
int fflush (FILE * pFic);
```

vide le tampon associé au flux `pFic`. Le fichier reste ouvert. Elle retourne 0 s'il n'y a pas eu d'erreur. Cette fonction permet de synchroniser les écritures, en particulier avec les périphériques standards, et son utilisation devient indispensable quand les écritures se trouvent à l'intérieur de boucles. Ainsi :

```
fflush (stdout);
```

vident le tampon de sortie.

Concernant le tampon d'entrée, il faut se méfier de l'utilisation de cette fonction. `fflush(stdin)` n'est pas standard et son comportement peut être différent d'un système à l'autre. Sur les sysèmes Solaris et Linux (depuis la `glibc 2.1.95`), il est possible d'utiliser la fonction `__fpurge(stdin)` qui fonctionne bien mais qui, elle non plus, n'est pas standard.

Si vous devez vider le tampon d'entrée d'un terminal utilisé en mode standard (appelé mode canonique sous UNIX), vous pouvez aussi employer la boucle suivante :

```
while (getchar() != '\n')
  {}
```

En effet, en mode canonique, toute entrée au clavier doit s'achever par une pression sur la touche `return`, ce qui se traduit par le caractère `lf ('\n')` dans le tampon (et non pas le caractère `'\r'` !). Ceci permet donc à l'utilisateur d'effectuer des corrections locales à la saisie avant de la valider par le `return`.

Un exemple d'utilisation de `__fpurge` avec l'entrée standard est donnée dans le paragraphe 8.2.

8 Entrées et sorties formatées

En langage C, les entrées/sorties se font seulement à l'aide de fonctions dont vous connaissez déjà les deux plus fréquemment utilisées :

```
scanf () pour les entrees,
printf () pour les sorties.
```

Remarquez que ces fonctions, ainsi que d'autres apparentées (`getchar(), putchar(), sscanf(), ssprintf(), fscanf(), fprintf()`) ou de traitement de chaînes (`atoi(), strlen()`), se trouvent déjà placées dans la bibliothèque standard du langage, consultée automatiquement par l'éditeur de liens. Il est cependant nécessaire d'inclure `stdio.h`, car l'absence des prototypes des fonctions utilisées dans un programme entraîne, au minimum, un message d'avertissement.

8.1 La fonction `printf()`

```
int printf (char * Format [,Liste d'arguments]);
```

Cette fonction convertit, met en forme et envoie les arguments à la sortie standard, sous le contrôle de la chaîne `Format`. Cette chaîne peut contenir trois types d'objets :

1. des caractères ordinaires, qui sont simplement recopiés dans la suite des résultats,

Spécification	L'argument est présenté en
c	un seul caractère
s	une chaîne de caractères
d	décimal entier
o	décimal octal
x	décimal hexadécimal
u	décimal non signé
f	réel (float)
lf	réel double
e	réel notation exponentielle
g	réel, printf() choisit f ou e.

TABLE 1 – Les conversions de type usuelles

2. des spécifications de conversion : chaque spécification commence par le caractère % et explique comment doit être affiché l'argument correspondant. La conversion est spécifiée par un (ou deux) caractères, dont voici les plus importants (tableau 1) :
3. Optionnellement, des nombres précisant la taille du champ de cadrage de la valeur à afficher et des indicateurs (flags) :
   - — - (moins) le cadrage est fait à gauche (par défaut, il est à droite),
   - — + (plus) le signe du nombre est toujours affiché,
   - — . (point) le nombre est précédé de un ou plusieurs 0 (zéros) jusqu'à compléter son cadrage.

Exemples, avec le résultat de l'exécution :

```
printf ("Bonjour !");
Bonjour !

int Val = 27;
printf ("%d  %o  %x", Val, Val, Val);
Resultat -> 27  33  1b

char Lettre = 'a';
printf ("Caractere : %c  HexASCII : %x  Decimal : %d", Lettre, Lettre, Lettre);
Resultat -> Caractere : a  HexASCII : 0x61  Decimal : 97

int Neg = -23;
printf ("\n Neg = %.8d", Neg);
Resultat -> Neg = -00000023
```

Attention ! C'est le nombre de conversions (%) qui fixe le nombre d'arguments sortis. Le langage C n'effectue, une fois de plus, aucune vérification de la cohérence entre le format et les arguments : si la correspondance entre arguments et conversions est erronée, l'affichage des résultats peut être tronqué (trop d'arguments) ou imprévisible (trop de conversions). La valeur entière retournée par la fonction printf() donne le nombre de caractères affichés.

Autre information importante concernant le format d'affichage :

```
int i=312;
printf("%5d\n",i);
```

Le format %5d signifie que l'entier i va être affiché exactement avec 5 caractères (dans le cas présent, 2 espaces sont placés juste avant i puisque 312 ne fait que 3 caractères). De la même manière, les nombres flottants peuvent être formatés, aussi bien pour la partie entière que la partie décimale. Par exemple :

```
float f=12.8957;
printf("%6.3f\n",f);
```

va provoquer l'affichage de :

```
 12.896
```

%6.3f signifie ici que l'affichage de f sera fait avec 6 caractères **au minimum, point inclus**, dont 3 sont réservés à la partie décimale.

8.2 La fonction scanf()

C'est le pendant de printf(), en entrée :

```
int scanf (char * Format, <Liste d'arguments>)
```

Puisque les arguments de la fonction sont des paramètres de sortie (elle leur affecte une valeur), il faut obligatoirement les passer par référence. Le principe du format de conversion est grosso modo le même que celui de printf(), mais l'utilisation du format dans scanf() est plus délicate : il faut savoir que la conversion s'arrête dès qu'elle a trouvé un caractère séparateur (espace, tabulation, ou interligne '\n'), ou un caractère incompatible avec le type spécifié dans le format. Par exemple, si l'on tape au clavier :

```
La Ville Rose
```

la chaîne affectée au tableau Article[] dans l'instruction :

```
scanf ("%s", Article);    /* Article est une reference ! */
```

se réduit à "La", le reste de la chaîne restant dans le tampon d'entrée. Notez que dans cet exemple, on pouvait saisir les trois mots à l'aide de trois tableaux ainsi :

```
char Article [4], Ville [31], Teinte [12];
scanf ("%s%s%s", Article, Ville, Teinte);
```

Voici un exemple d'erreur classique. On a l'instruction suivante :

```
int Entier;
scanf ("%d", &Entier);
```

qui recherche un entier dans le tampon d'entrée. Si au lieu de taper 207, on frappe par erreur 2o7, la valeur prise par Entier est 2, et non 207, et il reste o7 et un '\n' dans le tampon. Il est donc indispensable, en langage C, de toujours vérifier les valeurs enregistrées lors de l'exécution ou, au moins, de pratiquer l'affichage des valeurs lues (écho). Voici enfin une autre erreur classique. On désire faire une entrée au clavier correctement filtrée. On a alors écrit :

```
int Entier;
do
{
  printf ("Une valeur enti\‘ere entre 5 et 10, svp ? ");
  scanf ("%d", &Entier);
  printf ("\nVous avez fourni : %d\n", Entier);
  /* echo pour verification */
}
while (Entier < 5 || Entier > 10);
```

Une telle saisie fonctionne parfaitement tant que l'on fournit des entiers. Malheureusement, si l'on tape par erreur un caractère autre qu'un chiffre, la saisie rentre en boucle infinie. En effet, ici `Entier` n'est pas initialisé : il contient donc 0 ou une valeur indéfinie, selon la classe de mémorisation de cette variable. La fonction `scanf()` trouve dans son tampon d'entrée une lettre. Comme elle attend un ou des chiffres, elle s'interrompt, en laissant la lettre dans le tampon ! Elle ne change pas la valeur de `Entier`, si bien que l'on effectue un second passage dans la boucle. Comme le tampon d'entrée n'est pas vide, le programme ne s'arrête pas pour faire une entrée au clavier. La fonction `scanf()` examine le caractère pointé et, comme ce n'est pas un chiffre... etc ! (Boucle infinie.) Une fois compris le problème, la correction est facile. Il faut donc impérativement vider le tampon après la fonction d'entrée :

```
int Entier;
do
{
  printf ("Une valeur enti\‘ere entre 5 et 10, svp ? ");
  scanf ("%d", &Entier);
  __fpurge (stdin);
  printf ("\nVous avez fourni : %d\n", Entier);
                          /* echo pour verification */
}
while (Entier < 5 || Entier > 10);
```

Les caractères de conversion sont pratiquement les mêmes que pour `printf()`. Nous allons y rajouter deux caractères de conversion, `n` et `*`. `%n` associée à la conversion `%s`, fournit la taille de la chaîne lue (sans compter le caractère nul final). Par exemple, si on entre `Toulouse` grâce à cette instruction :

```
scanf ("%s%n", Ville, &NbCar);
```

le tableau `Ville` reçoit le nom de la ville, et `NbCar` reçoit 8.

Si on fait précéder un caractère de conversion de `*`, ceci permet d'ignorer l'argument associé à cette conversion. C'est donc une manière de retirer du tampon d'entrée certaines données attendues, sans les conserver. Si vous avez :

```
double Temperature;
scanf ("%lf%*s", &Temperature);
```

et que vous entrez :

```
364.5 degres_C
```

le mot `degres_C` est ignoré, et retiré du tampon d'entrée, et la variable `Temperature` contiendra la valeur `364.5`, sans erreur.

8.3 La fonction fscanf()

```
int fscanf (FILE *pFic, char * Format, <Arguments>)
```

permet de lire dans le flux pointé par `pFic` les données dont le nombre et le type sont précisés dans la chaîne de format, et les affecte aux arguments fournis par adresse. Remarquons tout de suite que `scanf()` et l'appel de `fscanf()` avec `stdin` en premier argument sont exactement la même chose. Pour que la lecture se fasse correctement, il faut que les variables formatées soient stockées dans le fichier avec un séparateur entre les variables. Ce séparateur peut être un ou plusieurs caractères espace, `'\n'`, ou tabulation. Attention ! Le caractère nul, `'\0'`, n'est pas un caractère de séparation !

Exemple :

```
fscanf (pFic, "%s%d", Nom, &Telephone);
```

lit une chaîne qui est rangée à partir de l'adresse `Nom`, puis lit la chaîne suivante qui est convertie en un entier affecté à `Telephone`. La chaîne et les caractères représentant le nombre doivent impérativement être séparés dans le fichier, par un espace par exemple, sinon, le nombre serait compris dans la chaîne.

8.4 La fonction fprintf()

Duale de la fonction précédente, elle permet d'écrire des données formatées dans un fichier. Il est de la responsabilité du programmeur de gérer correctement les séparateurs. Si on souhaite écrire un code postal suivi du nom d'une ville, il faut faire :

```
fprintf (pFic,"%5d  %s  ", Code, Ville);
```

ainsi vous obtiendrez dans le fichier la séquence de caractères suivante (les espaces y ont été soulignés) :

```
31520__RAMONVILLE__
```

Là aussi, `printf(...)` est équivalent à `fprintf (stdout,...)`. Une utilisation intéressante de `fprintf()` permet de sortir les messages d'erreur par le flux erreur prévu en langage C. Voici comme exemple une macro (cf. chapitre **??**) qui réalise l'affichage d'un message d'erreur numéroté puis arrête le programme :

```
#define Fatal(Num){fprintf (stderr,"\nErreur %d\n",Num); \
exit (1);}
```

8.5 Fonctions de conversion en mémoire

Il est aussi possible de faire des conversions formatées en prenant comme source et destination non plus un fichier, mais une chaîne de caractères conservée en mémoire. Ceci compense partiellement le fait que le langage C, à l'inverse du Pascal, par exemple, ne permet pas de créer des fichiers formatés directement en mémoire centrale. Les fonctions correspondantes sont `sprintf()` et `sscanf()`.

```
int sprintf ( char *Chaine, char *Format [, Arguments]);
```

exécute les mêmes conversions que `printf()`, mais range son résultat dans `Chaine` (comme toujours, à l'utilisateur de réserver la place suffisante). Si tout se passe bien, la fonction renvoie le nombre de caractères copiés. De manière analogue, `sscanf()` permet d'extraire, en la convertissant selon les spécifications de la chaîne `Format[]`, de l'information d'une chaîne de caractères. Par exemple :

```
sscanf (Chaine, "%d", &Entier);
```

est en fait similaire à :

```
Entier = atoi(Chaine);
```

8.6 Exemple final

```
/* Sort a l'ecran le mot le plus long contenu dans un
   fichier texte passe en argument de la ligne de commande.
   Ce mot peut indifferemment etre ecrit en majuscules ou
   en minuscules, ou les deux a la fois.
   Tout caractere autre qu'une lettre est un separateur.
*/

#include <stdio.h>

int main (int argc, char * argv[])
{
  FILE *pFic;
  char Mot[40], MotLong[40];
  char *Courant = Mot;
  int CarLu, Taille = 0, TailleMax = 0;

  /* Test de l'argument d'appel */
  if (argc != 2)
  {
    printf ( "\nUsage : il faut un fichier en argument\n");
    exit (1);
  }

  /* Ouverture du fichier en lecture */
  if ( (pFic = fopen (argv[1], "r")) == NULL)
  {
    printf ("\nErreur : on ne peut pas ouvrir %s\n", argv[1]);
    exit (2);
  }

  /* Lecture sequentielle et analyse du fichier */
  while ((Carlu = getc(pFic)) != EOF)/* Lit une lettre */
    if ( CarLu >= 'a' && CarLu <= 'z' || CarLu >= 'A' && CarLu <= 'Z' )
    {
                    /* Forme le mot par accumulation */
      *Courant++ = CarLu;
      Taille++;
    }
    else
    {               /* Mot le plus long ? */
      *Courant = '\0';
      if (Taille > TailleMax)
      {
                  /* recopie Mot[] en MotLong[] */
        strcpy (MotLong, Mot);
        TailleMax = Taille;
      }
      Taille = 0;
      Courant = Mot;
    }

  /* Fermeture du fichier */
  fclose (pFic);       /* Facultatif, mais c'est
                          une bonne habitude*/

  /* Affiche le resultat */
  printf ("\nLe mot le plus long est : %s (%2d caracteres)\n",
```

```
          MotLong, TailleMax);
  return 0;
}
```