Entrées / Sorties en C

Pierre Rousselin

22 octobre 2017

1 Rappel sur les fichiers sous Unix

Sur les systèmes Unix, *tout est fichier* et il y a une interface homogène depuis et vers toutes les ressources mises à disposition par la machine (« vrais » fichiers sur le disque dur, écran, clavier, souris, etc). Un *descripteur de fichier* est un entier qui, pour un processus donné, représente un fichier ouvert. Il y a 3 fichiers spéciaux qui sont toujours *ouverts* pour chaque programme :

— l'entrée standard `stdin` de descripteur `0`, en général associée au clavier ;

— la sortie stardard `stdout` de descripteur `1`, en général associée à la sortie du terminal ;

— et l'erreur standard `stderr` de descripteur `2`, aussi associée à la sortie du terminal.

On peut faire des *redirections* de ces sorties :

— `> nom_fic` : écrire sur `nom_fic`, c'est-à-dire rediriger `stdout` sur le fichier `nom_fic`.

— `>> nom_fichier` : « append », ajouter à la fin la sortie standard dans le fichier `nom_fic`.

— `< nom_fic` : l'entrée vient du fichier `nom_fic`.

— `>&` : rediriger un descripteur de fichier vers un autre descripteur de fichier

```
$ echo lol
lol
$ ls lol.txt
ls: impossible d'accéder à 'lol.txt': Aucun fichier ou dossier ↩
   de ce type
$ echo lol > lol.txt
$ ls lol.txt
lol.txt
$ cat lol.txt
lol
$ echo ahah > lol.txt
$ cat lol.txt
ahah
$ echo lol >> lol.txt
$ cat lol.txt
ahah
lol
$ wc lol.txt
2 2 9 lol.txt
$ wc
```

Remarque : sans argument, `wc` lit sur l'entrée standard pour dire que le fichier est fini (`EOF`) il faut entrer `Ctrl-D`.

```
$ wc < lol.txt
2 2 9
$ cat lol.txt > rofl.txt
$ cat lol.txt >> rofl.txt
$ cat rofl.txt
ahah
lol
```

```
ahah
lol
$ cat a
cat: a: Aucun fichier ou dossier de ce type
$ cat a 2> rofl.txt
$ cat rofl.txt
cat: a: Aucun fichier ou dossier de ce type
$ cat a 2>> rofl.txt
$ cat rofl.txt
cat: a: Aucun fichier ou dossier de ce type
cat: a: Aucun fichier ou dossier de ce type
$ cat lol.txt a
ahah
lol
cat: a: Aucun fichier ou dossier de ce type
$ cat lol.txt a >> rofl.txt 2>&1
$ cat rofl.txt
cat: a: Aucun fichier ou dossier de ce type
cat: a: Aucun fichier ou dossier de ce type
ahah
lol
cat: a: Aucun fichier ou dossier de ce type
```

Pourquoi je raconte tout ça ? Parce que vous connaissez déjà `printf`, qui écrit sur `stdout` et aussi `scanf` qui lit sur `stdin`. Donc vous pouvez déjà faire des entrées sorties en C.

```c
/* carres.c super-calculateur de carrés */
#include <stdio.h> /* printf et scanf */

int main()
{
    int i, n;
    for (i = 0; i < 5 ; i++) {
        scanf("%d", &n); /* explication du & plus tard */
        printf("%d\n", n * n);
    }
    return 0;
}
```

```
$ gcc -Wall carres.c -o carres
$ chmod u+x carres
$ ./carres
2
4
(...)
$ ./carres > carres.txt
2
7
(...)
$ cat carres.txt
(...)
```

```
$ ./carres < carres.txt > puiss4.txt
$ cat puiss4.txt
(...)
```

C'est déjà pas mal mais :

1. Si on veut écrire dans plusieurs fichiers ?
2. Si on veut se garder la sortie standard pour afficher des messages sur le déroulement du programme (par exemple : `printf("Je vais demander 5 nombres entiers\n");`) sans « polluer » le fichier écrit ?

La bibliothèque standard du C permet de faire ceci de façon très simple et efficace. Les types et fonctions utilisées sont déclarées dans le fichier d'entête `stdio.h` (*standard input output*). Mais d'abord je dois vous parler un peu des pointeurs.

2 Introduction aux pointeurs

Les pointeurs ne sont pas au programme de l'examen et sont présentés ici simplement dans un but pédagogique pour vous expliquer les parties suivantes sans vous dire « c'est comme ça ».

2.1 Qu'est-ce qu'un pointeur ?

Déclaration, initialisation d'un pointeur, valeur d'un pointeur. Opérateurs `&` et `*` :

```c
/* pointeurs1.c */
#include <stdio.h>

int main()
{
    int i, j, *p;
    i = 3; j = 5;
    p = &i;
    printf("i vaut %d, p vaut %p\n", i, p);
    printf("&i vaut %p, *p vaut %d\n", &i, *p);
    /* p est une variable et peut donc changer de valeur */
    p = &j;
    printf("p vaut %p, *p vaut %d\n", p, *p);
    /* p pointe vers j et en passant par p on peut modifier la ↩
       valeur de j !*/
    *p = 17;
    printf("j vaut %d\n", j);
    /* p étant une variable a aussi une adresse */
    printf("adresse de p : p vaut %p, &p vaut %p, *(&p) vaut %p\↩
       n", p, &p, *(&p));
    return 0;
}
```

Petit schéma du déroulement du programme.

2.2 Exemple d'utilisation : fonction swap (échanger)

Première version :

```c
#include <stdio.h>

void swap( int i, int j );
int main()
{
    int i = 3;
    int j = 5;
    printf("Avant l'appel à swap :\n");
    printf("i = %d, j = %d\n", i, j);
    swap(i,j);
    printf("Après l'appel à swap :\n");
    printf("i = %d, j = %d\n", i, j);
    return 0;
}
void swap( int i, int j )
{
    int k = i;
    i = j;
    j = k;
}
```

On compile et on execute... Déception ! Petit schéma pour illustrer le passage par valeur :

Comment faire ?

```c
#include <stdio.h>

void swap( int *pi, int *pj );
int main()
{
    int i = 3;
    int j = 5;
    printf("Avant l'appel à swap :\n");
    printf("i = %d, j = %d\n", i, j);
    swap(&i,&j);
    printf("Après l'appel à swap :\n");
    printf("i = %d, j = %d\n", i, j);
    return 0;
}
```

```c
void swap( int *pi, int *pj )
{
    int k = *pi;
    *pi = *pj;
    *pj = k;
}
```

Pourquoi les arguments qui suivent le format dans `scanf` ont un `&` ?

2.3 Identificateur de tableaux et constantes littérales

```c
#include <stdio.h>
int main()
{
    char s[] = "Hello!"; /* Taille du tableau ? */
    char c = 'A';
    printf("chaine pointée par s : %s\n", s);
    printf("caractère *s : %c\n", *s);
    printf("valeur de s : %p\n", s);
#if 0
    s = &c;
    printf("caractère *s : %c\n", *s);
    printf("valeur de s : %p\n", s);
#endif
    return 0;
}
```

Compiler et executer une première fois, puis une deuxième en enlevant les directives `#if 0` et `#endif`. La même chose avec un pointeur :

```c
#include <stdio.h>

int main()
{
    char *s = "Hello!";
    char c = 'A';
    printf("chaine pointée par s : %s\n", s);
    printf("caractère *s : %c\n", *s);
    printf("valeur de s : %p\n", s);
#if 0
    s = &c;
    printf("caractère *s : %c\n", *s);
    printf("valeur de s : %p\n", s);
#endif
```

```
    return 0;
}
```

Conclusion : les identificateurs de tableaux peuvent être considérés comme des pointeurs qui ne peuvent pas changer de valeur.
Remarque : le deuxième programme n'était pas du tout une bonne idée ! La chaîne "Hello !" est ici perdue, on ne peut plus y accéder ! But ici purement pédagogique, *ne pas reproduire dans un « vrai » programme !*

2.4 Un pointeur particulier : le pointeur `NULL`.

La constante `NULL` vaut `0` mais indique plus clairement qu'on a un *pointeur nul*. Un pointeur nul ne pointe vers aucun emplacement mémoire. Le déréférencement de ce pointeur (`*p`) est illicite. En général si une fonction de la bibliothèque standard retourne un pointeur `NULL`, c'est une information de grande importance.

3 `FILE`, `fopen`, `fprintf`, `fscanf`, `fclose`

On commence par un exemple : le super calculateur de carrés revisité. (Certes, ce programme est inutile, mais on veut ici insister sur les entrées sorties, donc on se contente pour l'instant d'un programme très simple).

```
/* carres-2.c */
#include <stdio.h> /* FILE, fprintf, printf, scanf, fopen, ↩
   fclose */
int main()
{
    int i, n;
    char filename[] = "carres.txt";
    FILE *f = fopen(filename, "w");
    for (i = 0; i < 5 ; i++) {
        printf("entrer nombre numéro %d:\n", i + 1);
        scanf("%d", &n);
        fprintf(f, "%d\n", n * n);
    }
    fclose(f);
    return 0;
}
```

Améliorations :

1. Mettre les résultats sous forme de table.
2. Demander à l'utilisateur le nom du fichier à écrire au lieu de le coder en dur.
3. Arrêter lorsque l'utilisateur entre une lettre.

```
/* carres-3.c */
#include <stdio.h> /* FILE, fprintf, printf, scanf, fopen, ↩
   fclose */
#define LONG_MAX 255
int main()
{
    int n;
    char filename[LONG_MAX];
    FILE *f;
```

```
    /* nom du fichier et ouverture */
    printf("Nom du fichier à écrire ?\n");
    scanf("%s", filename); /* pas besoin de & ici, pourquoi ?? ↩
       */
    printf("Le fichier aura le nom : %s\n", filename);
    f = fopen(filename, "w");

    for(;;) {
        printf("entrer prochain nombre (une lettre arrête le ↩
           programme) :\n");
        /* scanf retourne le nombre de champs remplis */
        if(scanf("%d", &n) == 0) /* mieux : if(!scanf("%d", &n))↩
            */
            break;
        else
            fprintf(f, "%d \t %d\n", n, n * n);
    }
    fclose(f);
    return 0;
}
```

Remarque importante : `fopen("nom_fic","w")` crée le fichier `nom_fic` s'il n'existe pas et *le supprime s'il existe déjà (et qu'il a la permission) !* De nombreux problèmes peuvent survenir lors de l'utilisation de `fopen`, par exemple :

- permissions insuffisantes (on essaie d'écraser un fichier pour lequel on n'a pas le droit `"w"` d'écrire).
- disque plein, espace insuffisant.

Bonne pratique : toujours tester si fopen a réussi. Pour le savoir : si `fopen` échoue, il retourne `NULL`, le pointeur nul.

```
/* carres-4.c */
#include <stdio.h> /* FILE, fprintf, printf, scanf, fopen, ↩
   fclose */
#include <stdlib.h> /* exit, EXIT_FAILURE */
#define LON_MAX 255
int main()
{
    int n;
    char filename[LON_MAX];
    FILE *f;

    /* nom du fichier et ouverture */
    printf("Nom du fichier à écrire ?\n");
    scanf("%s", filename); /* pas besoin de & ici, pourquoi ?? ↩
       */
    printf("Le fichier aura le nom : %s\n", filename);

    if( !(f = fopen(filename, "w")) ) { /* Expliquer */
        /* on écrit l'erreur sur stderr */
        fprintf( stderr, "ouverture du fichier %s en écriture ↩
```

```
            impossible\n",
             filename);
        exit( EXIT_FAILURE );
    }

    for(;;) {
        printf("entrer prochain nombre (une lettre arrête le ↩
           programme) :\n");
        /* scanf retourne le nombre de champs remplis */
        if ( scanf("%d", &n) == 0 ) /* mieux : if ( !scanf("%d",↩
            &n) ) */
             break;
        else
             fprintf(f, "%d \t %d\n", n, n * n);
    }
    fclose(f);
    return 0;
}
```

Avant d'executer le programme avec écriture du programme `a`, on entre les commandes suivantes.

```
$ sudo su
$ touch a
$ ls -lh a
$ exit
```

Message à retenir :

1. La structure `FILE` (déclarée dans `stdio.h`) est volontairement opaque. *Vous ne devez pas vous soucier de son implémentation.* Elle comporte entre autre informations le nom du fichier et la position d'un curseur sur ce fichier (où va se passer la prochaine lecture, écriture).
2. On ne manipule jamais un objet de type `FILE` mais *toujours un pointeur sur celui-ci.*
3. `FILE *fopen( char *filename, char *mode )` : la fonction `fopen` prend deux arguments de type chaîne de caractère terminées par `\0`. Le premier est le nom du fichier et le second le mode d'accès. Modes d'accès les plus utilisés :
   - `"w"` : écriture, crée le fichier s'il n'existe pas, l'écrase s'il existe ;
   - `"r"` : lecture, le fichier doit exister ;
   - `"a"` : « append » (ajouter à la fin).

   En cas de succès, retourne un pointeur non nul sur un objet de type `FILE` associé à ce fichier et à ce mode. En cas d'erreur, la valeur retournée est le pointeur nul : `NULL` et il est de bon ton de toujours tester la nullité de la valeur retournée.
4. Entrée-sortie formatée :
   - `int fprintf( FILE *f, char *format, ... )` comme `printf` mais sur le flot `*f` plutôt que sur la sortie standard. Retourne le nombre de caractères écrits ou `-1` en cas d'erreur. Remarque : `printf(fmt, ...)` est synonyme de `fprintf(stdout, fmt, ...)`.
   - `int fscanf( FILE *f, char *format, ... )` est comme `scanf` mais depuis le flot `f` plutôt que sur depuis l'entrée standard. *Attention, les champs qui suivent le format sont des pointeurs.* Retourne le nombre de champs renseignés. `scanf( fmt, ...)` est synonyme de `fscanf( stdin, fmt, ... )`.
5. `int fclose( FILE *f )` : ferme le flot `f`, fait de la maintenance (écriture du tampon, ...) puis libère les ressources allouées lors du fopen.

Remarque : Ce schéma d'allocation, utilisation via un pointeur puis libération des ressources est très fréquent en C. Presque toutes les bibliothèques l'utilisent.

4 Caractère par caractère : `fgetc`, `fputc`

— `int fgetc(FILE *f)` : retourne le caractère suivant sur `f` ou `EOF` si on a atteint la fin du fichier ou s'il y a une erreur.

— `int fputc(int c, FILE *f)` : écrit le caractère `c` sur le flot `f`. Retourne le caractère écrit ou `EOF` en cas d'erreur.

But du jeu : un programme qui enlève les commentaires d'un programme en C. On fait l'hypothèse, pour simplifier, qu'il n'y a pas de commentaires imbriqués.

On modélise la situation par 4 états : `HORS`, `OBLIQUE`, `DANS`, `ETOILE`.

— Dans l'état `HORS` : Si je lis le caractère `'/'`, je passe dans l'état `OBLIQUE` Sinon, j'y reste et j'imprime le caractère donné.

— Dans l'état `OBLIQUE` : Si je lis le caractère `'*'`, je passe dans l'état `DANS`. Sinon, j'imprime le caractère `'/'` puis le caractère courant et je passe dans l'état `HORS`.

— Dans l'état `DANS` : Si je lis un autre caractère que `'*'`, je l'ignore et je reste dans cet état. Si je lis `'*'`, je passe dans l'état `ETOILE`.

— Dans l'état `ETOILE` : Si je lis `'/'`, je passe dans l'état `HORS`. Si je lis un autre caractère, je passe dans l'état `DANS`.

```c
/* decommenter.c */
/* Enlève les commentaires dans un programme en C. On suppose ↩
   que ce programme
   n'a pas de commentaires imbriqués. */
#include <stdio.h>
#include <stdlib.h>
#define HORS 0
#define OBLIQUE 1
#define DANS 2
#define ETOILE 3
int main()
{
    char source[] = "decommenter.c";
    char but[] = "decommenter_sans_commentaire.c";
    FILE *s, *b; /* s pour source, b pour but */
    int c, etat;

    /* acquisition des ressources */
    if ( !(s = fopen(source, "r")) ) {
```

```
        fprintf(stderr, "Erreur de lecture de la source\n");
        exit( EXIT_FAILURE );
    }
    else if ( !(b = fopen(but, "w")) ) {
        fprintf(stderr, "Erreur de lecture du but\n");
        fclose(s); /* Expliquer */
        exit( EXIT_FAILURE );
    }
    /* utilisation des ressources */
    etat = HORS;
    while ( (c = fgetc(s)) != EOF ){ /* Expliquer */
        if (etat == HORS) {
            if (c == '/')
                etat = OBLIQUE;
            else
                fputc(c, b);
        }
        else if (etat == OBLIQUE) {
            if (c == '*')
                etat = DANS;
            else {
                fputc('/', b);
                fputc(c, b);
                etat = HORS;
            }
        }
        else if (etat == DANS) {
            if (c == '*')
                etat = ETOILE;
        }
        else { /* etat == ETOILE */
            if (c == '/')
                etat = HORS;
            else
                etat = DANS;
        }
    }
    /* libération des ressources */
    fclose(b);
    fclose(s);
    /* bonne habitude à avoir : dans l'ordre inverse des ↩
       acquisitions */
    return 0;
}
```

Message à retenir :

— `while ( (c = fgetc(s))!= EOF )` : ce genre de construction est très fréquente en C, et très pratique car l'affectation et le test doivent être faits au même moment.

— `EOF` vaut `-1` donc il faut que `c` soit de type `int`.

Améliorations possibles :

— Utiliser une construction `switch case` à la place de `if`, `else if`, ...

— un type énuméré (`enum`) à la place des 4 `#define` pour l'état.

— Utiliser les arguments de la ligne de commande pour connaître le nom du fichier en entrée et le nom du fichier en sortie.

— Gérer le cas des commentaires imbriqués.

5 Ligne par ligne : `fgets` et `fputs`

On va ici écrire un `grep` minimal. Le programme `grep` permet de rechercher des chaînes de caractères (et bien plus encore) dans des fichiers.

Ici on veut simplement afficher sur la sortie standard les lignes d'un fichier contenant une chaîne de caractère donnée. On va utiliser les fonctions suivantes :

— `char *fgets(char *s, int n, FILE *f)` : lit des caractères dans le flot `f` et les met dans `s`, avec un `\0` final. S'arrête si `n - 1` caractères ont été lus, ou bien si une fin de ligne est rencontrée. Retourne `s`, ou bien `NULL` si erreur ou fin de fichier.

— `int fputs(char *s, FILE *f)` : écrit la chaîne `s` sur `f` et retourne une valeur positive ou nulle si tout s'est bien passé, `EOF` en cas d'erreur.

— `char *strstr(char *s, char *t)`, déclarée dans `string.h` : retourne un pointeur sur la première occurence de la chaîne `t` dans la chaîne `s`, ou `NULL` si `t` n'est pas dans s.

```c
/* mon_petit_grep.c */
/* grep minimal pour illustrer fgets et fputs */

#include <stdio.h>
#include <stdlib.h>
#include <string.h> /* Pour strstr */

#define LON_MAX 200 /* longueur max d'une ligne : LON_MAX - 1 */

int main() {
    char nom_fic[] = "decommenter.c";
    FILE *f;
    char chaine[] = "==";
    char ligne[LON_MAX];

    if ( !(f = fopen(nom_fic, "r")) ) {
        fprintf(stderr, "erreur : ouverture de %s\n", nom_fic);
        exit( EXIT_FAILURE );
    }
    while ( fgets(ligne, LON_MAX, f) )  /* Expliquer */
        if ( strstr( ligne, chaine ) )
            fputs( ligne, stdout );
    fclose(f);
    return 0;
}
```

On va maintenant utiliser les arguments de la ligne de commande. Ce sont les arguments de la fonction `main`, lorsqu'ils sont présents. Dans ce cas ils doivent impérativement s'écrire :

— `int argc` : le nombre d'arguments.

— `char *argv[]` (ou bien `char **argv`) : tableau de pointeurs vers les arguments.

Si ces arguments sont présents, alors :
`argv[0]` est le nom du programme et se termine par `\0`.
`argv[1]` est le premier argument et se termine par `\0`.
...
`argv[argc - 1]` est le dernier argument et se termine par `\0`.
`argv[argc]` vaut toujours `NULL`.

```c
/* mon_petit_grep-2.c */
/* grep minimal pour illustrer fgets et fputs
 * et les arguments de la ligne de commande */
#include <stdio.h>
#include <stdlib.h>
#include <string.h> /* Pour strstr et strcpy */

#define LON_MAX 200 /* longueur max d'une ligne : LON_MAX - 1 */
#define LON_NOM 255

int main(int argc, char *argv[]) {
    /* argv[0] : nom du programme
     * argv[1] : chaine à chercher
     * argv[2] : fichier dans lequel chercher
     * argv[3] : NULL.
     * argc doit valoir 3
     */
    char nom_fic[255];
    FILE *f;
    char chaine[LON_MAX];
    char ligne[LON_MAX];

    if (argc != 3 ) {
        printf("usage : mon_petit_grep chaine nom_fichier\n");
        exit( EXIT_FAILURE );
    }
    else {
        strcpy( chaine, argv[1] );
        strcpy( nom_fic, argv[2] );
    }

    if ( !(f = fopen(nom_fic, "r")) ) {
        fprintf(stderr, "erreur : ouverture de %s\n", nom_fic);
        exit( EXIT_FAILURE );
    }
    while ( fgets(ligne, LON_MAX, f) )
        if ( strstr( ligne, chaine ) )
            fputs( ligne, stdout );
    fclose(f);
    return 0;
}
```

Améliorations possibles :

1. Tester la longueur des arguments (fonction `strlen`) pour voir s'ils entrent bien dans `chaine` et `nom_fic`.

2. Ajouter des fonctionnalités (voir `grep` et les expressions régulières).