Manipulation des fichiers en C

par Jessee Michaël C. Edouard (Accueil)

Date de publication : 07 avril 2008

Dernière mise à jour : 09 janvier 2010

La manipulation des fichiers en langage C est relativement simple mais nécessite une bonne compréhension des principes qui sont à leur base. C'est ce que ce tutoriel va tenter de vous expliquer.
Commentez cet article :

I - Généralités 3
I-A - La notion de flux 3
I-B - Les fichiers sur disque 3
I-C - Les messages d'erreur 3
I-D - Ouverture d'un fichier 4
I-E - Exemple : copier un fichier 5
II - Les erreurs d'E/S 6
III - Positionnement dans un fichier 7
La fonction fseek 7
La fonction ftell 8
Les fonctions fgetpos et fsetpos 8
La fonction rewind 8
Position courante 8
IV - Le traitement par blocs 8
V - Opérations sur les fichiers 9
V-A - Renommer ou déplacer un fichier 9
V-B - Supprimer un fichier 9
VI - Les fichiers temporaires 9
VI - Rediriger un flux d'E/S 10
VIII - Exercices 10
VIII-A - Enregistrement et chargement de données 10
VIII-B - Taille d'un fichier 10
VIII-C - Découpage et restitution d'un fichier 10
VIII-D - Chiffrement et déchiffrement 10
IX - Solutions des exercices 11
IX-A - Enregistrement et chargement de données (VIII-A) 11
IX-B - Taille d'un fichier (VIII-B) 11
IX-C - Découpage et restitution d'un fichier (VIII-C) 12
IX-D - Chiffrement et déchiffrement (VIII-D) 17
X - Remerciements 18

I - Généralités

I-A - La notion de flux

Les **entrées/sorties** (**E/S**) ne font pas partie du langage C car ces opérations sont dépendantes du système. Néanmoins puisqu'il s'agit de tâches habituelles, sa bibliothèque standard est fournie avec des fonctions permettant de réaliser ces opérations de manière portable. Ces fonctions sont principalement déclarées dans le fichier **stdio.h**.

Certaines ont déjà été présentées dans les tutoriels précédents (notamment ici), de même que quelques concepts relatifs aux entrées/sorties en langage C. Aucun rappel ne sera fait, sauf sur certains concepts jugés importants.
Les entrées/sorties en langage C se font par l'intermédiaire d'entités logiques, appelés **flux**, qui représentent des objets externes au programme, appelés **fichiers**. En langage C, un fichier n'est donc pas nécessairement un fichier sur disque (ou périphérique de stockage pour être plus précis). Un fichier désigne en fait aussi bien un fichier sur disque (évidemment) qu'un périphérique physique ou un tube par exemple. Selon la manière dont on veut réaliser les opérations d'E/S sur le fichier, qui se font à travers un flux, on distingue deux grandes catégories de flux à savoir les **flux de texte** et **les flux binaires**.
Les flux de texte sont parfaits pour manipuler des données présentées sous forme de texte. Un flux de texte est organisé en **lignes**. En langage C, une ligne est une suite de caractères terminée par le **caractère de fin de ligne** (inclus) : '\n'. Malheureusement, ce n'est pas forcément le cas pour le système sous-jacent. Sous Windows par exemple, la **marque de fin de ligne** est par défaut la combinaison de deux caractères : **CR** (Carriage Return) et **LF** (Line Feed) soit '\r' et '\n' (notez bien que c'est **CR/LF** c'est-à-dire CR suivi de LF pas LF suivi de CR). Sous UNIX, c'est tout simplement '\n'. On se demande alors comment on va pouvoir lire ou écrire dans un fichier, à travers un flux de texte, de manière portable. Et bien c'est beaucoup plus simple que ce à quoi vous-vous attendiez : lorsqu'on effectue une opération d'entrée/sortie sur un flux de texte, les données seront lues/écrites de façon à ce qu'elles correspondent à la manière dont elles doivent être représentées et non caractère pour caractère. C'est-à-dire par exemple que, dans une implémentation où la fin de ligne est provoquée par la combinaison des caractères CR et LF, l'écriture de '\n' sur un flux de texte va provoquer l'écriture effective des caractères '\r' et '\n' dans le fichier associé. Sur un flux binaire les données sont lues ou écrites dans le fichier caractère pour caractère.

I-B - Les fichiers sur disque

La communication avec une ressource externe (un modem, une imprimante, une console, un périphérique de stockage, etc.) nécessite un protocole de communication (qui peut être texte ou binaire, simple ou complexe, ...) spécifique de cette ressource et qui n'a donc rien à voir le langage C. La norme définit tout simplement les fonctions permettant d'effectuer les entrées/sorties vers un fichier sans pour autant définir la notion de matériel ou même de fichier sur disque afin de garantir la portabilité du langage (ou plus précisément de la bibliothèque). Cependant, afin de nous fixer les idées, nous n'hésiterons pas à faire appel à ces notions dans les explications.
Les fichiers sur disque servent à stocker des informations. On peut « naviguer » à l'intérieur d'un tel fichier à l'aide de **fonctions de positionnement** (que nous verrons un peu plus loin). A chaque instant, un « pointeur » indique la **position courante** dans le fichier. Ce pointeur se déplace, à quelques exceptions près, après chaque opération de lecture, d'écriture ou appel d'une fonction de positionnement par exemple. Bien entendu, cette notion de position n'est pas une spécificité exclusive des fichiers sur disque. D'autres types de fichiers peuvent très bien avoir une structure similaire.

I-C - Les messages d'erreur

En langage C, il est coutume de retourner un entier même quand une fonction n'est censée retourner aucune valeur. Cela permet au programme de contrôler les erreurs et même d'en connaître la cause. Certaines fonctions comme malloc par exemple retournent cependant une adresse, NULL en cas d'erreur. Pour fournir d'amples informations quant à la cause de l'erreur, ces fonctions placent alors une valeur dans une variable globale appelée **errno** (en fait la norme n'impose pas qu'errno doit être forcément une variable globale !), cette valeur bien entendu est à priori dépendante de l'implémentation. Ces valeurs doivent être déclarées dans le fichier **errno.h**. Par exemple, une implémentation peut définir un numéro d'erreur ENOMEM qui sera placée dans errno lorsqu'un appel à malloc a échoué car le système manque de mémoire.

La fonction **strerror**, déclarée dans **string.h**, permet de récupérer une chaîne à priori définie par l'implémentation décrivant l'erreur correspondant au numéro d'erreur passé en argument. La fonction **perror**, déclarée dans **stdio.h**, quant à elle utilise cette chaîne, plus une chaîne fournie en argument, pour afficher la description d'une erreur. Le texte sera affiché sur l'erreur standard (stderr). Pour résumer :

```
perror(msg);
```

est équivalent à :

```
fprintf(stderr, "%s: %s\n", msg, strerror(errno));
```

Si nous n'avons pas parlé de ces fonctions auparavant, c'est parce que nous n'en avions pas vraiment jusqu'ici besoin. A partir de maintenant, elles vont être très utiles car les fonctions d'entrées/sorties sont sujettes à de très nombreuses sortes d'erreurs : fichier non trouvé, espace sur disque insuffisant, on n'a pas les permissions nécessaires pour lire ou écrire dans le fichier, ...

I-D - Ouverture d'un fichier

Toute opération d'E/S dans un fichier commence par l'**ouverture** du fichier. Lorsqu'un fichier est ouvert, un **flux** lui est associé. Ce flux est représenté par un pointeur vers un objet de type **FILE**.
Pendant l'ouverture d'un fichier, on doit spécifier comment en désire l'ouvrir : en lecture (c'est-à-dire qu'on va lire dans le fichier), en écriture (pour écrire), ou en lecture et écriture. La fonction **fopen** :

```
FILE * fopen(const char * filename, const char * mode);
```

permet d'ouvrir un fichier dont le nom est spécifié par l'argument filename selon le mode, spécifié par l'argument mode, dans lequel on souhaite ouvrir le fichier. Elle retourne l'adresse d'un objet de type FILE qui représente le fichier à l'intérieur du programme. En cas d'erreur, NULL est retourné et une valeur indiquant la cause de l'erreur est placée dans errno.
En pratique, dans le cas d'un fichier sur disque, l'argument nom peut être aussi bien un chemin complet qu'un chemin relatif. Notez bien que les notions de chemin, répertoire (qui inclut également les notions de répertoire courant, parent, etc.), ... sont dépendantes du système, elles ne font pas partie du langage C. La plupart du temps, le répertoire courant est par défaut celui dans lequel se trouve le programme mais, si le système le permet, il est possible de spécifier un répertoire différent.
Fondamentalement, les valeurs suivantes peuvent être utilisées dans l'argument mode :

- **"r"** : ouvrir le fichier en **lecture**. Le fichier spécifié doit déjà exister.
- **"w"** : ouvrir le fichier en **écriture**. S'il n'existe pas, il sera créé. S'il existe déjà, son ancien contenu sera effacé.
- **"a"** : ouvrir le fichier en mode **ajout**, qui est un mode dans lequel toutes les opérations d'écriture dans le fichier se feront à la fin du fichier. S'il n'existe pas, il sera créé.

Quel que soit le cas, le fichier sera associé à un flux de texte. Pour spécifier qu'on veut ouvrir le fichier en tant que fichier binaire, il suffit d'ajouter le suffixe 'b' (c'est-à-dire **"rb"**, **"wb"** ou **"ab"**).
Sauf dans le cas du mode "ab" et dérivés, dans lequel le pointeur pourrait, selon l'implémentation, être positionné à la fin du fichier, il sera positionné au début du fichier.
On peut également ajouter un '+' dans l'argument mode. Par exemple : "r+", "r+b" ou encore "rb+" mais "r+b" et "rb +", c'est évidemment la même chose. La présence du '+' aura pour effet de permettre d'effectuer aussi bien des opérations de lecture que d'écriture sur le fichier. On dit alors que le fichier est ouvert en mode **mise à jour**. Il y a cependant une remarque très importante concernant les fichiers ouverts en mode mise à jour :

- avant d'effectuer une opération de lecture juste après une opération d'écriture, il faut tout d'abord appeler fflush ou une fonction de positionnement
- avant d'effectuer une opération d'écriture juste après une opération de lecture, il faut d'abord appeler une fonction de positionnement, à moins d'avoir atteint la fin du fichier

- dans de nombreuses implémentations, tout fichier ouvert en mode mise à jour est supposé être un fichier binaire qu'on ait mis oui ou non la lettre 'b'. Personnellement, je recommande donc de n'utiliser le mode mise à jour qu'avec les fichiers binaires.

Lorsqu'on n'en a plus besoin, il faut ensuite **fermer** le fichier :

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

Le programme suivant crée un fichier, hello.txt, pour y écrire ensuite une et une seule ligne : Hello, world.

```
#include <stdio.h>

int main()
{
    FILE * f;

    f = fopen("hello.txt", "w");
    if (f != NULL)
    {
        fprintf(f, "Hello, world\n");
        fclose(f);
    }
    else
        perror("hello.txt");

    return 0;
}
```

En supposant que pour le système la fin de ligne est représentée par la combinaison des caractères CR et LF, ce programme est aussi équivalent au suivant :

```
#include <stdio.h>

int main()
{
    FILE * f;

    f = fopen("hello.txt", "wb");
    if (f != NULL)
    {
        fprintf(f, "Hello, world\r\n");
        fclose(f);
    }
    else
        perror("hello.txt");

    return 0;
}
```

I-E - Exemple : copier un fichier

Il y a plusieurs moyens de réaliser la copie d'un fichier, le plus simple est peut-être de copier la source vers la destination octet par octet. Voici un programme qui réalise une telle copie :

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

char * saisir_chaine(char * lpBuffer, int buf_size);

int main()
{
    char src[FILENAME_MAX]; /* FILENAME_MAX est defini dans stdio.h */
    FILE * fsrc;
```

```c
    printf("Ce programme permet de copier un fichier.\n");
    printf("source : ");
    saisir_chaine(src, sizeof(src));

    fsrc = fopen(src, "rb");
    if (fsrc == NULL)
        perror(src);
    else
    {
        char dest[FILENAME_MAX];
        FILE * fdest;

        printf("dest : ");
        saisir_chaine(dest, sizeof(dest));

        if (strcmp(src, dest) == 0)
            printf("La source ne peut pas etre en meme temps la destination.\n");
        else
        {
            fdest = fopen(dest, "wb");
            if (fdest == NULL)
                perror(dest);
            else
            {
                int c;

                while ((c = getc(fsrc)) != EOF)
                    putc(c, fdest);

                fclose(fdest);
                printf("Copie terminee.\n");
            }
        }

        fclose(fsrc);
    }

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * ret = fgets(lpBuffer, buf_size, stdin);

    if (ret != NULL)
    {
        char * p = strchr(lpBuffer, '\n');
        if (p != NULL)
            *p = '\0';
        else
        {
            int c;

            do
                c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}
```

II - Les erreurs d'E/S

Une **erreur d'E/S** est erreur qui peut se produire lors d'une tentative d'opération d'E/S, par exemple : fin de fichier atteinte, tentative d'écriture dans un fichier ouvert uniquement en lecture, tentative de lecture dans un fichier ouvert uniquement en lecture, etc.

La fonction **feof** :

```
int feof(FILE * f);
```

peut être appelé à n'importe quel moment pour connaître si l'on a atteint la fin du fichier. Je rappelle qu'un programme (du moins d'après les fonctions du C) ne peut affirmer que la fin d'un fichier a été atteinte qu'après avoir tenté de lire dans le fichier alors qu'on se trouve déjà à la fin c'est-à-dire derrière le dernier octet, pas juste après avoir lu le dernier octet, donc faites gaffe. Evidemment, une telle fonction ne sera utilisée que sur un fichier ouvert en lecture. Elle retourne VRAI si la fin de fichier a été atteinte et FAUX dans le cas contraire. Cette information est en fait maintenue par un **indicateur de fin de fichier** qui indique à tout moment si on a oui ou non déjà atteint la fin du fichier. Après un appel fructueux à une fonction de positionnement, l'indicateur de fin de fichier est remis à zéro.
La fonction **ferror** :

```
int ferror(FILE * f);
```

permet de connaître si une erreur différente de "fin de fichier atteinte" s'est produite lors de la dernière opération d'E/S sur f. Cette information est maintenue en permanence par un **indicateur d'erreur**. Bien qu'une lecture à la fin d'un fichier soit également une erreur, elle n'est pas considérée par la fonction ferror car les indicateurs d'erreur et de fin de fichier sont des données bien distinctes.
Attention, une fois qu'une erreur s'est produite, l'indicateur d'erreur ne sera remis à zéro qu'après un appel à **clearerr**.

```
void clearerr(FILE * f);
```

Cette fonction remet à zéro l'indicateur d'erreur du fichier f.

III - Positionnement dans un fichier

La fonction fseek

```
int fseek(FILE * f, long offset, int origin);
```

Permet de modifier la position courante dans un fichier. La nouvelle position dépend des valeurs des arguments offset qui représente le déplacement et origin qui représente l'origine. Les valeurs qu'on peut donner à origin sont :

- **SEEK_SET** : le pointeur sera ramené à offset caractères par rapport au début du fichier
- **SEEK_CUR** : le pointeur sera ramené à offset caractères par rapport à la position courante
- **SEEK_END** : le pointeur sera ramené à offset caractères par rapport à la fin du fichier.

Bien entendu, le déplacement peut être positif ou négatif. Si la fonction réussit, 0 est retourné (une valeur différente de zéro indique donc une erreur).
L'utilisation de cette fonction avec un flux de texte peut donner des résultats inattendus à cause du coup du '\n' ...
Pour aller au n-ième caractère d'un fichier associé à un flux de texte (ce qui entraîne que la fin de ligne sera vue du programme comme un et un seul caractère : '\n') par exemple, utilisez la méthode suivante :

```
fseek(f, 0L, SEEK_SET);
for(i = 0; i < n; i++)
    getc(f);

/* On est maintenant a la position n dans le fichier */
```

Cette technique est aussi connue sous le nom d'**accès séquentiel**, par opposition à **accès aléatoire** (utilisant fseek, etc.).

Dans la pratique, on ne sera jamais confronté à un tel cas. Bon, « jamais » c'est peut être un peu trop mais en tout cas je peux vous affirmer que je n'ai jamais eu à le faire. Si on doit faire beaucoup de « va et vient » dans un fichier, c'est tout simplement une erreur de l'avoir ouvert en tant que fichier texte.

La fonction ftell

```
long ftell(FILE * f);
```

Permet de connaître la position courante dans le fichier. Dans le cas où le fichier est associé à un flux binaire, il s'agit du nombre de caractères entre le début du fichier et la position courante. Si le fichier est associé à un flux de texte, la valeur retournée par cette fonction est tout simplement une information représentant la position actuelle dans le fichier et on ne peut rien dire de plus. Quel que soit le cas, le retour de ftell pourra éventuellement être utilisée dans un futur appel fseek avec SEEK_SET pour revenir à la même position.

Les fonctions fgetpos et fsetpos

La fonction **fgetpos** permet de sauvegarder la position courante dans un fichier. On pourra ensuite ultérieurement revenir à cette position à l'aide de la fonction **fsetpos**.

```
int fgetpos(FILE * f, fpos_t * p_pos);
int fsetpos(FILE * f, const fpos_t * p_pos);
```

Le type **fpos_t** est un type limité à usage de ces fonctions. On ne doit passer à fsetpos qu'une valeur obtenue à l'aide de fgetpos. En cas de succès, 0 est retourné.

La fonction rewind

Cette fonction permet de « rembobiner » un fichier.

```
rewind(f);
```

est équivalent à :

```
fseek(f, 0L, SEEK_SET);
clearerr(f);
```

Position courante

En général, les opérations de lecture (respectivement d'écriture) commencent la lecture (respectivement l'écriture) à partir de la position courante puis ensuite déplacent le pointeur selon le nombre de caractères lus (respectivement écrits). Il existe toutefois des exceptions comme dans le cas ou le fichier est ouvert en mode ajout par exemple, auquel cas toutes les opérations d'écriture se feront à la fin du fichier, indépendamment de la position courante.

IV - Le traitement par blocs

La bibliothèque standard offre également des fonctions permettant de réaliser des opérations d'entrées/sorties par **blocs** d'octets. Il s'agit des fonctions **fread** et **fwrite**.

```
size_t fread(void * buffer, size_t size, size_t nobj, FILE * f);
size_t fwrite(const void * buffer, size_t size, size_t nobj, FILE * f);
```

fread lit nobj objets de taille size chacune à partir de la position courante dans f pour les placer dans un buffer. Elle retourne ensuite le nombre d'objets effectivement lus.
A l'inverse fwrite écrit nobj objets de buffer de taille size chacune en destination de f. Elle retourne ensuite le nombre d'objets effectivement écrits.
Par exemple, voici une manière d'écrire "Bonjour" dans un fichier en utilisant fwrite.

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

int main()
{
    FILE * f;
    char s[] = "Bonjour";

    f = fopen("bonjour.txt", "wb");

    if (f != NULL)
    {
        fwrite(s, sizeof(char), strlen(s), f);
        fclose(f);
    }
    else
        perror("bonjour.txt");

    return 0;
}
```

V - Opérations sur les fichiers

V-A - Renommer ou déplacer un fichier

La fonction **rename** :

```c
int rename(const char * oldname, const char * newname);
```

permet de renommer un fichier lorsque cela est possible. Si la fonction réussit, 0 est retourné.

V-B - Supprimer un fichier

La fonction **remove** :

```c
int remove(const char * filename);
```

permet de supprimer un fichier. Si la fonction réussit, 0 est retourné.

VI - Les fichiers temporaires

Un **fichier temporaire** est un fichier utilisé par un programme puis supprimé lorsque celui-ci se termine. La fonction **tmpfile** :

```c
FILE * tmpfile(void);
```

permet de créer un fichier en mode "wb+" qui sera automatiquement supprimé à la fin du programme.
On peut toujours bien sûr créer un fichier temporaire « manuellement ». Dans ce cas, il vaut mieux lui donner un nom généré par **tmpnam** afin de s'assurer qu'aucun autre fichier porte déjà ce nom.

```c
char * tmpnam(char * name);
```

Si name est utilisé, il doit pointer sur un buffer d'au moins **L_tmpnam** octets autrement il est possible qu'elle ne pourra pas contenir le nom généré par cette fonction ce qui va provoquer un débordement de tampon ! On peut tout simplement passer NULL à cette fonction si on n'a pas de buffer à passer en argument.

VI - Rediriger un flux d'E/S

La fonction **freopen** :

```
FILE * freopen(const char * filename, const char * mode, FILE * f);
```

ferme le fichier associé au flux f, ouvre le fichier dont le nom est spécifié par filename selon le mode spécifié par mode en lui associant le flux représenté par f puis retourne f ou NULL si une erreur s'est produite. Dans le programme suivant, la sortie standard est redirigée vers le fichier out.txt.

```
#include <stdio.h>

int main()
{
    if (freopen("out.txt", "w", stdout) != NULL)
    {
        printf("Hello, world\n");
        fclose(stdout);
    }
    else
        perror("out.txt");

    return 0;
}
```

VIII - Exercices

VIII-A - Enregistrement et chargement de données

Ecrire un programme qui écrit le contenu entier d'un tableau de 10 entiers dans un fichier puis écrire un autre programme qui lit les dix entiers du fichier puis les affiche sur la sortie standard.

VIII-B - Taille d'un fichier

Ecrire un programme qui affiche la longueur d'un fichier dont le nom sera fourni par l'utilisateur (astuce : utiliser fseek/ftell).

VIII-C - Découpage et restitution d'un fichier

Ecrire un programme qui permet de découper un fichier en un ou plusieurs morceaux puis écrire un autre programme qui permet de restituer le fichier original à l'aide des découpes. Une autre possibilité est d'écrire un seul programme avec menu qui permet aussi bien le découpage que la restitution.

VIII-D - Chiffrement et déchiffrement

Chiffrer une information c'est modifier cette information de sorte que l'original ne puisse être restitué que par quelqu'un qui connaît les modifications que celui-ci a subi. La restitution de l'information originale à partir de l'information chiffrée s'appelle le déchiffrement. La cryptographie, branche des mathématiques qui s'intéresse aux techniques de chiffrement, a des applications fondamentales dans la sécurité informatique d'aujourd'hui.

Pour chiffrer une information x, il suffit de trouver une fonction bijective f, transformer x par f(x) et ne communiquer f qu'aux personnes dignes de confiance. Pour restituer x, il suffit évidemment d'appeler $f^{-1}(x)$. La grande problématique de la cryptographie est de trouver une fonction f qui ne soit pas facilement devinable par les gens de l'extérieur.
Trouver une méthode simple permettant de chiffrer un fichier et écrire un programme de chiffrement et de déchiffrement de fichier en utilisant cette méthode.

IX - Solutions des exercices

IX-A - Enregistrement et chargement de données (VIII-A)

Ecriture du tableau

```c
#include <stdio.h>

int main()
{
    FILE * f = fopen("test.tab", "wb");
    if (f == NULL)
        perror("test.tab");
    else
    {
        int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        fwrite(tab, sizeof(tab[0]), sizeof(tab) / sizeof(tab[0]), f);
        fclose(f);
        printf("Termine.\n");
    }

    return 0;
}
```

Chargement du tableau

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

int main()
{
    FILE * f = fopen("test.tab", "rb");
    if (f == NULL)
        perror("test.tab");
    else
    {
        int n, i, tab[10];
        n = fread(tab, sizeof(tab[0]), sizeof(tab) / sizeof(tab[0]), f);
        for(i = 0; i < n; i++)
            printf("%d\n", tab[i]);
        fclose(f);
    }

    return 0;
}
```

IX-B - Taille d'un fichier (VIII-B)

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

char * saisir_chaine(char * lpBuffer, int buf_size);
long filesize(FILE * f);

int main()
{
    char name[FILENAME_MAX];
    FILE * f;

    printf("Ce programme permet de determiner la taille d'un fichier.\n");
```

```
    printf("Entrez le nom du fichier : ");
    saisir_chaine(name, sizeof(name));

    f = fopen(name, "rb");
    if (f != NULL)
    {
        printf("La taille du fichier est : %ld bytes\n", filesize(f));
        fclose(f);
    }
    else
        perror(name);

    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * ret = fgets(lpBuffer, buf_size, stdin);

    if (ret != NULL)
    {
        char * p = strchr(lpBuffer, '\n');
        if (p != NULL)
            *p = '\0';
        else
        {
            int c;

            do
                c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}

long filesize(FILE * f)
{
    long pos, size;

    pos = ftell(f); /* lire la position courante */
    fseek(f, 0, SEEK_END); /* aller a la fin du fichier */
    size = ftell(f); /* lire l'offset de la nouvelle position */
    fseek(f, pos, SEEK_SET); /* revenir a la position de depart */

    return size;
}
```

IX-C - Découpage et restitution d'un fichier (VIII-C)

Découpage

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

#define MAX_BUFSIZE 4096

char * saisir_chaine(char * lpBuffer, int buf_size);
long filesize(FILE * f);
void split_file(const char * szFile, FILE * f, size_t fsize, size_t bytes_max);

int main()
{
    FILE * f;
    char szFile[FILENAME_MAX - 3];

    printf("Entrez le nom du fichier a decouper : ");
```

Découpage

```
    saisir_chaine(szFile, sizeof(szFile));
    f = fopen(szFile, "rb");

    if (f == NULL)
        perror(szFile);
    else
    {
        long fsize = filesize(f);

        if (fsize < 0)
            fprintf(stderr, "Une erreur s'est produite lors de la lecture de la taille du fichier.\n");
        else
        {
            double mbytes_max; /* taille maximale de chaque morceau en Mo */

            printf("Entrez la taille maximale autorisee pour chaque morceau (en Mo) : ");
            scanf("%lf", &mbytes_max);

            if (mbytes_max > 0)
            {
                size_t bytes_max; /* taille maximale de chaque morceau en o */
                long nb_morceaux;

                bytes_max = (size_t)ceil(mbytes_max * 1024 * 1024);
                nb_morceaux = fsize / bytes_max + (fsize % bytes_max != 0);

                if (nb_morceaux >= 100)
                    fprintf(stderr, "Cette valeur est trop petite par rapport au fichier.\n");
                else
                {
                    printf("Le fichier va etre decoupe en %ld morceaux.\n", nb_morceaux);
                    split_file(szFile, f, fsize, bytes_max);
                }
            }
        }

        fclose(f);
    }

    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * ret = fgets(lpBuffer, buf_size, stdin);

    if (ret != NULL)
    {
        char * p = strchr(lpBuffer, '\n');
        if (p != NULL)
            *p = '\0';
        else
        {
            int c;

            do
                c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}

long filesize(FILE * f)
{
    long pos, size;

    pos = ftell(f);
    fseek(f, 0, SEEK_END);
```

Découpage

```
    size = ftell(f);
    fseek(f, pos, SEEK_SET);

    return size;
}

void split_file(const char * szFile, FILE * f, size_t fsize, size_t bytes_max)
{
    size_t bufsize = bytes_max < MAX_BUFSIZE ? bytes_max : MAX_BUFSIZE;
    char * lpBuffer = malloc(bufsize);

    if (lpBuffer == NULL)
        fprintf(stderr, "Impossible d'allouer la memoire necessaire.\n");
    else
    {
        int i = 1, success = 1; /* i : numero de session */
        size_t total_to_write = fsize, total_written = 0;

        do
        {
            char out_name[FILENAME_MAX];
            FILE * f_out;

            sprintf(out_name, "%s.%02d", szFile, i);

            f_out = fopen(out_name, "wb");

            if (f_out == NULL)
            {
                perror(out_name);
                success = 0;
            }
            else
            {
                size_t session_max, session_to_write, session_written;

                session_max = total_to_write >= bytes_max ? bytes_max : total_to_write;
                session_to_write = session_max;
                session_written = 0;

                while (success && session_to_write > 0)
                {
                    size_t bytes_read, bytes_written;
                    double progression = (100.0 * session_written)/session_max;

                    printf("\rEcriture de %s : %.0f%% effectue", out_name, progression);
                    fflush(stdout);
                    bytes_read = fread(lpBuffer, 1, bufsize, f);

                    if (ferror(f))
                    {
                        putchar('\n');
                        perror(szFile);
                        success = 0;
                    }
                    else
                    {
                        bytes_written = fwrite(lpBuffer, 1, bytes_read, f_out);
                        session_written += bytes_written;
                        total_written += bytes_written;
                        session_to_write -= bytes_written;
                        total_to_write -= bytes_written;

                        if (ferror(f_out))
                        {
                            putchar('\n');
                            perror(out_name);
                            success = 0;
                        }
                    }
                }
```

Découpage

```
                fclose(f_out);
                if (success)
                {
                    printf("\rEcriture de %s : succes.      \n", out_name);
                    i++;
                }
            }
        }
        while (success && total_written != fsize);

        free(lpBuffer);

        if (success)
            printf("Termine.\n");
    }
}
```

Restitution

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

#define BUF_SIZE 4096

char * saisir_chaine(char * lpBuffer, int buf_size);
long filesize(FILE * f);

char lpBuffer[BUF_SIZE];

int main()
{
    FILE * f;
    char szFile[FILENAME_MAX - 3];

    printf("Entrez le nom du fichier a reconstituer : ");
    saisir_chaine(szFile, sizeof(szFile));
    f = fopen(szFile, "wb");

    if (f == NULL)
        perror(szFile);
    {
        FILE * f_part;
        char part_name[FILENAME_MAX];
        int i = 1, success = 1;

        do
        {
            sprintf(part_name, "%s.%02d", szFile, i);

            f_part = fopen(part_name, "rb");

            if (f_part == NULL)
                perror(part_name);
            else
            {
                long partsize = filesize(f_part);

                if (partsize < 0)
                    printf("Une erreur s'est produite lors de la lecture de la taille du fichier.\n");
                else
                {
                    size_t bytes_read, session_bytes_written = 0;

                    do
                    {
                        double progression = (100.0 * session_bytes_written) / partsize;

                        printf("\rCopie de %s : %.0f%% effectue", part_name, progression);
                        bytes_read = fread(lpBuffer, 1, BUF_SIZE, f_part);
                        if (ferror(f_part))
```

Restitution

```
                        {
                            putchar('\n');
                            perror(part_name);
                            success = 0;
                        }
                        else
                        {
                            session_bytes_written += fwrite(lpBuffer, 1, bytes_read, f);
                            if (ferror(f))
                            {
                                putchar('\n');
                                perror(szFile);
                                success = 0;
                            }
                        }
                    }
                    while (success && bytes_read == BUF_SIZE);

                    fclose(f_part);
                    if (success)
                    {
                        printf("\rCopie de %s : succes.      \n", part_name);
                        i++;
                    }
                }
            }
        }
        while (lpBuffer != NULL && f_part != NULL);

        fclose(f);

        if (success)
            printf("Termine.\n");
    }

    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * ret = fgets(lpBuffer, buf_size, stdin);

    if (ret != NULL)
    {
        char * p = strchr(lpBuffer, '\n');
        if (p != NULL)
            *p = '\0';
        else
        {
            int c;

            do
                c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}

long filesize(FILE * f)
{
    long pos, size;

    pos = ftell(f);
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, pos, SEEK_SET);

    return size;
```

Restitution

```
}
```

IX-D - Chiffrement et déchiffrement (VIII-D)

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

char * saisir_chaine(char * lpBuffer, int buf_size);
unsigned char f1(unsigned char c);
unsigned char f1_inv(unsigned char c);

int main()
{
    char choix[2];

    printf("Tapez 1 pour chiffrer et 2 pour dechiffrer ");
    saisir_chaine(choix, sizeof(choix));

    if (*choix != '1' && *choix != '2')
        fprintf(stderr, "Cette valeur est invalide.\n");
    else
    {
        char src[FILENAME_MAX];
        FILE * fsrc;

        printf("Entrez le nom du fichier a %s : ", *choix == '1' ? "chiffrer" : "dechiffrer");
        saisir_chaine(src, sizeof(src));

        fsrc = fopen(src, "rb");
        if (fsrc == NULL)
            perror(src);
        else
        {
            char dest[FILENAME_MAX];
            FILE * fdest;

            printf("Entrez le nom du fichier de destination : ");
            saisir_chaine(dest, sizeof(dest));

            if (strcmp(src, dest) == 0)
                printf("La source ne peut pas etre en meme temps la destination.\n");
            else
            {
                fdest = fopen(dest, "wb");
                if (fdest == NULL)
                    perror(dest);
                else
                {
                    /* Notre technique : chiffrement/dechiffrement octet par octet */

                    int c;
                    unsigned char (*f)(unsigned char) = *choix == '1' ? f1 : f1_inv;

                    printf("Traitement en cours ...");
                    fflush(stdout);

                    while ((c = getc(fsrc)) != EOF)
                        putc(f(c), fdest);

                    fclose(fdest);
                    printf("\nTermine.\n");
                }
            }

            fclose(fsrc);
        }
    }
```

```
    return 0;
}

char * saisir_chaine(char * lpBuffer, int buf_size)
{
    char * ret = fgets(lpBuffer, buf_size, stdin);

    if (ret != NULL)
    {
        char * p = strchr(lpBuffer, '\n');
        if (p != NULL)
            *p = '\0';
        else
        {
            int c;

            do
                c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}

unsigned char f1(unsigned char c) /* Notre fonction de chiffrement */
{
    /* Principe : on fait decaler de maniere circulaire tous les bits de un pas vers la gauche */
    unsigned char msb = (c & 0x80) >> 7; /* Le bit le plus a gauche */
    c <<= 1;
    c |= msb;
    return c;
}

unsigned char f1_inv(unsigned char c) /* Notre fonction de dechiffrement */
{
    unsigned char lsb = c & 1; /* Le bit le plus a droite */
    c >>= 1;
    c |= (lsb << 7);
    return c;
}
```

X - Remerciements

Merci à **ram-0000** pour sa relecture de cet article.