Archivos en C

En este apunte se verá una referencia de las funciones y conceptos de archivos usado en C, resaltando algunas peculiaridades que no se ven en otros lenguajes. Pero de ninguna manera pretende ser un apunte completo sobre el uso de archivos en general y se asume cierta experiencia al respecto.

Una de las peculiaridades de C es que, todos los programas al ejecutarse ya tienen tres archivos abiertos, estos son: la entrada estándar (*stdin*), salida estándar (*stdout*) y salida de error (*stderr*). Los primeros dos son los que usan las funciones de entrada y salida del usuario, como `scanf` y `printf`, respectivamente. El tercero es un archivo de salida destinado al envío de errores de ejecución y por omisión saldrán en la misma salida que los de salida externa.

Siendo que lo que tiene que ver con archivos es normalmente entrada o salida del programa, las funciones listadas en este apunte están declaradas en el encabezado `<stdio.h>`.

1. Entrada y salida de una terminal

La entrada y salida de una terminal en C se comporta de una forma similar a la lectura y escritura de archivos, por lo que se listan a continuación algunas de las funciones de entrada y de salida.

Tanto la entrada como la salida estándar suelen tener un *buffer*, es decir una memoria intermedia, en este caso por líneas, por lo que al intentar leer de entrada estándar mediante `scanf`, el programa se quedará esperando hasta que se termine una línea en la entrada, aún si sólo se quiere leer un caracter. El resto de línea no procesada será la entrada de las siguientes llamadas a las funciones de entrada.

Algo similar sucede con la salida por consola, cuando se utiliza `printf`, la salida suele tener también un *buffer* orientado a líneas, por lo que hasta que no se termine una línea, la salida no se emite en la terminal.

1.1. Manejo de caracteres de a uno

Sin embargo, no es la única opción. Existen otras funciones como:

```
int getchar(void);
```

Esta función permite leer un único caracter desde la entrada estándar, devuelve el valor del caracter leído o, en caso de haberse terminado la entrada, el valor especial `EOF`.

De la misma forma, para emitir un único caracter por la terminal:

```
int putchar(int c);
```

Esta función permite escribir un caracter en la terminal, devuelve el valor del caracter escrito o bien `EOF` en caso de error.

2. Abrir archivos

Para abrir un archivo en C se utiliza la función `fopen`[1], cuyo prototipo es:

```
FILE *fopen(const char *ruta, const char *modo);
```

El primer parámetro es el nombre del archivo, y el segundo el modo de apertura, que puede ser:

- `r` Sólo lectura, se posiciona al principio del archivo.
- `r+` Lectura y escritura, se posiciona al principio del archivo.
- `w` Borra el contenido del archivo o crea uno nuevo, sólo escritura, se posiciona al principio del archivo.
- `w+` Borra el contenido del archivo o crea uno nuevo, lectura y escritura, se posiciona al principio del archivo.
- `a` Abre para añadir (escribir al final del archivo). El archivo se crea si no existe. Se posiciona al final del archivo.
- `a+` Abre para leer y añadir (escribir al final del archivo). El archivo se crea si no existe. Se posiciona al final del archivo.

Además, el archivo puede abrirse en modo *archivo de texto* (por omisión) o en modo *archivo binario* (agregándole una `b` al modo). Los archivos de texto tienen un tratamiento especial para el caracter fin de línea, mientras que con los archivos binarios se accede a los datos en crudo.

El valor devuelto por `fopen` es un puntero de tipo `FILE` que representa a los archivos en la biblioteca estándar. En caso de error, el valor devuelto es `NULL`.

3. Cerrar archivos

Cerrar un archivo es más sencillo:

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

Devuelve `0` si tuvo exito, o `EOF` en caso de error.

4. Leer o escribir de un archivo

De la misma manera que `getchar` para leer un caracter de la entrada estándar, existe `fgetc` [2] para leer un único caracter de un archivo.

```
int fgetc(FILE *archivo);
```

De hecho, la siguiente función es prácticamente equivalente a la función `getchar()`.

```
int mi_getchar(void)
{
    return fgetc(stdin);
}
```

De la misma forma, existen `fputc`, para escribir un caracter a un archivo, `fscanf`, para leer con formato de un archivo, `fprintf` un archivo.[3], para escribir con formato a Sus prototipos son:

[1] Para más información: `man 3 fopen`.
[2] Para más información: `man 3 fgetc`.
[3] Para más información: `man 3 fputc`, `man 3 fscanf`, `man 3 fprintf`.

```
int fputc(int c, FILE *archivo);
int fscanf(FILE *archivo, const char *formato, ...);
int fprintf(FILE *archivo, const char *formato, ...);
```

Además de estas funciones existen:

```
char *fgets(char *buffer, int tamanio, FILE *archivo);
int fputs(const char *buffer, FILE *archivo);
```

La función `fgets` lee el archivo hasta encontrar un fin de línea, un fin de archivo o haber llegado a leer `tamanio` bytes. Cuando lee un fin de línea lo deja en el `buffer`. Devuelve la dirección del `buffer` o bien `EOF` si se trata de leer estando al final del archivo.

La función `fputs` escribe la cadena apuntada por `buffer` en `archivo`. Devuelve la cantidad de bytes escritos o bien `EOF` en caso de error.

Las funciones `fgets` y `fputs` constituyen la forma estándar de leer o escribir líneas en un archivo, si bien puede suceder que lo que se lea no sea una línea completa (cuando la línea ocupa más espacio que `tamanio`.

Si bien tienen un paralelo que trabaja sobre la entrada y salida estándar, esas funciones no se utilizan ya que pueden dar lugar a varios problemas de seguridad.

5. Otras funciones de archivos

Otras funciones que vale la pena mencionar son:

```
int fflush(FILE *archivo);
int feof(FILE *archivo);
```

La función `fflush` fuerza la escritura de los buffers que estén pendientes en el archivo. Devuelve 0 si se ejecutó correctamente, o `EOF` en caso de error. Puede utilizarse para evitar el comportamiento del buffer por líneas de las salida estándar.

La función `feof` devuelve algo distinto de cero si se encuentra al final del archivo o `0` en caso contrario.

6. Archivos binarios

Los archivos de texto son sencillos de procesar y faciles de leer aún fuera del programa que los usa, el éxito en los últimos años de los formatos XML, HTML, SVG, etc, demuestra su gran flexibilidad. Por otro lado, los archivos binarios permiten almacenar la información de forma que sea muy eficiente acceder a ella.

El formato a utilizar en una aplicación se debe decidir según el uso que se le vaya a dar a los archivos, si se quiere que sean legibles por seres humanos, si se quiere poder compartir la información entre aplicaciones, o si simplemente se quiere poder leer y guardar la información de la forma más eficiente.

Las funciones vistas hasta ahora son las más utilizadas al trabajar sobre archivos de texto, estas pueden servir para archivos binarios, pero además se necesitarán las siguientes:

```
size_t fread(void *buffer, size_t tamanio, size_t cantidad, FILE *archivo);
size_t fwrite(const void *buffer, size_t tamanio, size_t cantidad,
              FILE *archivo);
```

La función `fread` lee `cantidad` bloques de bytes de `tamanio` bytes cada uno, de un archivo, almacenandolos en `buffer`. Devuelve la cantidad de elementos leídos del archivo, en el caso de estar en el final del archivo devolverá 0.

De la misma forma `fwrite` escribe `cantidad` bloques de bytes de `tamanio` bytes cada uno en archivo y devuelve la cantidad de elementos escritos.

```
int fseek(FILE *archivo, long desplazamiento, int origen);
```

Se mueve dentro el archivo, `desplazamiento` es un valor relativo a `origen`, puede referirse al principio del archivo (`SEEK_SET`), a la posición actual (`SEEK_CUR`) o al final del archivo (`SEEK_END`). El valor devuelto será 0 en caso de exito o `-1` en caso de error.

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

Devuelve la posición actual del archivo, o `-1` en caso de error.

7. Ejemplo: Copiar un archivo

En este ejemplo vemos el uso de varias de las funciones mencionadas anteriormente. El código copia un archivo de un forma muy ineficiente, leyendolo de 1 caracter. Se muestra también el uso de `stderr`.

Podemos mejorarlo un poco leyendo por lineas en vez de caracter a caracter.

```
enum {MAXLINE = 1024};
...
    char buffer[MAXLINE], *aux;
    do {
        aux = fgets(buffer, MAXLINE, origen);
        if ( aux != NULL ) {
            fputs(buffer, destino);
        }
    } while (aux != NULL);
```

Se puede mejorar la eficiencia de este código utilizando `fread` y `fwrite`.

Código 1 `copiar.c`**:** Copia un archivo

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

int main(void)
{
    FILE *origen, *destino;
    int  valor;

    origen = fopen("copiar.c","r");
    if ( origen == NULL ) {
        fprintf(stderr, "Error al abrir el archivo origen");
        exit(1);
    }

    destino = fopen("copiar2.c","w");
    if ( destino == NULL ) {
        fprintf(stderr, "Error al abrir el archivo destino");
        exit(1);
    }

    do {
        valor = fgetc(origen);
        if ( valor != EOF ) {
            fputc(valor,destino);
        }
    } while (valor != EOF);
    fclose(origen);
    fclose(destino);

    return 0;
}
```