

Archivos en C

75.41/95.15 - Wachenchauzer

Sebastián Santisi
04/09/2015

Generalidades

- En C los archivos se acceden a través de un TDA: FILE *.
- El acceso a los archivos es secuencial, en forma de flujo (stream).
- Los archivos se discriminan entre archivos de entrada y de salida.
- La entrada y salida estándar está implementada sobre archivos.

Secuencia

- Un archivo debe abrirse (o crearse) a partir de un nombre de archivo y el modo de apertura.
- Luego puede operarse sobre funciones de entrada/salida.
- Al final, debe cerrarse, para liberar el recurso.

Tipos: Texto y binarios.

- No existe una distinción interna entre un archivo de texto y un archivo binario.
- Archivos de texto: Se interpreta que cada byte representa un carácter ASCII y que lo que se almacena es texto (por ejemplo, el número 25 puede almacenarse como '2' y '5').
- Archivos binarios: El archivo es un volcado de memoria. Los bytes tienen sentido en el contexto del tipo que representen (por ejemplo, el número 25 podría guardarse como los bytes 0x00000019).
- En algunas plataformas pueden hacerse diferencias en cómo se manejan caracteres especiales como '\n' en modo texto.

En general

- Tanto para archivos de texto como para archivos binarios necesito saber qué es lo que está escrito y en qué formato para poder recuperarlo.

fopen()

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

- Abre un archivo.
- Modos: “r”: read, “w”: write, “t”: text, “b”: binary. (ej: “rt”, “wb”, etc.)
- Si no puede abrir el archivo, devuelve NULL.
- Si quiero abrir un archivo para escritura, si no existe lo crea, si existe limpia su contenido.

fclose()

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

- Cierra un archivo.
- Devuelve EOF (es una macro) si falla el cierre. La falla del cierre puede indicar errores previos en la escritura del archivo.

Lectura

- Cuando se abre un archivo, el puntero de lectura se encuentra en el primer byte del mismo.
- Cada operación de lectura incrementa el puntero tantas veces como bytes leídos.
- El archivo puede leerse hasta que el puntero sobrepasa el final del archivo (EOF: end of file).
- Una vez que se sobrepasa el final, las funciones de lectura fallarán.

feof()

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

- Función booleana, devuelve true si el archivo YA se terminó.
- “YA se terminó”: Si acabo de leer el último byte todavía no se terminó, necesito sobrepasar esa marca.
- Generalmente, el chequeo puede hacerse sin necesidad de esta función porque todas las funciones de lectura avisan cuando el archivo se terminó.

Funciones texto/binario

- Como se dijo, la diferencia entre un archivo de texto y uno binario es más semántica que de contenido real del archivo.
- Hay funciones que tienen más sentido si son usadas para texto o para binario, pero no hay nada que las limite.

Funciones para binarios

```
int fwrite(const void *p, size_t s,  
           size_t n, FILE *f);
```

```
int fread(void *p, size_t s, size_t n,  
          FILE *f);
```

- Escriben o leen, respectivamente, n estructuras de tamaño s referenciadas por el puntero p en el archivo f.
- Devuelven la cantidad de estructuras que pudieron escribir/leer (si leyó menos que n => se terminó el archivo).

Lectura texto

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

- Devuelve o un carácter o EOF si se terminó el archivo (¡¡¡La devolución es entera!!!, no alcanza un char para guardar {char U EOF}).

```
char *fgets(char *s, size_t n, FILE *f);
```

- Lee una cadena hasta el '\n' inclusive o n - 1 caracteres. Devuelve NULL si se terminó el archivo.

```
int fscanf(FILE *f, const char *fmt, ...);
```

- Ídem scanf() sobre un archivo. Devuelve la cantidad de campos del formato interpretados.

Escritura texto

```
int fputc(int c, FILE *f);
```

- Escribe el carácter c en el archivo.

```
int fputs(const char *s, FILE *f);
```

- Escribe la cadena s en el archivo.

```
int fprintf(FILE *f, const char  
            *fmt, ...);
```

- Escribe con formato, como printf() en el archivo.

Flujos estándar

- Tres flujos por omisión
- Generalmente:
 - stdin: Asociado al teclado
 - stdout: Asociado al monitor
 - stderr: Asociado al monitor
- Este comportamiento puede modificarse



Piping

```
$ ./prog > archivo.txt
```

- Redirige el stdout de prog a archivo.txt.

```
$ ./prog1 | ./prog2
```

- Redirige el stdout de prog1 al stdin de prog2.

```
$ ./prog < archivo.txt
```

- Redirige el archivo al stdin de prog.
- Todos estos comportamientos se pueden combinar.

Ejemplos

01_argc.c

- Ejemplo de CLA.
- Suma los argumentos recibidos como si fueran números.
- Utiliza la función `atoi()` para realizar la conversión: Si falla, devuelve el número interpretado hasta lo primero no-número.

01_argc.c (salida)

```
[santisis@ss]7541$ ./01_argc
```

Uso:

```
./01_argc [numero]
```

```
[santisis@ss]7541$ ./01_argc 1 2 3 4
```

La suma es: 10

```
[santisis@ss]7541$ ./01_argc 1 2 3 4 hola
```

La suma es: 10

```
[santisis@ss]7541$ ./01_argc hola 1 2 3 4
```

La suma es: 10

```
[santisis@ss]7541$ ./01_argc 1hola 2hola 3hola 4hola
```

La suma es: 10

02_argc2.c

- Ídem 01_argc.c
- Realiza la conversión de cadena a entero con `strtol()`, con completo control sobre el estado de la conversión.

```
[santisis@ss]7541$ ./02_argc2
```

Uso:

```
./02_argc2 [numero]
```

```
[santisis@ss]7541$ ./02_argc2 1 2 3 4
```

La suma es: 10

```
[santisis@ss]7541$ ./02_argc2 1 2 hola
```

El argumento "hola" no es un numero!

```
[santisis@ss]7541$ ./02_argc2 1 2 1hola
```

El argumento "1hola" no es un numero!

03_generar_vector.c

- Genera un vector de una cantidad recibida por CLA de elementos enteros 0..99 pseudoaleatorios.
- Usa rand() para generar un aleatorio entre 0 y MAX_RANDOM; lo acota quedándose con el resto.
- Para obtener una nueva serie cada vez inicializa la serie con el valor de la fecha y hora actual.

03_generar_vector.c (salida)

```
[santisis@ss]7541$ ./03_generar_vector 3
```

```
3
```

```
56
```

```
62
```

```
39
```

```
[santisis@ss]7541$ ./03_generar_vector 3 > vector_3.txt
```

```
[santisis@ss]7541$ cat vector_3.txt
```

```
3
```

```
84
```

```
94
```

```
74
```

04_copiar.c

- Copia un archivo en un archivo nuevo.
- La iteración se efectúa de a un char por vez.

04_copiar.c (salida)

```
[santisis@ss]7541$ ./04_copiar
```

```
Uso: ./04_copiar <origen> <destino>
```

```
[santisis@ss]7541$ ./04_copiar archivoinexistente.txt  
xxx.txt
```

```
No pudo abrirse "archivoinexistente.txt"!
```

```
[santisis@ss]7541$ ./04_copiar vector_3.txt  
/acanotengopermisos.txt
```

```
No pudo abrirse "/acanotengopermisos.txt"!
```

```
[santisis@ss]7541$ ./04_copiar vector_3.txt  
vector_3_copia.txt
```

```
[santisis@ss]7541$ diff vector_3.txt vector_3_copia.txt
```

05_escribir_vector.c

- Genera un archivo binario con un vector de enteros.
- El formato del binario es:
 - 1)Entero que dice cuántos elementos hay
 - 2)Tantos enteros como elementos.
- El origen puede ser stdin o un archivo.
- En caso de que sea stdin no puedo diferenciar si hay un usuario o me hacen piping.

05_escribir_vector.c (salida)

```
[santisis@ss]7541$ ./05_escribir_vector
```

```
Uso: ./05_escribir_vector (<origen>) <destino>
```

```
[santisis@ss]7541$ ./05_escribir_vector vector.bin
```

```
Ingrese la cantidad de elementos: 3
```

```
Ingrese el valor numero 1: 0
```

```
Ingrese el valor numero 2: 1
```

```
Ingrese el valor numero 3: 2
```

```
[santisis@ss]7541$ hd vector.bin
```

```
00000000  03 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00  |.....|
```

```
[santisis@ss]7541$ ./05_escribir_vector vector_3.txt vector_3.bin
```

```
[santisis@ss]7541$ ./05_escribir_vector vector_3.bin < vector_3.txt
```

```
Ingrese la cantidad de elementos: Ingrese el valor numero 1: Ingrese el valor numero 2:
```

```
Ingrese el valor numero 3:
```

```
[santisis@ss]7541$ ./03_generar_vector 10 | ./05_escribir_vector vector_10.bin
```

```
Ingrese la cantidad de elementos: Ingrese el valor numero 1: Ingrese el valor numero 2:
```

```
Ingrese el valor numero 3: Ingrese el valor numero 4: Ingrese el valor numero 5: Ingrese el  
valor numero 6: Ingrese el valor numero 7: Ingrese el valor numero 8: Ingrese el valor  
numero 9: Ingrese el valor numero 10:
```

06_leer_vector.c

- Levanta en memoria un vector desde un archivo binario e imprime sus elementos.
- Usa memoria dinámica.
- Asume que el primer entero del archivo representa la cantidad de elementos.
- Para leer el paquete de enteros hace una única llamada a fread() con la cantidad del lote que espera recibir.

06_leer_vector.c (salida)

```
[santisis@ss]7541$ ./06_leer_vector vector_3.bin
```

```
3
```

```
84
```

```
94
```

```
74
```

```
[santisis@ss]7541$ ./06_leer_vector vector_3.bin >  
vector_3_salida.txt
```

```
[santisis@ss]7541$ diff vector_3.txt vector_3_salida.txt
```

```
[santisis@ss]7541$ ./06_leer_vector vector.bin
```

```
3
```

```
0
```

```
1
```

```
2
```

07_leer_vector2.c

- Ídem 06_leer_vector.c pero ahora no espera recibir el tamaño del vector. El archivo es un conjunto de enteros indeterminado.
- Usa memoria dinámica con una estrategia de crecimiento exponencial: 16, 32, 64, 128, etc.
- Luego de cada pedido de memoria intenta leer tantos elementos como espacio libre tenga en el vector en un sólo bloque:
 - Si lee lo pedido: Sigue leyendo.
 - Si lee menos: Entonces el archivo se terminó, y tiene todos los elementos.
- Notar que si bien el formato de archivo es diferente, es compatible con los archivos anteriores. Simplemente la “cantidad” pasa a ser un elemento más del vector.
- Notar que en los últimos dos ejemplos nunca leo los elementos de a uno por vez.