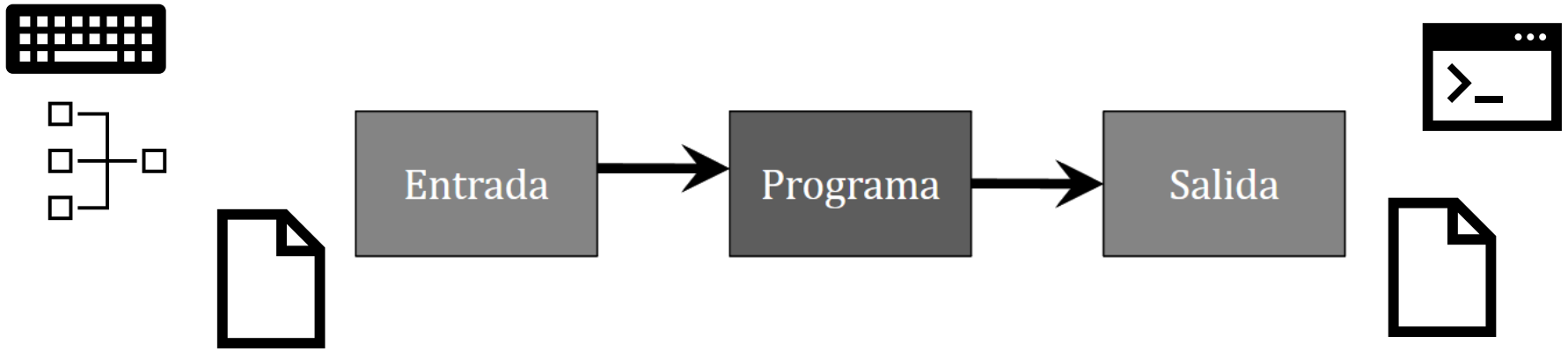
Manejo de Archivos (File Handling) en C

INFO II – Profesor: Ing. Federico Weber

Introducción al File Handling en C



¿Qué conocemos hasta ahora?

Hasta este momento, hemos trabajado principalmente con operaciones de entrada y salida a través de las funciones declaradas en el header <stdio.h>, como printf() y scanf().

Recientemente, aprendimos a interactuar con variables del Sistema Operativo como fuentes de entrada para nuestros programas.

¿Qué aprenderemos ahora?

El objetivo de esta sección del curso es expandir las capacidades de nuestros programas, permitiéndoles utilizar archivos de disco tanto para operaciones de entrada como de salida.

Esto nos permitirá crear aplicaciones que puedan almacenar información permanentemente y procesar datos provenientes de archivos externos.

Introducción al File Handling en C

¿Qué son los archivos en C?

- En lenguaje C, el manejo de archivos permite almacenar y recuperar información de forma persistente, más allá de la ejecución del programa.
- A diferencia de los datos en memoria (que se pierden al terminar el programa), un archivo es una secuencia de bytes en un dispositivo de almacenamiento (disco, memoria flash, etc.) que puede ser leído y escrito por el programa.

El concepto de flujo (stream)

- C abstrae los archivos mediante el concepto de flujo (stream) de datos: todas las operaciones de E/S se realizan sobre flujos de bytes secuenciales.

Streams que ya conocemos

- En realidad, ya hemos trabajado con streams previamente:
  - **stdin**: flujo de entrada estándar (utilizado por **scanf**())
  - **stdout**: flujo de salida estándar (utilizado por **printf**())
  - **stderr**: flujo de salida para errores
- Nosotros ya habíamos trabajado con streams, con **stdin** (con **scanf**) y **stdout** (con **printf**). Pero no pensábamos en estas cosas, ¿Por qué?
- Para tomar un dato del teclado, jamás pensamos en el teclado como hardware y esa es justamente la razón por la que el stream **stdin** es útil.
- Pero tampoco pensábamos en **stdin**. Eso es porque es el Sistema Operativo el que se encarga de abrir esos flujos, **stdin** y **stdout**, de manera automática para nosotros.

Introducción al File Handling en C

En C, las operaciones de archivo siguen generalmente estos pasos:

1. **Abrir** el archivo con la función **fopen**, obteniendo un puntero de tipo **FILE*** que representa el flujo hacia el archivo.
2. **Realizar operaciones** de lectura/escritura mediante funciones de la biblioteca estándar (por ejemplo **fprintf**, **fscanf**, **fread**, **fwrite**, etc.), utilizando el **FILE*** obtenido.
3. **Cerrar** el archivo con **fclose** cuando ya no se necesite, liberando el recurso.

A partir de ahora y en el transcurso de esta clase y otras, desarrollaremos cada uno de estos pasos y conceptos en detalle, cubriendo el acceso secuencial, el acceso random, los modos de apertura (lectura/escritura, **texto vs. binario**), las funciones estándar (**fopen**, **fseek**, **ftell**, **fflush**, **fprintf**, etc.), así como las **buenas prácticas** asociadas.

APERTURA DE ARCHIVOS

fopen y modos de acceso

- Para trabajar con un archivo en C primero se debe **abrir**.
- La apertura de un archivo se realiza con la función fopen, definida en <stdio.h>.
- Esta función recibe el nombre o ruta del archivo y un *modo de apertura* que indica la finalidad (lectura, escritura, etc.), y retorna un puntero de tipo FILE* que representa el archivo abierto.
- El tipo FILE es una estructura opaca (para nosotros) proporcionada por la biblioteca estándar que internamente guarda información sobre el estado del archivo (posición actual, buffer, etc.).
- **Sintaxis básica:** FILE *f = fopen("nombre_archivo.txt", "modo");
- Si la apertura falla (por ejemplo, si el archivo no existe en modo lectura, o falta permiso), fopen devuelve NULL.
- **Siempre se debe comprobar** que el puntero devuelto no sea NULL antes de usarlo, para evitar desreferenciar un puntero inválido (lo que causaría errores en tiempo de ejecución)

Nuestro primer programa, ejemplo01.C

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("datos.txt", "r");
    if (fp == NULL) {
        printf ("Error abriendo archivo");
        return 1;
    }

    // Resto del código...

    fclose(fp);
    return 0;
}
```

```
Error abriendo archivo
Press <RETURN> to close this window...
```

Abrimos el archivo llamado:
"**datos.txt**" en el modo "**r**".
Si bien no vimos aun los modos podemos sospechar que:
r: (read)lectura
w: (write)escritura

¿Dónde va a buscar nuestro programa a ese archivo?
Al no especificar ruta, deberá ir a buscarlo a algún lugar predefinido. Si estamos bajo un IDE la ubicación será típicamente la dirección del proyecto, si estamos ejecutando un programa ya compilado y ejecutado manualmente la carpeta será desde donde se ejecuta el programa.

Luego hacemos una verificación, si la apertura falla motramos un error y salimos del programa.

Siempre, Siempre asociado a un **fopen** exitoso, va a haber un **fclose**.

Mejoramos a ejemplo01.C

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

int main() {
    FILE *fp = fopen("datos.txt", "r");
    if (fp == NULL) {
        perror ("Error abriendo archivo");
        return 1;
    }

    // Resto del código...

    fclose(fp);
    return 0;
}
```

```
Error abriendo archivo: No such file or directory
Press <RETURN> to close this window...
```

Ventajas de perror vs printf:

Muestra el error específico del sistema:
perror automáticamente imprime el mensaje que proporcionas junto con la descripción del error del sistema operativo basada en la variable global **errno**.

Práctica estándar:
Es considerada una buena práctica en programación C para reportar errores de sistema.

printf	stdout
perror	stderr

Modos de apertura de archivos

El modo de apertura es un string de uno o dos caracteres que indica cómo se va a usar el archivo:

- **"r" (read)** – Abrir para **lectura**. El archivo debe existir; si no existe, **fopen** retornará **NULL**.
- **"w" (write)** – Abrir para **escritura**. Si el archivo existe, se **trunca** (borra su contenido previo); si no existe, se crea un archivo nuevo.
- **"a" (append)** – Abrir para **añadir** al final. Si no existe, se crea. Si existe, los datos que se escriban se agregarán después del contenido existente.
- **"r+" (read/update)** – Abrir para **lectura y escritura** (modo *lectura/escritura*). El archivo debe existir (no se crea nuevo) , **fopen** retornará **NULL**. . Permite tanto leer como escribir en él.
- **"w+" (write/update)** – Abrir un archivo para **lectura y escritura**, *truncando* el archivo si ya existe (empieza vacío). Si no existe, lo crea.
- **"a+" (append/update)** – Abrir para **lectura y escritura**, pero posicionándose al final. Si no existe, se crea. Permite leer el contenido ya existente y, al escribir, agregar datos al final.

Además cuando trabajamos en Windows, es NECESARIO indicar si el archivo va a ser de texto o binario.

Modos de apertura de archivos

La distinción entre archivos de texto y binarios es **conceptual** y está relacionada con el contenido, no con el modo de apertura en sí:

1. **Archivos de texto**:
    1. Contienen caracteres legibles por humanos (letras, números, símbolos)
    2. Cada byte o secuencia de bytes representa un carácter según alguna codificación (ASCII, UTF-8, etc.)
    3. Se pueden abrir y editar con editores de texto (como Notepad, VS Code, vi, etc.)
2. **Archivos binarios**:
    1. Contienen datos estructurados en formato nativo del C
    2. Pueden contener bytes que no representan caracteres imprimibles
    3. No son legibles directamente con editores de texto normales
    4. Ejemplos: bases de datos simples, registro de logs, snapshots de estructuras, etc.

Modo	Descripción	Texto/Binario	Notas
"r"	Abrir para lectura	Texto	El archivo debe existir
"rt"	Abrir para lectura	Texto	Igual que "r", t es redundante
"rb"	Abrir para lectura	Binario	El archivo debe existir
"w"	Abrir para escritura	Texto	Crea archivo o trunca si existe
"wt"	Abrir para escritura	Texto	Igual que "w", t es redundante
"wb"	Abrir para escritura	Binario	Crea archivo o trunca si existe
"a"	Abrir para añadir	Texto	Crea archivo si no existe
"at"	Abrir para añadir	Texto	Igual que "a", t es redundante
"ab"	Abrir para añadir	Binario	Crea archivo si no existe
"r+"	Abrir para lectura/escritura	Texto	El archivo debe existir
"r+t"	Abrir para lectura/escritura	Texto	Igual que "r+", t es redundante
"r+b", "rb+"	Abrir para lectura/escritura	Binario	El archivo debe existir, ambas formas válidas
"w+"	Crear para lectura/escritura	Texto	Crea archivo o trunca si existe
"w+t"	Crear para lectura/escritura	Texto	Igual que "w+", t es redundante
"w+b", "wb+"	Crear para lectura/escritura	Binario	Crea archivo o trunca si existe, ambas formas válidas
"a+"	Abrir/crear para lect./escr.	Texto	Escritura solo al final
"a+t"	Abrir/crear para lect./escr.	Texto	Igual que "a+", t es redundante
"a+b", "ab+"	Abrir/crear para lect./escr.	Binario	Escritura solo al final, ambas formas válidas

Modos de apertura de archivos

Saquemos los redundantes

Modo	Descripción	Tipo	Notas
"r"	Abrir para lectura	Texto	El archivo debe existir
"rb"	Abrir para lectura	Binario	El archivo debe existir
"w"	Abrir para escritura	Texto	Crea archivo o trunca si existe
"wb"	Abrir para escritura	Binario	Crea archivo o trunca si existe
"a"	Abrir para añadir	Texto	Crea archivo si no existe
"ab"	Abrir para añadir	Binario	Crea archivo si no existe
"r+"	Abrir para lectura/escritura	Texto	El archivo debe existir
"r+b", "rb+"	Abrir para lectura/escritura	Binario	El archivo debe existir, ambas formas válidas
"w+"	Crear para lectura/escritura	Texto	Crea archivo o trunca si existe
"w+b", "wb+"	Crear para lectura/escritura	Binario	Crea archivo o trunca si existe, ambas formas válidas
"a+"	Abrir/crear para lect./escr.	Texto	Escritura solo al final
"a+b", "ab+"	Abrir/crear para lect./escr.	Binario	Escritura solo al final, ambas formas válidas

Modos de apertura de archivos

Si bien en sistemas UNIX la “b” no es necesaria, es buena practica usarlo siempre. Hace nuestro código mas portable.
En Windows, sin el modificador "b", podrías tener transformaciones no deseadas que corromperían datos binarios.

Ejemplo02.c “apertura y cierre de un archivo de texto en modo escritura”

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

int main() {
    FILE *fp = fopen("datos.txt", "w");
    if (fp == NULL) {
        perror ("Error abriendo archivo");
        return 1;
    }

    // Resto del código...

    fclose(fp);
    return 0;
}
```

Vamos al IDE y buscamos donde creo el archivo.

Le cambiamos la ruta y vemos que pasa

Modificamos el archivo creado y volvemos a ejecutar el programa.

Importancia del **fclose**

- Siempre que terminemos de usar un archivo, debemos cerrarlo con **fclose**.
- Al cerrar el archivo, se liberan los recursos asociados y se vacían (flushed) los buffers pendientes al almacenamiento.
- Si olvidamos llamar a **fclose**, existe un límite en la cantidad de archivos que un programa puede tener abiertos simultáneamente, y además los datos podrían no guardarse correctamente hasta la terminación del programa.
- El sistema operativo usualmente cierra automáticamente todos los archivos abiertos al finalizar el programa, pero es una buena práctica cerrar los archivos.

Escritura secuencial en archivos de texto

Escritura secuencial en archivos de texto

Una vez abierto un archivo en un modo que permita escritura (por ejemplo "w", "a", "w+" o "r+"), podemos **escribir datos** en él. En C, las funciones más comunes para escritura de **archivos de texto** son:

- **int fprintf(FILE *f, const char *formato, ...)**
  Escribe texto formateado en el archivo (similar a printf en pantalla, pero indicando el archivo destino) . Devuelve:
  - El número de caracteres impresos
  - **EOF** (-1) si ocurre un error al escribir

- **int fputs(const char *s, FILE *f)**
  Escribe un string completo en el archivo (equivalente a imprimir el string). No añade automáticamente salto de línea. Devuelve:
  - Un valor no negativo (normalmente 0) si tuvo éxito
  - **EOF** si ocurre un error al escribir el string

- **int fputc(int c, FILE *f)**
  Escribe un único carácter (byte) al archivo. Devuelve:
  - El carácter escrito (como **unsigned char** promovido a **int**)
  - **EOF** si falla la escritura

Ejemplo03.c – Uso de fprintf

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

int main() {
    FILE *fp = fopen("G:\\Mi unidad\\datos.txt", "w");
    if (fp == NULL) {
        perror ("Error abriendo archivo");
        return 1;
    }

    char nombre[] = "Alice";
    int edad = 30;

    fprintf (fp, "Nombre: %s, Edad: %d", nombre, edad);

    fclose(fp);
    return 0;
}
```

La función fprintf se usa exactamente igual que printf, salvo que se especifica como primer argumento el FILE* del archivo donde escribir.

fprintf devuelve el número de caracteres escritos correctamente (sin contar el \0). En caso de error devuelve -1.

Agregar y quitar salto de línea al final

¿ Pero qué es exactamente acceso secuencial?

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

int main() {
    FILE *fp = fopen("D:\\Googledrive\\datos.txt", "w");
    if (fp == NULL) {
        perror ("Error abriendo archivo");
        return 1;
    }

    char nombre[] = "Alice";
    int edad = 30;

    for (int i=0; i<10; i++)
        fprintf (fp, "Nombre: %s, Edad: %d", nombre, edad);


    fclose(fp);
    return 0;
}
```

Mostrar que se imprime en pantalla

Se está escribiendo varias veces, pero siempre en la siguiente posición del archivo, avanzando automáticamente. Esto es escritura secuencial automática, y es justamente lo que caracteriza a este tipo de acceso: no tenés control sobre la posición, se escribe donde quedó el puntero de archivo.

Mas adelante veremos como realizar acceso aleatorio en el archivo (random acces)

Buffering - Almacenamiento temporal

- Las operaciones de E/S en C emplean **buffering** (almacenamiento temporal). Esto significa que al llamar a fprintf (u otras funciones de escritura), la información puede quedar en un **buffer en memoria** y no escribirse inmediatamente en el disco.
- La biblioteca estándar suele vaciar (flush) automáticamente estos buffers en ciertas condiciones:
  - El buffer se llena completamente
  - Se cierra el archivo (fclose())
  - El programa termina normalmente
- Si necesitamos forzar que los datos se escriban inmediatamente (por ejemplo, en un log crítico que debe volcarse en tiempo real), podemos usar fflush() para forzar la escritura pendiente de ese stream al archivo físico.

Ejemplo04.c – Modo append y fprintf

Notar uso de flose
fflush
ejecutar varias veces

```c
#include <stdio.h>

int main() {
    FILE *out = fopen("G:\\Mi Unidad\\temperaturas.txt", "a");
    if (out == NULL) {
        perror("No se pudo abrir el archivo de temperaturas");
        return 1;
    }

    float temp;
    printf("Ingrese temperatura medida: ");
    if (scanf("%f", &temp) != 1) {
        fprintf(stderr, "Entrada inválida.\n");
        fclose(out);
        return 1;
    }

    fprintf(out, "Temperatura: %.2f C\n", temp);
    fflush(out);  // opcional: forzar escritura inmediata
    printf("Dato guardado en el log.\n");

    fclose(out);
    return 0;
}
```

Ejemplo05.c – Uso de fputc

Ver el **scanf** y evitar desbordamiento

fputc devuelve:

- El carácter escrito (como un int) si todo salió bien
- **EOF** si hubo un error

```c
#include <stdio.h>

int main() {
    FILE *fp;
    int i;
    char str[10];

    fp = fopen("G:\\Mi Unidad\\temperaturas.txt", "w");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo de temperaturas");
        return 1;
    }

    printf("Ingrese su nombre [9 caracteres maximo]: ");
    scanf("%9s", str);

    for (i=0; str[i]!='\0'; i++)
    {
        fputc (str[i], fp);
    }
    fputc ('\n', fp);

    fclose(fp);
    return 0;
}
```

EOF – End Of File

EOF significa End Of File (fin de archivo), pero en C tiene un sentido más general:

- Es un valor especial (definido como **#define EOF (-1)** en **<stdio.h>**) que indica un error o el fin del flujo de entrada/salida.
- En funciones de escritura como **fputc** o **fprintf**, EOF actúa exclusivamente como indicador de error.
- En funciones de lectura como **fgetc**, **EOF** puede significar fin de archivo o error de lectura.
- Por eso, no debemos asumir directamente que **EOF** implica fin de archivo, sino que debemos consultar otras funciones que veremos mas adelante.
- En resumen: usamos **EOF** como valor de retorno especial para detectar condiciones fuera de lo normal, pero debemos interpretarlo cuidadosamente según la función y contexto.

Ejemplo06.c - uso fputs

No podemos dar formatos como con **fprintf.**

Ingresar fechas, ver saltos de linea

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char mensaje[] = "Este es un mensaje con fputs.\n";

    fp = fopen("G:\\Mi Unidad\\temperaturas.txt", "a");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    if (fputs(mensaje, fp) == EOF) {
        perror("Error al escribir con fputs");
        fclose(fp);
        return 1;
    }

    fclose(fp);
    printf("Mensaje guardado con fputs.\n");
    return 0;
}
```

Lectura secuencial en archivos de texto

Lectura secuencial en archivos de texto

Para **leer** desde un archivo de texto usamos funciones análogas a las de escritura, proporcionadas por **<stdio.h>**:

- **int fscanf(FILE *f, const char *formato, ...)**
  Lee datos formateados desde el archivo (similar a **scanf** desde teclado, pero indicando el archivo fuente). Devuelve:
  - El número de elementos correctamente leídos y asignados
  - **EOF** (-1) si ocurre un error o se alcanza el fin de archivo antes de leer algo

- **char* fgets(char *buffer, int tamaño, FILE *f)**
  Lee *una línea* completa (o hasta tamaño-1 caracteres) desde el archivo, copiándola en buffer (incluye el salto de línea si lo encuentra, y agrega '\\0' al final). Devuelve:
  - Un puntero al buffer (str) si se leyó correctamente
  - **NULL** si ocurre un error o se alcanza EOF antes de leer algún carácter

- **int fgetc(FILE *f)**
  Lee un **carácter** (byte) del archivo. Devuelve:
  - El carácter leído, como **unsigned char** promovido a **int**
  - **EOF** si se alcanza el fin de archivo o ocurre un error .

La elección de función depende del caso de uso.

Ejemplo07.c – uso de fscanf

.
Por ejemplo, si tenemos un archivo con *tres números enteros* en una línea separados por barras, podemos leerlos como en el programa de la derecha.

Es útil para traer este tipo de datos como por ejemplo:
hh:mm:ss
dd/mm/aaaa

No es recomendable para traer strings. Ya que corta con los espacios en blanco

```c
int main() {
    FILE *fp;
    int dia, mes, anio;
    fp = fopen("D:\\googledrive\\numeros.txt", "r");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo");
        return 1;   }
    if (fscanf (fp, "%d/%d/%d", &dia, &mes, &anio) == 3)
    {
        printf ("fecha: %d//%d//%d\n", dia, mes, anio);
        fclose(fp);
        return 0;
    }
    else
    {
        printf ("Formato inesperado o fin de archivo.\n");
        fclose(fp);
        return 1;
    }
}
```

Ejemplo07.c – uso de fscanf

.
¿Y si tengo mas de una línea?

17/06/1986
30/10/1986
4/4/2019
30/10/2024

fscanf con **%d**, **%f**, **%s** ignora automáticamente espacios, tabs y saltos de línea antes de los datos. Esto permite que funcione incluso si los datos están separados por saltos de línea o espacios.

```c
#include <stdio.h>

int main() {
    FILE *fp;
    int dia, mes, anio;
    fp = fopen("D:\\googledrive\\numeros.txt", "r");
    if (fp == NULL)
    {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    while (fscanf (fp, "%d/%d/%d", &dia, &mes, &anio) == 3)
        printf ("fecha: %d//%d//%d\n", dia, mes, anio);

    printf ("Formato inesperado o fin de archivo.\n");
    fclose(fp);
    return 1;

}
```

fscanf me permite traer datos de un archivo de texto y guardarlos directamente en variables del tipo correcto. No tengo que convertir strings a números manualmente: si el archivo dice 14/3/2024, fscanf puede meter esos tres valores directamente en mis variables int. Esa es su mayor ventaja, siempre y cuando el archivo esté bien formateado.

Ejemplo08.c – uso de fgets

- **fgets** me da una línea entera como texto. Si yo quiero los datos dentro de esa línea, tengo que parsearla.
- En cambio, fscanf ya me entrega los datos convertidos directamente a int, float, etc., si el formato es el correcto.
- Entonces, cuando sé cómo está armado el archivo, fscanf me simplifica mucho las cosas.
- Pero si el formato es muy variable o quiero más control, fgets me da flexibilidad.

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char str[50];

    fp = fopen("G:\\Mi Unidad\\numeros.txt", "r");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    if (fgets (str, sizeof(str), fp) == NULL)
    {
        perror ("Error al leer la entrada: ");
        fclose(fp);
        return 1;
    }

    printf ("%s", str);

    fclose(fp);

    return 0;
}
```

Ejemplo08.c – uso de fgets

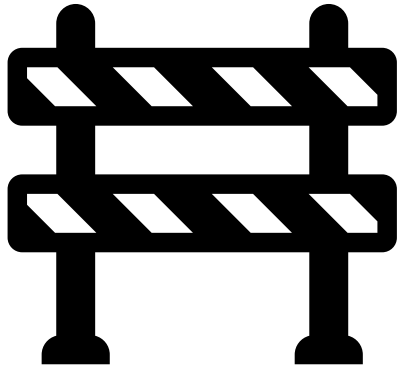
```c
#include <stdio.h>

int main() {
    FILE *fp;
    char str[50];

    fp = fopen("D:\\googledrive\\numeros.txt", "r");
    if (fp == NULL) {
        perror("No se pudo abrir el archivo");
        return 1;
    }

    // Leer línea por línea (cada fecha como un string)
    while (fgets(str, sizeof(str), fp) != NULL) {
        // Imprimir la línea (ya incluye \n si entra en el buffer)
        printf("Fecha (texto): %s", str);
    }

    fclose(fp);
    return 0;
}
```

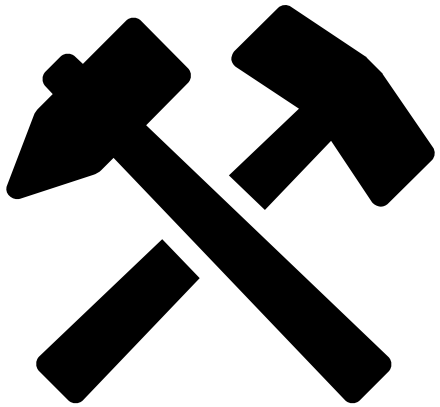
Feof() - **f**ile **e**nd **o**f **f**ile

- Ver ejemplo09.c
- Ejemplo10.c

Diapositiva en construcción.

- En el ejemplo09.c escribimos caracteres desde el teclado a un archivo.
- En el ejemplo10.c usando otro archivo, mucho mas grande, mostramos la lectura usando únicamente EOF como verificacion y forzamos un error.
- Observamos que usando solo EOF no sabemos si leímos todo el archivo o si se produjo un error.





feof() - **f**ile **e**nd **o**f **f**ile

- Así que sí, aunque suene redundante, literalmente significa "fin de archivo de archivo", porque sigue la convención de los nombres de funciones de la biblioteca estándar (**fgetc**, **fputc**, **fread**, etc.), que todas comienzan con f (de file).
- Y su prototipo es:
  - **int feof(FILE *stream);**
  - Devuelve un valor distinto de cero (true) si el indicador de fin de archivo está activado.
  - Devuelve 0 si todavía no se ha alcanzado el fin de archivo.

¿Qué hace feof() exactamente?

- feof(fp) devuelve true (no cero) si ya intentaste leer más allá del final del archivo
- Solo sirve después de un intento de lectura fallido
- En otras palabras:"feof() no predice el final. Solo confirma que ya te pasaste."

Ver ejemplo11.c

Problemas propuestos

1. Hacer un programa que cuente la cantidad de bytes que contiene un archivo de texto.
2. Hacer un programa que compare dos archivos de texto e indique si tienen o no igual contenido.
3. Concatenar dos archivos de texto (poner uno a continuación del otro en un mismo archivo).
4. Realizar un programa que busque una palabra en un archivo de texto y si está varias veces indicar cuántas. La palabra se debe ingresar como argumento del main.
5. Realizar un programa que muestre el contenido de un archivo de texto, cuyo nombre es ingresado por argumentos del main.
6. Realizar un programa que copie un archivo a otro cuyos nombres estaran indicados como argumentos del main. Asi: copy archivo_original.txt archivo_copia.txt.
7. Escriba un programa que lea el contenido de la variable de entorno PATH y lo almacene en un archivo de texto llamado path.txt.
    1. Como extra, se pide que el contenido se procese de forma tal que cada ruta contenida en PATH (separadas por ; en Windows o : en Linux) quede en una línea distinta del archivo.

FIN DE LA CLASE