LENGUAJE C: MANEJO DE ARCHIVOS (FILE I/O FUNCTIONS)

- Los programas almacenan e intercambian información leyendo y escribiendo en **ficheros** (**archivos**, **files**). Un fichero se considera como un flujo de datos (***stream***) de **E/S**, cuyo destino puede ser una impresora, un monitor o un disco.
- La gestión del control de almacenamiento externo no forma parte original de C, por lo que se recurre a funciones desarrolladas en la Librería Estándar **stdio.h**, aunque las mismas no cubren todas las necesidades (por ejemplo manejo de directorios).
- Algunos ambientes de desarrollo proporcionan funciones adicionales (de bajo nivel) que permiten suplir esa deficiencia (**io.h**), aunque con el grave inconveniente de la falta de portabilidad y de ser soluciones limitadas a plataformas específicas.
- En todos los casos, previo a cualquier operación se requiere la apertura del fichero. En ese momento se le asocia un **handle** que puede ser una estructura **FILE** (estándar) o un **int** (funciones de bajo nivel). Al finalizar **DEBE** ser cerrado.

LENGUAJE C: TIPO DE ARCHIVOS

- De acuerdo a la forma en que se almacena la información, los ficheros se pueden clasificar en dos grupos: de **texto** (con formato) y **binarios** (sin formato).
  - Archivos de texto: están compuestos por una serie de caracteres organizados en líneas terminadas por caracteres de ***newline*** ('\n') y ***carriage return*** (‘\r’).
  - Archivos binarios: formados por una secuencia de bytes. Cualquier fichero que no sea de texto, será binario. En general se usan con datos estructurados.

  Los archivos de texto ocupan mas lugar en el disco. Para almacenar 3.1415927 se necesitan 9 bytes en el modo texto, y sólo 4 bytes en el modo binario.

- De acuerdo a las operaciones utilizadas para acceder a los datos el acceso es:
  - Secuencial: para acceder a una determinada información se deben recorrer todas las datos desde el principio hasta llegar al deseado.
  - Directo: se accede a la posición requerida en forma directa, sin tener que pasar por las posiciones anteriores. Son mas versátiles, se puede acceder a cualquier parte del fichero en cualquier momento como si fueran arrays en memoria.

LIBRERÍA ESTÁNDAR – stdio.h

- Para acceder a la información las funciones utilizan un puntero a una estructura **FILE**, cuya declaración puede variar de acuerdo al compilador utilizado:

Turbo C

```
typedef struct
{
   short level;
   unsigned flags;
   char  fd;
   unsigned char hold;
   short bsize;
   unsigned char *buffer, *curp;
   unsigned istemp;
   short token;
} FILE;
```

CodeBlocks (*)

```
typedef struct _iobuf
{
   char *_ptr;
   int   _cnt;
   char *_base;
   int   _flag;
   int   _file;
   int   _charbuf;
   int   _bufsiz;
   char *_tmpfname;
} FILE;
```

(*) en la declaración figura: **Some believe that nobody in their right mind should make use of the internals of this structure.**

- El programador sólo **necesita saber cómo utilizar las funciones** provistas por la librería que proporcionan el soporte para comunicarse con el sistema operativo.

FUNCIONES DE LA LIBRERÍA ESTÁNDAR

- **stdio.h** considera los dispositivos de E/S estándar también como archivos, declarando tres identificadores especiales (no requieren apertura para su uso):
  - **stdin**: dispositivo de entrada estándar = teclado.
  - **stdout**: dispositivo de salida estándar = monitor.
  - **stderr**: dispositivo de salida de errores estándar = monitor.
- Funciones:

Control:	**fopen**	**fclose**	**fseek** **rewind**	**rename** **remove**
Escribir/leer un carácter:	**fputc**	**fgetc**	**putc** **putchar**	**getc** **getchar**
Escribir/leer un string:	**fputs**	**fgets**	**puts**	**gets**
Escribir/leer con formato:	**fprintf**	**fscanf**	**printf** **sprintf**	**scanf** **sscanf**
Escribir/leer a un ***buffer***:	**fwrite**	**fread**		

ESQUEMA DE ACCESO A LOS DATOS

- Acceso secuencial:

  1: Abrir el archivo ← **fopen**

  2: Mientras haya datos para procesar,

  2.1: Realizar las operaciones necesarias

  2.2: Guardar/Recuperar los datos ← **fprintf, fscanf**

  3: Cerrar el archivo ← **fclose**

- Acceso directo:

  1: Abrir el archivo ← **fopen**

  2: Mientras haya datos para procesar,

  2.1: Realizar las operaciones necesarias

  2.2: Posicionar el puntero del archvio ← **fseek, ftell, rewind**

  2.3: Guardar/Recuperar los datos ← **fwrite, fread**

  3: Cerrar el archivo ← **fclose**

APERTURA Y CIERRE DE ARCHIVOS

- **FILE *fopen(char *_nombre_, char *_modo_)**: abre y/o crea el fichero ***nombre*** (se puede especificar el path completo) y lo asocia a un flujo de datos (***stream***). Retorna un puntero a un objeto de tipo **FILE** que representa al archivo, o **NULL** si no es accesible. Las demás funciones utilizan ese puntero para acceder al archivo. ***modo*** es de la forma: "r/w/a[t/b][+]":

  r = abre el fichero para lectura. Si el fichero no existe devuelve error.

  w = abre el fichero para escritura. Si no existe lo crea, **si existe lo sobrescribe**.

  a = abre el fichero para añadir datos al final del mismo. Si no existe, se crea.

  t = el fichero es de tipo texto (default).

  b = el fichero es de tipo binario.

  + = se abre el fichero para lectura y escritura.

- **int fclose(FILE *_fp_)**: cierra el fichero apuntado por ***fp***. Retorna 0x0 si se pudo cerrar ó **EOF** (-1) en caso de error. Si no se cierra el archivo en forma adecuada se pueden perder todos los datos.

CONTROL DE ARCHIVOS

- No existe una función estándar para verificar la existencia de un fichero. Esta comprobación **DEBERÍA** realizarse antes de abrir un archivo en el modo escritura ("**w**") para evitar perder información. Se puede realizar en forma indirecta:

```
int exists(const char *fname)
{
    FILE *file = fopen(fname, "r");
    if (file != NULL)
    {
        fclose(file);  return 1;
    }
    return 0;
}
```

- **int rename(const char *old_name, const char *new_name)**: cambia el nombre del fichero ***old_name*** a ***new_name***. Retorna 0x0 si se pudo realizar ó un número distinto de 0 en caso de error.

- **int remove(const char *name)**: borra (delete) el fichero ***name***. Retorna 0x0 si se pudo realizar ó un número distinto de 0 en caso de error.

Ej. 1: open

ESCRITURA / LECTURA DE CARACTERES

- **int fputc(int *car*, FILE **fp*)**: escribe el carácter ***car*** en el stream de salida ***fp*** y avanza la posición del índice (puntero de lectura/escritura). Retorna un **int** con el carácter escrito o **EOF** si se produce un error.

- **int putc(int *car*, FILE **fp*)**: es una macro que tiene el mismo efecto que **fputc()**.

  int putchar(int *car*): es una macro que se define como **putc(*car*, stdout)**.

- A una macro es más difícil pasarle parámetros variables (por ejemplo si ***car*** es el resultado de una evaluación). No se puede definir su dirección, por lo que no se puede usar como argumento de una función. Las macros tienen un código optimizado y en general son más rápidas y su invocación es más rápida.

- **int fgetc(FILE **fp*)**: retorna el carácter apuntado por el indicador de posición (que se incrementa en uno). En caso de error ó llegar al fin del archivo devuelve **EOF**.

- **int getc(FILE **fp*)**: es una macro que tiene el mismo efecto que **fgetc()**.

  int getchar(): es una macro que se define como **getc(stdin)**.

ESCRITURA / LECTURA DE STRINGS

- **int fputs(const char *_str_, FILE *_fp_)**: escribe el string apuntado por ***str*** al ***stream*** de salida ***fp***. No se escribe el 0x0 final ni un '\n'. En caso de encontrar un '\n' en ***str*** se sustituye por el par "**\n\r**" (0xA + 0xD = *nl* + *cr*). Retorna **EOF** si hay un error.

- **int puts(char *_str_)**: escribe el string apuntado por ***str*** en **stdout**. Se agrega al final un carácter de nueva línea '\n'. Retorna un int > 0 o **EOF** si se produce un error.

- **char *fgets(char *_str_, int _cnt_, FILE *_fp_)**: lee caracteres desde el ***stream*** apuntado por ***fp*** y los almacena en la memoria apuntada por ***str***. La lectura continúa hasta que ingresan ***cnt***-1 caracteres, se lea un '\n' o se alcance el fin de archivo. El carácter de nueva línea también es leído ('\r' es descartado). Se agrega 0x0 como fin del string. En caso de error se retorna **NULL**. ***cnt*** evita el desborde del string.

- **char *gets(char *_str_)**: lee caracteres desde **stdin** y los almacena en la memoria apuntada por ***str***. La lectura continúa hasta que se alcance '\n'. El carácter de nueva línea se descarta y se agrega 0x0. En caso de error retorna **NULL**.

Ej. 2: put_get

ESCRITURA / LECTURA CON FORMATO (1/2)

- **int fprintf(FILE *_fp_, const char *_formato_, arg_1, arg_2, … arg_n)**: el funcionamiento es igual a **printf** pero la salida se envía al ***stream*** de salida ***fp*****,** en lugar de la pantalla. Retorna el número de caracteres escritos o **EOF** en caso de error.

  formato : incluye caracteres de control: **% [flags] [width] [.prec] [{h|l}] type**

  width: cantidad **mínima** de caracteres a imprimir (no se trunca).

  prec: para enteros = cantidad **mínima** de dígitos. Para flotantes = **máximo** número de dígitos después del punto decimal. Para strings = **máximo** de caracteres

flags	Especificación
None	alinea a derecha (completa con ‘ ‘ hasta [**width**])
0	alinea a derecha (completa con ‘0‘ hasta [**width**])
-	alinea a izquierda (agrega ‘ ‘ hasta [**width**])
+	siempre imprime el signo (+/-)
blank	imprime ‘-’ para números negativos, sino un ‘ ‘
#	type = **o** → octal (0ddd) type = **x**/**X** → hexa (0xddd)

Ej. 3: print

ESCRITURA / LECTURA CON FORMATO (2/2)

- **int sprintf(char *_buffer,_ const char *_formato_, _arg$_1$, arg$_2$, … arg$_n$_)**: se utiliza igual que **fprintf**, la salida se envía a ***buffer***. Retorna el número de caracteres escritos.

- **int fscanf(FILE *_fp,_ const char *_formato_, _&arg$_1$, … arg$_n$_)**: el funcionamiento es igual a **scanf** pero la entrada se obtiene desde el ***stream*** de entrada ***fp***, en lugar del teclado. Retorna el número de campos procesados correctamente.

  formato : incluye caracteres de control: **% [flags] [width] [{h|l}] type**

  %[ ]: La conversión finaliza cuando se encuentra un carácter que no corresponde al conjunto especificado: %[0-9]; %[AD-G34] (A, D~G, 3 ó 4); %[^A-C] (No ABC).

  %*_type_: lee una entrada de tipo *type* pero no la almacena en ninguna variable.

  ✓ La entrada finaliza cuando se encuenta ‘ ‘ o se alcanza el valor **width**.

  ✓ Al efectuar la lectura **convierte texto a entero y/o flotante directamente**.

✓ **int sscanf(char *_buffer,_ const char *_formato_, _&arg$_1$, … &arg$_n$_)**: se utiliza igual que **fscanf**, leyendo desde ***buffer***. Retorna el número de caracteres entrados.

Ej. 4: scanf

POSICIONAMIENTO EN EL *STREAM* (1/2)

- **void rewind(FILE *_stream_)**: mueve el puntero (marca una posición) de lectura /escritura al inicio del ***stream*** y resetea los flags de **EOF** y de error (si se produjo).

- **int fseek(FILE *_stream_, long _offset_, int _origen_)**: establece la posición del puntero para efectuar una operación de lectura o escritura en el ***stream***. ***offset***: cantidad de bytes que se desplaza el puntero hacia el final (>0) ó hacia el inicio (<0) del archivo desde la posición fijada por ***origen***: 0 (SEEK_SET) indica el comienzo del archivo; 1 (SEEK_CUR) es la posición actual y 2 (SEEK_END) indica el fin del archivo. En caso de error retorna un valor distinto de 0x0.

- **long ftell(FILE *_stream_)**: retorna la posición del puntero o **EOF** en caso de error.

- **long feof(FILE *_stream_)**: retorna ≠ 0 si se alcanzó la posición final del archivo. **NO usar** en un bucle para controlar la lectura: **while**(!**feof**(***fp***)) { … fgetc/fgets …}. La función comprueba el indicador **EOF**. El seteo del mismo lo realiza la función de lectura al ingresar -1, por lo que el bucle anterior realizaría un lectura de más.

POSICIONAMIENTO EN EL *STREAM* (2/2)

- **int fsetpos(FILE *_stream_, const fpos_t *_position_)**: mueve la posición del puntero en el ***stream*** a ***position***, que es de tipo **fpos_t** (definido en stdio.h). Retorna 0x0 si no hay error. Es análoga a **fseek()** pero menos flexible, ya que ***position*** **sólo** puede obtenerse mediante **fgetpos()**. Se pierde la posibilidad de hacer referencia al principio ó al fin del archivo.

- **int fgetpos(FILE *_stream_, fpos_t *_position_)**: almacena la posición del puntero en el ***stream*** en ***position***. Retorna 0x0 si no hay error. Es análoga a **ftell().**

- Tamaño del archivo: en algunos casos se necesita conocer cuántos bytes ocupa el archivo (reservar memoria para su almacenamiento o para copiar todos los bytes). El C estándar no cuenta con una función que retorne la longitud del fichero.

```
long longitud;
fseek(fp, 0L, SEEK_END);        // posiciona el puntero al final
longitud = ftell(fp);           // retorna un long
```

El máximo valor que se puede obtener es ≈ 2 GBytes (sizeof(long) = 4 bytes).

CONTROL DE ERRORES

- **int fflush(FILE *_stream_)**: fuerza la salida de los datos acumulados en el *buffer* de salida/entrada del ***stream*** (válido también para **stdin**/**stdout**). Si el puntero pasado como parámetro es NULL se hará el vaciado de todos los ficheros abiertos.

- **int ferror(FILE *_stream_)**: retorna 0x0 si la última operación realizada sobre el ***stream*** no produjo error. Todas las operaciones afectan la condición de error (sólo se tiene el último valor). Con la función **perror()** se puede determinar el mismo.

- **int perror(char *_str_)**: transforma el número de error (**errno**) en un mensaje de error y lo imprime en el **stderr** (salida estándar de errores) con el formato:

  str + ':' + ' ' + mensaje de error (sistema) + '\n'.

- **char *strerror(int _num_)** (declarada en **string.h**): traduce el error identificado por ***num*** en un mensaje y retorna un puntero al string. Los mensajes correspondientes a cada valor de **errno** se encuentran en el archivo de cabecera **errno.h**.

- **void clearerr(FILE *_stream_)**: resetea los *flags* de error y el indicador **EOF.**

Ej. 5: control

ESCRITURA / LECTURA SIN FORMATO

- **int fwrite(const void *_buffer_, size_t _size_, size_t _count_, FILE *_stream_)**: escribe al ***stream*** ***count*** objetos, de tamaño ***size*** cada uno, desde el ***buffer***. Retorna la cantidad de bytes escritos.

- **int fread(void *_buffer_, size_t _size_, size_t _num_, FILE *_stream_)**: lee ***num*** objetos, de tamaño ***size*** cada uno, desde el ***stream***, y los almacena en ***buffer***. Retorna la cantidad de objetos leídos.

```
struct stRegistro {char Nombre[34];  int dato;  int matriz[23];  float valor};
FILE *fp;
struct stRegistro reg[10];                    // vector de 10 estructuras

// Para hacer una lectura del registro número 6 se saltan los 5 primeros:
fseek(fp, 5*sizeof(stRegistro), SEEK_SET);
fread(&reg[5], sizeof(stRegistro), 1, fp);
```

Ej. 6: write_read

MANTENIMIENTO – GESTION DE ARCHIVOS

- Existencia: abrir el archivo en modo lectura, si no existe, crear en modo escritura.
- Altas: abrir el archivo en modo añadir (append).
- Bajas: utilizar un archivo auxiliar. Abrir el original para lectura y el auxiliar para escritura. Transferir al auxiliar todos los registros excepto el que se debe eliminar. A continuación borrar el original, y renombrar al auxiliar como original.
- Modificaciones: utilizar también un archivo auxiliar. Una vez realizadas las modificaciones, transferir todos los registros al auxiliar. Borrar y renombrar.
- Consultas: abrir el archivo en modo lectura.
- Listado por pantalla: abrir en modo lectura y leer mientras no se llegue al final.
- Listado por impresora: utilizar dos *streams* de texto: el original (modo lectura) y el de impresora (modo escritura):

```
FILE *fimp = fopen ("prn", "w");          // prn representa la impresora (lpt, lpt1)
```