

# 11 Funciones de Biblioteca

---

## 11.1 Introducción

Uno de los principales objetivos de la programación modular es la reutilización de código: alguien escribe alguna función, de forma que cualquiera pueda utilizarla en su código.

En el estándar C se establecen una serie de funciones, denominadas de biblioteca, que cualquier compilador debe incorporar, de forma que cualquier programa que utilice dichas funciones pueda ser compilado sin errores en cualquier máquina.

En este tema se estudiarán algunas de estas funciones de biblioteca.

Para obtener una mayor información sobre cualquier función de biblioteca del estándar C utilice el comando **man**. En dicha ayuda encontrará el fichero de tipo **include** en el que se encuentra la declaración de la función de biblioteca en cuestión, y que deberá incluir en todos los archivos que vayan a utilizar dicha función.

## 11.2 Reserva dinámica de memoria

Hemos visto que para poder utilizar una variable es necesario reservarle cierta cantidad de memoria para su uso exclusivo; eso se consigue habitualmente mediante la declaración de la variable.

Pero hay situaciones en las que no es posible declarar todas las variables que vamos a necesitar; por ejemplo, porque se trate de una tabla cuyo tamaño no conoceremos hasta que el programa no se ejecute, o por necesitar estructuras de datos dinámicas cuyo tamaño va cambiando durante la ejecución del programa.

En esos casos se hace imprescindible la utilización de las funciones de reserva dinámica de memoria que incorpora el estándar C.

### 11.2.1 Función de reserva **malloc**

```
void * malloc(size_t1 tamanio);
```

Esta función reserva la cantidad de memoria indicada por **tamanio**, y devuelve el puntero al comienzo de la zona de memoria asignada, o **NULL** si no se ha podido hacer la reserva.

Hay tres aspectos importantes a reseñar en esta función.

El primero es cómo calcular la memoria que necesitamos reservar. Eso dependerá de para qué queremos reservar la memoria, pero en cualquier caso siempre será de utilidad el operador **sizeof**. Así, por ejemplo, si queremos reservar memoria para una estructura de tipo **struct generic**, la cantidad de memoria a reservar será **sizeof(struct generic)**, mientras que si queremos reservar memoria para una tabla de enteros de **tam** elementos, la memoria a reservar será de **tam \* sizeof(int)**.

El segundo aspecto a considerar es que devuelve un puntero de tipo genérico. La función **malloc** se utiliza para reservar memoria para cualquier tipo de datos, por lo que a priori no puede saber de qué tipo va a ser la zona de memoria devuelta. Por eso es imprescindible que el valor devuelto por **malloc** lo asociemos al tipo adecuado, mediante el operador de conversión forzada. Siguiendo con los dos ejemplos anteriores, tendríamos:

---

<sup>1</sup> **size\_t** es un tipo definido mediante **typedef** y que normalmente equivale a un **unsigned int**.

- en el primer caso el puntero devuelto lo sería a una estructura de tipo `struct generic`, por lo que la llamada correcta a la función `malloc` sería:

```
(struct generic *) malloc(sizeof(struct generic))
```

- en el segundo ejemplo devolvería memoria reservada para una tabla de enteros, por lo que la llamada correcta sería:

```
(int *) malloc(tam * sizeof(int))
```

El tercer aspecto a destacar es que la función devuelve `NULL` cuando no ha podido reservar la memoria. Puesto que para poder utilizar una variable necesitamos espacio reservado para la misma, *SIEMPRE* hay que comprobar que se ha podido realizar la reserva comprobando el valor devuelto por la función `malloc`.

Una llamada típica a la función `malloc` sería por tanto de la siguiente forma:

```
struct generic * paux = NULL;  
  
paux = (struct generic *) malloc(sizeof(struct generic));  
if (NULL == paux)  
    /* Código que gestiona el error de falta de memoria */  
else  
    /* Código normal de la función */
```

### 11.2.2 Función de reserva `calloc`

```
void * calloc(size_t numero, size_t tamanio);
```

La función `calloc` es análoga a `malloc`, con dos diferencias:

- La primera, visible en la propia declaración de la función, es que, mientras en `malloc` sólo teníamos un parámetro, y cuando queríamos reservar memoria para una tabla necesitamos realizar la multiplicación de forma explícita, en `calloc` esta multiplicación se hace implícitamente a partir de los dos parámetros con los que se llama.
- La segunda, no detectable en la declaración, es que la función `calloc` inicializa a cero toda la zona de memoria reservada.

Al igual que sucede con `malloc`, `calloc` devuelve `NULL` cuando no ha podido reservar la memoria, por lo que *SIEMPRE* hay que comprobar que se ha podido realizar la reserva comprobando el valor devuelto por la función `calloc`.

Una llamada típica a la función `calloc` sería por tanto de la siguiente forma (suponga que `tam` es un parámetro de la función en la que se incluye este fragmento de código):

```
int * paux = NULL;  
  
paux = (int *) calloc(tam, sizeof(int));  
if (NULL == paux)  
    /* Código que gestiona el error de falta de memoria */  
else  
    /* Código normal de la función */
```

Comparando ambas funciones, es evidente que `malloc` es adecuada cuando queremos reservar memoria para un único elemento (generalmente una estructura), mientras que `calloc` lo es cuando queremos reservar memoria para un conjunto de elementos (habitualmente una tabla).

### 11.2.3 Función de liberación `free`

Con las funciones `malloc` y `calloc` reservamos memoria para uso exclusivo por el programa que realiza la llamada. Lógicamente, esa zona de memoria debemos reservarla única y exclusivamente durante el tiempo que sea necesario, de forma que cuando ya no la necesitemos pueda ser utilizada por otro programa.

Por lo tanto, igual que existen funciones para reservar memoria, existe una función para liberarla: la función `free`.

```
void free(void * puntero);
```

La función `free` libera la zona de memoria apuntada por `puntero`, que previamente ha debido ser reservada con `malloc` o `calloc`. Una vez realizada la llamada a `free`, la zona de memoria apuntada

por **puntero** *NO PUEDE* volver a utilizarse. Si **puntero** vale NULL, la llamada a la función no tiene ningún efecto.

Aunque es evidente, no está de más señalar que el valor de **puntero** no se ve modificado por la llamada a **free**, como cualquier parámetro de cualquier función.

Como norma general, y para no mantener sin liberar ninguna zona de memoria, en los programas debe existir una llamada a **free** por cada llamada a **malloc** o **calloc** existente.

Si llamamos a **free** con un valor que no haya sido asignado por una llamada a **malloc** o **calloc**, o con un valor que haya sido liberado con anterioridad, el sistema de gestión de memoria dinámica del sistema operativo puede corromperse; este problema es tanto más grave cuanto que sus efectos no se hacen visibles inmediatamente, sino transcurrido un tiempo variable, lo que hace muy difícil su localización. Por ello es conveniente que asignemos el valor NULL a cualquier puntero ya liberado, de forma que una posterior llamada a **free** con dicho puntero no tenga efectos nocivos. Y así protegemos además el programa de la posible utilización de un puntero ya liberado, puesto que estaríamos utilizando un puntero igualado a NULL y la ejecución del programa se abortaría en ese punto exacto, pudiendo corregirlo.

Una llamada típica a la función **free** sería por tanto de la siguiente forma (suponga que **paux** almacena un valor devuelto por **malloc** o **calloc**):

```
free(paux);
paux = NULL;
```

#### 11.2.4 La reserva dinámica y las tablas multidimensionales

Supongamos que en un determinado programa necesitamos reservar memoria para una tabla de **int** de dos dimensiones, con **nfilas** filas y **ncolumnas** columnas. Por lo visto en el tema de tablas, una tabla de esas características equivale a una sucesión de **nfilas \* ncolumnas** **ints**, por lo que haríamos la reserva de la siguiente forma:

```
int * pTabla = NULL;

pTabla = (int *) calloc(nfilas * ncolumnas, sizeof(int));
if (NULL == pTabla)
    /* Código que gestiona el error de falta de memoria */
else
    /* Código normal de la función */
```

El acceso a los elementos de la tabla (que como puede observarse es una tabla de **int**) se hará calculando previamente la posición del elemento dentro de esa tabla; así, para acceder al elemento de la fila **fila** y columna **columna**, escribirímos:

```
pTabla[fila * ncolumnas + columna]
```

El acceso a esta tabla no podría hacerse mediante la utilización de doble índice puesto que no se conocía el número de columnas en tiempo de compilación.

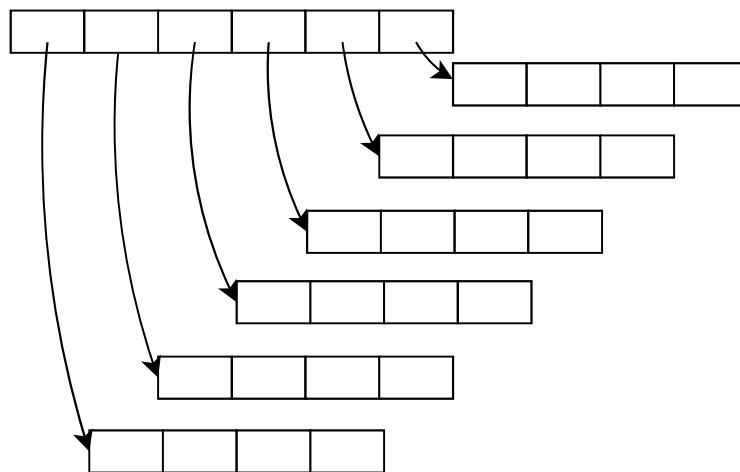
Existe un segundo procedimiento para la reserva dinámica de tablas multidimensionales; este segundo método complica la reserva de memoria, pero simplifica el acceso a los elementos. Consiste en considerar la tabla multidimensional no como una tabla de tablas, sino como una tabla de punteros a tablas, según se recoge en la figura 11.1.

El código necesario para generar dicha estructura sería:

```
int ** reservaTabla(int nfilas, int ncolumnas)
{
    int ** pTabla = NULL;
    int i = 0;
    int j = 0;

    /* Reservamos memoria para la tabla de punteros a tabla */
    pTabla = (int **) calloc(nfilas, sizeof(int));

    for (i = 0; (NULL != pTabla) && (i < nfilas); i++)          (1)
    {
        pTabla[i] = (int *) calloc(ncolumnas, sizeof(int));
        if (NULL == pTabla[i])
        {
            /* Si tenemos problemas en la reserva de un elemento,
               devolvemos la memoria previamente reservada ... */
            for (j = i - 1; j >= 0; j--)                                (2)
                free(pTabla[j]);
            /* ... incluyendo la tabla de punteros original */
        }
    }
}
```



**Figura 11.1** Tabla multidimensional como tabla de punteros..

```

        free(pTabla);
        pTabla = NULL;
    }

    return pTabla;
}

```

(4)

- (1) El **if** comprobando que se ha podido reservar memoria está implícito en la condición del **for**: si ha habido un error **pTabla** valdrá **NULL** y no se entrará en el bucle.
- (2) La última reserva que hicimos correctamente fue la correspondiente al elemento anterior al actual; desde ese elemento (hacia atrás) tendremos que realizar la liberación de memoria.
- (3) En este caso no es necesario igualar **pTabla[j]** a **NULL** puesto que la variable va a ser destruida.
- (4) Al igualar **pTabla** a **NULL**, además de cumplir con lo enunciado en un apartado anterior para protegernos contra errores en ejecución, evitamos que vuelva a entrar en el bucle **for** y asignamos el valor que tiene que ser devuelto por la función en caso de error.

Cuando reservamos memoria para una tabla de esta forma, obtenemos un puntero a puntero a **int**, que como sabemos podemos tratar como el nombre de una tabla de punteros a **int**; si lo indexamos (**pTabla[i]**, por ejemplo), lo que obtenemos es un puntero a **int**, que podemos tratar como el nombre de una tabla de **int**; si lo volvemos a indexar (**pTabla[i][j]**, por ejemplo) obtendremos un **int**. Es decir, con este método de reserva de memoria dinámica para tablas multidimensionales podemos utilizar la indexación múltiple sin necesidad de conocer el valor máximo de ninguna de las componentes en tiempo de compilación.

### 11.3 Entrada y Salida en C.

Muchas aplicaciones requieren escribir o leer información de un dispositivo de almacenamiento masivo. Tal información se almacena en el dispositivo de almacenamiento en forma de un archivo de datos. Los archivos de datos permiten almacenar información de modo permanente, pudiendo acceder y alterar la misma cuando sea necesario.

En C existe un conjunto extenso de funciones de biblioteca para crear y procesar archivos de datos. En C podrá trabajar con ficheros de dos tipos:

- texto: se trabaja con secuencias de caracteres divididos en líneas por el carácter nueva línea (' \n ').
- binarios: se trabaja con una secuencia de octetos que representan zonas de memoria.

Cuando se trabaja con archivos el primer paso es establecer un área de búffer, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria del ordenador y el archivo de datos. Este área de búffer (también llamada *flujo* o *stream*) la gestiona la propia biblioteca estándar, y permite leer y escribir información del archivo más rápidamente de lo que sería posible de otra manera. Para poder intercambiar información entre el área de buffer y la aplicación que utiliza las funciones de biblioteca, se utiliza un puntero a dicha zona de buffer. En la siguiente declaración:

```
FILE *ptvar;
```

**FILE** (se requieren letras mayúsculas) es un tipo especial de estructura que establece el área de búffer, y **ptvar** es la variable puntero que indica el principio de este área. El tipo de estructura **FILE**, que está definido en el archivo de cabecera del sistema **stdio.h**, también se conoce como **descriptor de fichero**.

Las operaciones de entrada y salida en ficheros se realiza a través de funciones de la biblioteca estándar cuyos prototipos están en **stdio.h**, no existiendo palabras reservadas del lenguaje que realicen estas operaciones.

### 11.3.1 Funciones de apertura y cierre

Un archivo debe ser “abierto” para poder realizar operaciones sobre él. La apertura de un archivo asocia el archivo indicado a un descriptor de fichero, y devuelve su puntero. Este puntero es el que se deberá utilizar en lo sucesivo para cualquier operación sobre el fichero. Cuando se ha finalizado de operar sobre él, debe ser cerrado para que el sistema operativo libere los recursos que le tenía asignados.

#### Función **fopen**

```
FILE *fopen(const2 char *nombre, const char *modo);
```

Esta función abre el archivo **nombre** y devuelve como resultado de la función el descriptor de fichero, o **NULL** si falla en el intento.

La cadena de caracteres **modo** determina cómo se abre el fichero; los valores más utilizados para dicho parámetro son los siguientes:

Modo	Operación	Si existe ...	Si no existe ...
<b>r</b>	lectura	lo abre y se posiciona al principio	error
<b>w</b>	escritura	borra el antiguo y crea uno nuevo	lo crea
<b>a</b>	escritura	lo abre y se posiciona al final	lo crea

#### Función **fclose**

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

Esta función cierra el archivo referenciado por **fp**. Esto implica descartar cualquier buffer de entrada no leído, liberar cualquier buffer asignado automáticamente y cerrar el flujo. Devuelve **EOF** si ocurre cualquier error y **0** en caso contrario.

Antes de terminar cualquier programa, debe asegurarse de que ha cerrado todos los archivos que haya abierto previamente.

### 11.3.2 Funciones de entrada y salida

#### Función **fgetc**

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

Esta función devuelve el siguiente carácter al último leído de **fp** como un **int**, o **EOF** si se encontró el fin de archivo o un error.

#### Función **fgets**

```
char * fgets(char *s, int n, FILE *fp);
```

Esta función lee hasta **n-1** caracteres (a partir del último carácter leído de **fp**) en la cadena **s**, deteniéndose antes si encuentra nueva línea o fin de fichero. Si se lee el carácter de nueva línea se incluye en la cadena **s**. La cadena **s** se termina con '**\0**'. La función devuelve **s**, o **NULL** si se encuentra fin de fichero o se produce un error.

<sup>2</sup> El calificador de tipo **const** indica que el valor del parámetro no se modifica dentro de la función llamada.

### Función `fputc`

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

Esta función escribe el carácter **c** (convertido a **unsigned char**) a continuación del último carácter escrito en **fp**. Devuelve el carácter escrito, o **EOF** en caso de error.

### Función `fputs`

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

Esta función escribe la cadena **s** (no necesita que contenga '\n', pero sí que termine en '\0') en **fp**. Devuelve un número no negativo si no ha habido problemas, o **EOF** en caso de error.

### Copia de un fichero de texto

El siguiente ejemplo realiza la copia de un fichero de texto en otro utilizando las funciones de lectura y escritura de caracteres vistas hasta ahora. Los nombres de los ficheros origen y destino se proporcionan por la línea de comandos:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE * fent = NULL;
    FILE * fsal = NULL;
    int     caracter;

    if (argc != 3)
        puts("Se necesitan dos parametros: origen y destino\n");
    else
    {
        if ((fent = fopen(argv[1], "r")) == NULL)
            puts("No se puede abrir el fichero origen\n");
        else
        {
            if ((fsal = fopen(argv[2], "w")) == NULL)
                puts("No se puede abrir el fichero destino\n");
            else
            {
                // Leemos cada caracter del fichero de entrada ...
                while ((caracter = fgetc(fent)) != EOF)
                    // ... y lo escribimos en el fichero de salida.
                    fputc(caracter, fsal);
                fclose(fsal);
            }
            fclose(fent);
        }
    }
    return 0;
}
```

#### 11.3.3 Salida con formato: función `fprintf`

Con las funciones vistas hasta ahora, sólo es posible imprimir secuencias de caracteres. Sin embargo, en muchas ocasiones es necesario imprimir datos numéricos, y con un cierto formato. Para lograr esto se utiliza la función:

```
int fprintf(FILE *stream, const char *format, ...);
```

Esta función escribe la cadena de formato **format** en **stream**. Devuelve el número de caracteres escritos, o un valor negativo si se produce algún error. La cadena de formato contiene dos tipos de objetos: caracteres ordinarios (que se copian en el flujo de salida), y especificaciones de conversión (que provocan la conversión e impresión de los argumentos). Cada especificación de conversión comienza con el carácter % y termina con un carácter de conversión.

Los caracteres de conversión más utilizados son los siguientes:

**c** para representar caracteres; el argumento se convierte previamente a **unsigned char**.

**u, o, X, x** para representar enteros sin signo en notación decimal, octal (sin ceros al principio), hexadecimal (sin el prefijo **0x** y con los dígitos **A** a **F** en mayúsculas), y hexadecimal (sin el prefijo **0x** y con los dígitos **a** a **f** en minúsculas), respectivamente.

**d** para representar enteros con signo en notación decimal.

**s** para representar cadenas de caracteres; los caracteres de la cadena son impresos hasta que se alcanza un '\0' o hasta que ha sido impreso el número de caracteres indicados por la precisión.

**f, E, e** para representar números de tipo **double**, en notación decimal, en notación científica con el exponente **E** en mayúscula, o en notación científica con el exponente **e** en minúscula, respectivamente. Por defecto se escriben seis dígitos significativos.

**G, g** para representar números de tipo **double**; es equivalente a la especificación **E, e** si el exponente del número expresado en notación científica es inferior a -4 o mayor o igual que la precisión (que por defecto es seis), y equivalente a **f** en caso contrario.

**p** se utiliza para imprimir el valor de un puntero (representación dependiente de la implementación).

Entre el % y el carácter de conversión pueden existir, en el siguiente orden:

a) **banderas**: modifican la especificación y pueden aparecer en cualquier orden

- el dato se ajusta a la izquierda dentro del campo (si se requieren espacios en blanco para conseguir la longitud del campo mínima, se añaden *después* del dato en lugar de *antes*).
- + Cada dato numérico es precedido por un signo (+ o -). Sin este indicador, sólo los datos negativos son precedidos por el signo - .

**espacio** si el primer carácter no es un signo, se prefijará un espacio

**0** Hace que se presenten ceros en lugar de espacios en blanco en el relleno de un campo. Se aplica sólo a datos que estén ajustados a la derecha dentro de campos de longitud mayor que la del dato

**#** Con las conversiones tipo **o** y tipo **x**, hace que los datos octales y hexadecimales sean precedidos por **0** y **0x**, respectivamente. Con las conversiones tipo **e**, tipo **f** y tipo **g**, hace que se presenten todos los números en coma flotante con un punto, aunque tengan un valor entero. Impide el truncamiento de los ceros de la derecha realizada por la conversión tipo **g**.

**número** Un número que estipula un ancho mínimo de campo. El argumento convertido será impreso en un campo de por lo menos esta amplitud, y en una mayor si es necesario. Si el argumento convertido tiene menos caracteres que el ancho de campo, será llenado a la izquierda (o derecha, si se ha requerido justificación a la izquierda) para completar el ancho de campo. El carácter de relleno normalmente es espacio, pero es 0 si está presente la bandera de relleno con ceros.

. Un punto, que separa el ancho de campo (número previo) de la precisión (número posterior).

**número** Un número, la precisión, que estipula el número máximo de caracteres de una cadena que serán impresos, o el número de dígitos que serán impresos después del punto decimal para conversiones **e**, **E** o **f**, o el número de dígitos significativos para conversiones **g** o **G**, o el número mínimo de dígitos que serán impresos para un entero (serán agregados ceros al principio para completar el ancho de campo necesario).

b) Un modificador de tipo:

**hh** indica que el argumento correspondiente va a ser impreso como **signed char** o **unsigned char**.

**h** indica que el argumento correspondiente va a ser impreso como **short** o **unsigned short**.

**l** indica que el argumento es **long** o **unsigned long**.

**ll** indica que el argumento es **long long** o **unsigned long long**.

**L** indica que el argumento es **long double**.

#### 11.3.4 Entrada con formato: función **fscanf**

```
int fscanf(FILE *fp, const char *format, ...);
```

Esta función lee a través del descriptor de fichero **fp** bajo el control de **format**, y asigna los valores convertidos a través de los argumentos, cada uno de los cuales debe ser un puntero. La función devuelve **EOF** si se llega al final del archivo o se produce un error antes de la conversión; en cualquier otro caso devuelve el número de valores de entrada convertidos y asignados. La cadena de formato contiene especificaciones de conversión, que se utilizan para interpretar la entrada. La cadena de formato puede contener:

- espacios o tabuladores, que se ignoran.
- caracteres ordinarios (distintos de %), que se espera que coincidan con los siguientes caracteres que no son espacio en blanco del flujo de entrada.
- especificaciones de conversión, consistentes en %, un carácter optativo de supresión de asignación \*, un número optativo que especifica una amplitud máxima de campo, una h, l o L optativa que indica la amplitud del objetivo, y un carácter de conversión.

Una especificación de conversión determina la conversión del siguiente campo de entrada. Normalmente el resultado se sitúa en la variable apuntada por el argumento correspondiente, excepto en el caso que se indique supresión de asignación con \* como en %\*s. El campo de entrada simplemente se salta; no se hace asignación. Un campo de entrada está definido por una cadena de caracteres diferentes del espacio en blanco; se extiende hasta el siguiente carácter de espacio en blanco o hasta que el ancho de campo, si está especificado, se haya agotado.

Esto implica que **fscanf** leerá más allá de los límites de la línea para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. Los caracteres de espacio en blanco son el blanco, tabulador, nueva línea, retorno de carro, tabulador vertical y avance de línea.

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un puntero. Los caracteres de conversión pueden ser los siguientes:

- d** int \*; entero decimal.
- i** int \*; entero. El entero puede estar en octal (iniciado con 0) o hexadecimal (iniciado con 0x o 0X).
- o** int \*; entero octal (con o sin cero al principio).
- u** unsigned int \*; entero decimal sin signo.
- x** int \*; entero hexadecimal (con o sin 0x o 0X).
- c** char \*; caracteres. Los siguientes caracteres de entrada se pondrán en la cadena indicada, hasta el número especificado por el ancho del campo; el valor por omisión es 1. No se agrega '\0'. En este caso se suprime el salto normal sobre los caracteres de espacio en blanco; para leer el siguiente carácter que no sea blanco, hay que usar %1s (lee un carácter y lo guarda como una cadena).
- s** char \*; cadena de caracteres que no es espacio en blanco (no entrecomillados). Apunta a una cadena de caracteres suficientemente grande para contener la cadena y un '\0' terminal que se le agregará.
- e, f, g** float \*; número de punto flotante. El formato de entrada para los float es un signo optativo, una cadena de números posiblemente con un punto decimal, y un campo optativo de exponente con una E o e seguida posiblemente de un entero con signo.
- n** int \*; escribe en el argumento el número de caracteres consumidos o leídos hasta el momento por esta llamada. No se lee entrada alguna. La cuenta de elementos convertidos no se incrementa.

### 11.3.5 Funciones de lectura/escritura binaria

Hasta ahora se ha estudiado la lectura/escritura en ficheros que son tratados como una secuencia de caracteres. Ahora se describen las funciones que se utilizan para tratar los ficheros como una secuencia de octetos, leyendo o escribiendo trozos de memoria.

#### Lectura binaria: función **fread**

```
unsigned fread(void *ptr, size_t size, size_t nobj, FILE *fp);
```

Esta función lee de **fp** en la cadena **ptr** hasta **nobj** objetos de tamaño **size**. La función devuelve el número de objetos leídos (este valor puede ser menor que el número solicitado).

#### Escritura binaria: función **fwrite**

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

Esta función escribe de **ptr** **nobj** objetos de tamaño **size** en **fp**. Devuelve el número de objetos escritos, que es menor que **nobj** en caso de error.

### 11.3.6 Funciones especiales

#### Detección de fin de fichero: función `feof`

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

Esta función devuelve un valor diferente de cero si se ha llegado al final del archivo. No se detecta el final del fichero hasta que no se intente leer después de haber finalizado la lectura del fichero. Si se lee el último carácter NO se ha llegado al final del archivo. Para que esta función devuelva un valor diferente de cero hay que intentar leer después del último carácter.

#### Detección de error: función `ferror`

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

Esta función devuelve un valor diferente de cero si se ha producido algún error.

### 11.3.7 Flujos estándar: `stdin`, `stdout` y `stderr`

Al hablar de ficheros se ha introducido el concepto de búfferes y descriptores o punteros a ficheros. Existen unos flujos de texto (*streams*) especiales que se abren automáticamente cuando un programa empieza a ejecutarse, y sus descriptores son:

**stdin** entrada estándar.

**stdout** salida estándar.

**stderr** salida de errores.

A partir de ese momento se puede utilizar estos identificadores (del tipo `FILE *`) con cualquiera de las funciones vistas previamente. Así, por ejemplo:

```
printf("hola mundo");
scanf("%d", &edad);
```

son equivalentes a

```
fprintf(stdout, "hola mundo");
fscanf(stdin, "%d", &edad);
```

La salida estándar y la salida de errores están por defecto dirigidas a la pantalla del ordenador.

### 11.3.8 Errores frecuentes en el uso de las funciones de E/S

El error más simple, y no por ello infrecuente, es olvidar que `scanf` recibe como parámetros las direcciones donde almacenar los valores que recibe de la entrada estándar.

```
float precio;
scanf("%f", &precio);
```

en vez de

```
float precio;
scanf("%f", &precio);
```

Recuerde que cuando se trata de leer cadenas de caracteres se le pasa el nombre de la tabla.

```
char nombre[25];
scanf("%s", nombre);
```

Otro error relacionado con el uso de punteros es que el puntero utilizado no apunte a una zona de memoria reservada.

En definitiva, es necesario comprobar antes de utilizar una variable de tipo puntero si apunta a una zona de memoria válida, y con el tamaño suficiente.

#### `scanf` no es perfecto

Otro posible error en el uso de `scanf` es suponer que siempre funcione como nosotros esperamos (cosa que en general no se debe suponer de ninguna función, ya sea de biblioteca o codificada por nosotros, y especialmente si estamos depurando el programa). Por ejemplo, edite y compile el siguiente código:

```
#include <stdio.h>

int main()
{
    float saldo;
    char accion;

    printf(stdout, "Introduzca accion (D) y saldo\n");
    scanf("%c %f", &accion, &saldo);
    if ('D' == accion)
        saldo = saldo * 2;
    else
        saldo = saldo / 2.0;
    printf(stdout, "Accion : %c\n", accion);
    printf(stdout, "Saldo : %f\n", saldo);

    return(0);
}
```

Este programa funcionará correctamente si introduce una cadena de caracteres y un número:

```
salas@318CDCr12:~$ a.out
Introduzca accion (D) y saldo
D 1000
Accion : D
Saldo : 2000.000000
salas@318CDCr12:~$
```

Pero si ejecuta el programa y al introducir el número se equivoca:

```
salas@318CDCr12:~$ a.out
Introduzca accion (D) y saldo
D q123
Accion : D
Saldo : -316884466973887584331540463616.000000
salas@318CDCr12:~$
```

El programa dejará valores inesperados en la variable **saldo** si el segundo valor a leer no es del tipo numérico. Este valor podrá variar en distintas ejecuciones.

Por lo tanto, habría que comprobar que **scanf** (o **fscanf**) devuelve tantos valores como se espera. Si no, existe la posibilidad de que las variables utilizadas en **scanf** conserven los valores previos, o incluso cualquier valor no esperado si no estaban iniciadas.

Observe también que **scanf** se detiene en la lectura del número en el momento que encuentra un espacio en blanco genérico o un carácter no numérico. En el siguiente ejemplo intentamos introducir la cantidad **1000** y tecleamos en vez de un cero final el carácter **o**:

```
salas@318CDCr12:~$ a.out
Introduzca accion (D) y saldo
D 100o
Accion : D
Saldo : 200.000000
salas@318CDCr12:~$
```

**scanf** lee el número 100, y continua su ejecución normalmente, ignorando el carácter '**o**'.

### **scanf y los espacios en blanco**

Es peligroso abandonar espacios en blanco finales en la cadena de formato de **scanf**. Compruébelo con el siguiente programa, que deja un inocente espacio en blanco antes de terminar la cadena de formato:

```
#include <stdio.h>

int main()
{
    int precio;

    printf("Precio del libro?: ");
    scanf("%d ", &precio);
    printf("\n %d \n ", precio);

    return 0;
}
```

**scanf** lee el precio, y a continuación se queda descartando los espacios en blanco (o tabuladores, caracteres de nueva línea, etc.). Parece que el programa esté bloqueado. Para terminar la lectura del precio es necesario que se teclee algún carácter distinto de espacio. Para corregir este programa basta eliminar ese espacio en blanco final.

Esta propiedad de **scanf** se puede utilizar para leer datos con un formato conocido previamente. Por ejemplo, una fecha de la forma **dd/mm/yyyy**: los tres números están separados por barras. La lectura se puede hacer con el siguiente programa:

```
#include <stdio.h>

int main()
{
    int dia;
    int mes;
    int year;

    printf("Fecha de nacimiento ? (dd/mm/yyyy): ");
    scanf("%d/%d/%d", &dia, &mes, &year);
    printf("Fecha de nacimiento es %d-%d-%d \n",
           dia, mes, year);

    return 0;
}
```

### scanf y fgets

Otro error en el uso de **scanf** es leer un número con **scanf("%d", ...)** y a continuación intentar leer una cadena de caracteres con la función **fgets**: da la impresión de que el código se salta la llamada a la función **fgets**. Observe el siguiente ejemplo:

```
#include <stdio.h>
#define MAX 25

int main()
{
    int precio;
    char nombre[MAX];

    printf("Precio del libro?: ");
    scanf("%d", &precio);
    printf("Nombre del libro?: ");
    fgets(nombre, MAX-1, stdin);
    printf("\nNombre: %s Precio: %d euros\n", nombre, precio);

    return 0;
}
```

Si ejecutamos el programa generado, tendremos:

```
salas@318CDCr12:~$ a.out
Precio del libro?: 12
Nombre del libro?:
Nombre: Precio: 12 euros
salas@318CDCr12:~$ a.out
Precio del libro?: 26 El perfume
Nombre del libro?:
Nombre: El perfume Precio: 26 euros
salas@318CDCr12:~$ a.out
```

El comportamiento observado se debe a lo siguiente: en la primera ejecución, la sentencia **scanf** lee el precio del libro, y se detiene en el carácter de nueva línea, sin leerlo. La función **fgets** reanuda la lectura donde **scanf** la dejó, lee el carácter nueva línea, y como su condición de finalización es leer un carácter nueva línea detiene su ejecución, ignorando el resto de la entrada.

En la segunda ejecución, **scanf** lee el precio y se detiene al detectar un separador (en este caso un espacio en blanco, no nueva línea). **fgets** continúa leyendo donde **scanf** se quedó, y no se detiene hasta llegar al carácter de nueva línea.

Para no depender de cómo el usuario introduzca los datos, una opción es leer todos los caracteres de nueva línea pendientes antes de **fgets**, mediante la siguiente sentencia intercalada entre la lectura del precio y la lectura del título:

```
while (fgetc(stdin) != '\n');
```

### **scanf y los formatos numéricos**

Existen unas pequeñas diferencias entre las especificaciones de formato de tipo numérico en **printf** y **scanf**. Se presentan en la siguiente tabla:

	<b>printf</b>	<b>scanf</b>
<b>float</b>	<b>%f</b>	<b>%f</b>
<b>double</b>	<b>%f</b>	<b>%lf</b>
<b>long double</b>	<b>%Lf</b>	<b>%Lf</b>

### **Lecturas de cadenas de caracteres**

Para leer cadenas de caracteres con espacios en blanco no es posible usar **scanf**, puesto que considera los espacios en blanco separadores de campo. Suponga que desea leer el título de un libro con **scanf**: a priori no conoce cuantas palabras va a contener el título del libro, ni el autor, etc.

Para solucionar este 'problema' se utiliza la función **fgets**, que lee hasta encontrar un carácter de nueva línea (y no un espacio genérico como hace **scanf**).

Otro inconveniente de **scanf** para este tipo de situaciones es que no comprueba si se rebasan los límites de la cadena donde se almacena la entrada estándar. Ya se sabe que rebasar los límites del espacio de memoria reservado conduce a errores de memoria, muchos de ellos de difícil localización. Por eso es también preferible usar **fgets**, al que se le indica el tamaño máximo de almacenamiento.

Pero no hay que olvidar que **fgets** no sustituye el carácter de nueva línea '\n' por el terminador '\0'.

Otra solución a la lectura de datos genérica es leer líneas de texto completas (de fichero o de entrada estándar) con **fgets**. Una vez que tengamos una cadena de caracteres, podemos procesarla para determinar si es correcta o no, qué formato tiene, etc. Incluso se pueden procesar con la función **sscanf**, o con funciones creadas por el programador. **sscanf** funciona exactamente igual que **scanf** o **fscanf**, con la única diferencia que lee los datos de una cadena que se le pasa como primer parámetro:

```
int sscanf(char *cad, const char *format, ...)
```

## **11.4 Funciones de Cadenas de Caracteres**

El uso de cadenas de caracteres es tan frecuente que la biblioteca estándar de C incorpora algunas funciones de manejo de cadenas. Sus declaraciones o prototipos se encuentran en **string.h**, y enunciaremos algunas de ellas.

### **11.4.1 Funciones de copia**

```
char * strcpy(char *dest, const char *orig);
```

Copia la cadena **orig** en la cadena **dest**, incluyendo '\0'. Devuelve **dest**.

```
char * strncpy(char *dest, const char *orig, int n);
```

Copia hasta **n** caracteres de la cadena **orig** en la cadena **dest**. Devuelve **dest**. Rellena con '\0' si **orig** tiene menos de **n** caracteres.

### **11.4.2 Funciones de encadenamiento**

```
char * strcat(char *inicio, const char *final);
```

Concatena la cadena **final** al final de la cadena **inicio**. Devuelve **inicio**.

```
char * strncat(char *inicio, const char *final, int n);
```

Concatena hasta **n** caracteres de la cadena **final** al final de la cadena **inicio**, terminando con '\0'. Devuelve **inicio**.

### **11.4.3 Funciones de comparación**

Recuerde que no es posible en C comparar dos cadenas con el operador ==. En su lugar se utilizan las funciones siguientes:

```
int strcmp(const char *s1, const char *s2);
```

Compara la cadena **s1** con la cadena **s2**. Devuelve un número negativo si **s1** es anterior a **s2**, un número positivo si **s1** es anterior a **s2**, o cero si las dos cadenas son iguales.

```
int strncmp(const char *s1, const char *s2, int n);
```

Compara hasta **n** caracteres de la cadena **s1** con la cadena **s2**. Devuelve un número negativo si **s1** es anterior a **s2**, un número positivo si **s1** es anterioresposterior a **s2**, o cero si las dos cadenas son iguales.

#### 11.4.4 Funciones de búsqueda

```
char * strchr(char *s, char c);
```

Devuelve un puntero a la primera ocurrencia de **c** en **s**, o NULL si no está presente.

```
char * strrchr(char *s, char c);
```

Devuelve un puntero a la última ocurrencia de **c** en **s**, o NULL si no está presente.

```
int strlen(char *s);
```

Devuelve la longitud de **s**.

La función **strlen** devuelve la longitud efectiva de la cadena, sin contar el terminador. Luego para copiar una cadena en otra hay que reservar un carácter más que la longitud que devuelva **strlen**.

Por otra parte, las funciones de copia y concatenación de caracteres NO reservan espacio en memoria para las cadenas destino, y además suponen que el espacio reservado es suficiente.

#### 11.4.5 Otras funciones

Otras funciones que pueden ser de interés, y que se encuentran declaradas en **stdlib.h** son:

```
int atoi (const char *s);
```

Devuelve el valor de **s** convertido a **int**.

```
double atof (const char *s);
```

Devuelve el valor de **s** convertido a **double**.

