Curso de C, por Antonio Lebrón Bocanegra

Este manual está extraido del paquete de software “Tutor C/C++ 1.0”, desarrollado por Antonio Lebrón Bocanegra como proyecto fin de carrera en la Facultad de Informática de Sevilla, y tutelado por Manuel Mejías Risoto. El paquete original era un programa para MsDos, que actuaba como lector paginado del texto del curso. Dicho paquete original no sólo incluía este texto sobre C, sino otro similar sobre C++, así como ejercicios de C y ejercicios de C++.

Tanto esta versión convertida a PDF como el curso original están disponibles en

www.nachocabanes.com/c/

LECCIÓN 1

INTRODUCCION AL CURSO DE C

El objetivo de este curso es enseñar el lenguaje C, o dicho de otro modo, enseñar a programar en lenguaje C.

INDICE DE ESTA LECCION

En esta lección se va a estudiar los siguientes puntos:

ORIGENES: Breve historia del lenguaje C.

CARACTERISTICAS: Algunas características importantes del lenguaje.

USO: Pasos para realizar un programa en este lenguaje.

EJEMPLOS: Tres programas para empezar a programar en C cuanto antes.

ORIGENES DEL C

El lenguaje C fue inventado por **Dennis Ritchie** en 1972 cuando trabajaba, junto con Ken Thompson, en el diseño del sistema operativo UNIX.

El lenguaje C deriva del lenguaje B de Thompson, el cual, a su vez, deriva del lenguaje BCPL desarrollado por Martin Richards. Durante muchos años el estándar de C fue la versión proporcionada con el sistema operativo UNIX versión 5. Pero pronto empezaron a surgir muchas implementaciones del C a raíz de la popularidad creciente de los microordenadores. Por este motivo, se hizo necesario definir un C estándar que está representado hoy por el **ANSI C**.

En este tutor se va a estudiar el **C estándar**. No obstante, si la opción turbo está activada, también se incluirá en la explicación la versión Turbo C de Borland

International, que es uno de los mejores compiladores de C que existen en el mercado.

Cuando nos referimos a Turbo C, estamos hablando indistintamente de las distintas versiones que existen sobre los paquetes **Turbo C**, **Turbo C++** y **Borland C++**, puesto que en todos ellos se puede programar en C.

El lenguaje C suministrado por Turbo C es, simplemente, una ampliación del ANSI C, sobre todo en el número de funciones de librería suministradas.

CARACTERISTICAS DEL LENGUAJE C

Algunas características del lenguaje C son las siguientes:

- Es un **lenguaje de propósito general**. Este lenguaje se ha utilizado para el desarrollo de aplicaciones tan dispares como: hojas de cálculos, gestores de bases de datos, compiladores, sistemas operativos, ...

- Es un **lenguaje de medio nivel**. Este lenguaje permite programar a alto nivel (pensando a nivel lógico y no en la máquina física) y a bajo nivel (con lo que se puede obtener la máxima eficiencia y un control absoluto de cuanto sucede en el interior del ordenador).

- Es un **lenguaje portátil**. Los programas escritos en C son fácilmente transportables a otros sistemas.

- Es un **lenguaje potente y eficiente**. Usando C, un programador puede casi alcanzar la eficiencia del código ensamblador junto con la estructura del Algol o Pascal.

Como desventajas habría que reseñar que es más complicado de aprender que otros lenguajes como Pascal o Basic y que requiere una cierta experiencia para poder aprovecharlo a fondo.

USO DEL C

Los pasos a seguir desde el momento que se comienza a escribir el programa C hasta que se ejecuta son los siguientes:

1.- Escribirlo en un editor.

2.- Compilarlo en un compilador.

3.- Enlazarlo en un enlazador.

4.- Ejecutarlo.

Paso 1: ESCRIBIRLO

El programa se puede escribir en cualquier editor que genere ficheros de texto estándar, esto es, que los ficheros generados no incluyan códigos de control y caracteres no imprimibles.

Estos ficheros que contienen código C se llaman **ficheros fuentes**. Los ficheros fuentes son aquellos que contienen código fuente, es decir, ficheros con texto que el usuario puede leer y que son utilizados como entrada al compilador de C.

Los programas pequeños suelen ocupar un solo fichero fuente; pero a medida que el programa crece, se va haciendo necesario distribuirlo en más ficheos fuentes.

Paso 2: COMPILARLO

El compilador produce **ficheros objetos** a partir de los ficheros fuentes. Los ficheros objetos son los ficheros que contienen código objeto, es decir, ficheros con código máquina (número binarios que tiene significado para el microprocesador) y que son utilizados como entrada al enlazador.

La extensión de estos ficheros es OBJ, aunque también los hay con extensión LIB. A estos últimos se les llama también **ficheros de librería o biblioteca**; contienen código máquina perteneciente a código compilado suministrado por el compilador.

Paso 3: ENLAZARLO

El enlazador produce un **fichero ejecutable** a partir de los ficheros objetos.

Los ficheros ejecutables son aquellos que contienen código máquina y se pueden ejecutar directamente por el sistema operativo.

La extensión estos ficheros es EXE o COM.

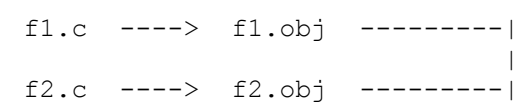
Al proceso de enlazado también se le suele llamar el proceso de linkado.

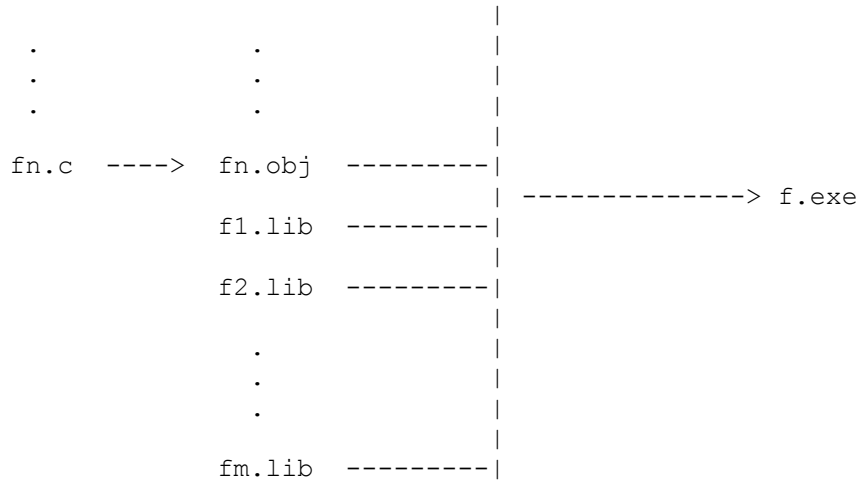
Paso 4: EJECUTARLO

El programa se puede ejecutar simplemente tecleando su nombre desde la línea de comandos del sistema operativo.

ESQUEMA

Los pasos anteriores se resumen en el siguiente esquema:





Hoy día los compiladores de C son muy sofisticados e incluyen **entornos integrados** desde los cuales editamos, compilamos, enlazamos, y podemos realizar una multitud de servicios más.

En algunos de ellos se pueden realizar los pasos de compilado, enlazado y ejecutado con la pulsación de una sola tecla.

En Turbo C tenemos las siguientes teclas relacionadas con este tema:

ALT-F9: Compilar a OBJ.

F9: Hacer fichero EXE.

CTRL-F9: Ejecutar.

Se puede pulsar directamente CTRL-F9 para compilar, enlazar y ejecutar.

partir

En programación, la experiencia es el gran maestro. Por ello es conveniente empezar a hacer programas en C cuanto antes.

A continuación se van a presentar varios programas completos en C muy sencillos para ir familiarizóndonos en la programación de este lenguaje.

NUESTRO PRIMER PROGRAMA C

```
#include <stdio.h>

main ()
{
  printf ("Mi primer programa en C.");
}
```

La salida de este programa por pantalla es:

Mi primer programa en C.

Analicemos a continuación nuestro primer programa.

Los programas C están compuestos de unidades de programa llamadas **funciones**,

las cuales son los módulos básicos del programa.

En este caso, el programa está compuesto por una sola función llamada **main**.

Todos los programas C deben tener una función main (en español significa principal) pues es la primera función que se ejecuta cuando se hace funcionar el programa.

Los **paréntesis** que siguen a main identifican a ésta como nombre de función.

Un método de comunicación de datos entre funciones es el uso de **argumentos**. Los argumentos son los datos que se les pasa a las funciones. Estos se encierran entre paréntesis; main es una función sin argumentos, lo que se indica mediante ().

La línea con:

```
printf ("Mi primer programa en C.");
```

realiza una **llamada a una función** denominada **printf**, con el argumento "Mi primer programa en C."; printf es una función de biblioteca que realiza una escritura en la salida estándar. Normalmente la salida estándar es el monitor.

La función printf escribe concretamente una **cadena** (conocido también con los nombres de cadena de caracteres, constante de caracteres y string).

Una cadena es una secuencia de caracteres.

Cada instrucción en C termina con un punto y coma. La línea del main () no va seguida de punto y coma porque no se trata de una llamada a una función sino de la **definición de una función.** La definición de una función especifica las instrucciones que posee dicha función asi como los argumentos que tiene.

printf es una función de librería que está definida en el fichero **stdio.h** (standard input/output header). Las funciones de librería son aquellas suministradas por el compilador y que están a nuestra disposición. Para utilizar una función de librería es necesario incluir el correspondiente fichero de cabecera antes de utilizarla.

Cada fichero de librería incluye muchas funciones. Por ejemplo, **la librería stdio.h define funciones de entrada y salida estándar.** Pero en el fichero ejecutable de nuestro programa sólo están las funciones de librería que hemos utilizado. De esta forma el tamaño del código ejecutable se reduce al mínimo.

La función

```
main ()
{
  printf ("Mi primer programa en C.");
}
```

también se podía haber escrito así:

```
main(){printf("Mi primer programa en C.");}
```

pero en este caso el código es menos legible para el usuario.

El C es sensintivo al caso. Esto significa que las letras mayúsculas son distintas a las minúsculas. De este modo, no es lo mismo para el C printf que PRINTF o que Printf.

Todas las líneas que empiezan con **#** no son en realidad instrucciones del lenguaje C sino que son líneas para ser manipuladas por el **preprocesador**. El preprocesador realiza algunas tareas antes de empezar a actuar el compilador.

La línea

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

lo que hace es **incluir** la información que hay en el fichero stdio.h en nuestro programa.

En el fichero stdio.h se encuentra la definición de la función printf. Si no se pusiera este include en nuestro programa, el compilador no sabría cómo es la función printf y daría error.

Resumen del análisis del programa:

Nuestro programa está compuesto de una sola función: la función main. Esta función siempre debe aparecer en los programas C pues es la primera función que se ejecuta. La función main en este caso no tiene argumentos.

Esta función está compuesta de una sola instrucción: llamada a la función printf para que escriba una cadena en pantalla.

La definición de la función printf está en el fichero stdio.h. Por lo tanto, hay que incluir (con #include) este fichero para poder utilizar la función printf.
Veamos ahora un segundo ejemplo algo más complicado que el anterior.

NUESTRO SEGUNDO PROGRAMA C

```
#include <stdio.h>

main ()
{
  printf ("Mi segundo programa en C.\n");
  printf ("Pulsa la tecla RETURN para terminar.");
  getchar ();
}
```

La salida por pantalla de este programa es:

```
Mi segundo programa en C.
Pulsa la tecla RETURN para terminar.
```

Analicemos a continuación nuestro segundo programa.

Hay dos novedades con respecto al primer ejemplo: la primera es la aparición del código \n dentro de la cadena del primer printf. La segunda es la aparición de una nueva función de librería: getchar ().

En C, todo lo que va entre comillas es una cadena. Ya dijimos antes que una cadena es una secuencia de caracteres. La secuencia **\n** es un carácter especial que denota el carácter de nueva línea. Al ser \n un carácter se puede incluir en cualquier parte de una cadena como cualquier otra carácter. En nuestro programa, si no se hubiese incluido, la salida habría sido:

```
Mi segundo programa en C. Pulsa la tecla RETURN para terminar.
```

A continuación se van a mostrar tres programas equivalentes al del ejemplo.

```
#include <stdio.h>

main ()
{
  printf ("Mi segundo programa en C.");
  printf ("\nPulsa la tecla RETURN para terminar.");
  getchar ();
}
```

```
#include <stdio.h>

main ()
{
  printf("Mi segundo programa en C.\nPulsa la tecla RETURN para terminar.");
  getchar();
}
```

```
#include <stdio.h>

main ()
{
  printf ("Mi segundo programa en C.");
  printf ("\n");
  printf ("Pulsa la tecla RETURN para terminar.");
  getchar ();
}
```

A todos los caracteres empezados por **** se les llaman **secuencias de escape**.

Las secuencias de escape son mecanismos para representar caracteres no imprimibles.

Antes vimos la secuencia de escape **\n** que representaba a la nueva línea.

Otras secuencias de escape son **\r** para el retorno de carro, **\t** para el tabulador, **\b** para retroceso, **\"** para la comilla, **\'** para el apóstrofe y **** para la barra diagonal invertida.

Si nuestro primer programa lo ejecutamos desde un entorno integrado, en muchos de estos entornos, a la finalización de la ejecución de nuestro programa C, la frase impresa desaparece inmediatamente y se vuelve a la pantalla del entorno.

Todos estos entornos poseen mecanismos para visualizar la pantalla de ejecución.

En Turbo C, pulsando **ALT-F5** se puede ver la pantalla de ejecución.

Una solución alternativa es incluir la función getchar () al final de la función main ().

getchar () es una función que espera la pulsación de la tecla return por parte del usuario.

Esta función no necesita argumentos pero los paréntesis son necesarios puesto que se trata de una función.

getchar () se encuentra en la librería stdio.h, por lo tanto, siempre que utilicemos esta función en un programa es necesario incluir la línea:

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

Veamos nuestro último programa ejemplo de esta lección.

NUESTRO TERCER PROGRAMA C

```
#include <stdio.h>
main () /* Tercer ejemplo */
{
  int horas, minutos;
  horas = 3;
  minutos = 60 * horas;
  printf ("Hay %d minutos en %d horas.", minutos, horas);
  getchar ();
}
```

La salida por pantalla de este programa es:

Hay 180 minutos en 3 horas.

Analicemos a continuación nuestro tercer programa.

En C, todo aquello que vaya entre un **/*** y un ***/** es ignorado. Las secuencias /* y */ denotan el principio y el final de un comentario en C. Se deben utilizar comentarios en los programas para hacerlos más comprensibles.

La línea:

```
  int horas, minutos;
```

es una **sentencia de declaración.**

En este caso se declaran dos cosas:

1) En algún sitio de la función se utilizarán las "variables" hora y minutos.

2) Estas dos variables son de tipo entero (integer).

El punto y coma final de la línea de declaración la identifican como una sentencia o instrucción C.

También se podría haber escrito:

```
int horas;
int minutos;
```

Las **variables** son posiciones de memoria donde el valor de su contenido puede variar a lo largo del programa.

Nos la podemos imaginar como cajas donde podemos meter cualquier cosa que le venga bien a la caja.

En C, todas las variables utilizadas ha de ser declaradas antes de su uso. Las líneas

```
  horas = 3;
  minutos = 60 * horas;
```

son **sentencias de asignación.**

La primera línea significa: "dar a la variable horas el valor 3". La segunda línea significa: "dar a la variable minutos el resultado de multiplicar 60 por horas". Nótese que las dos líneas terminan en punto y coma por ser dos sentencias o instrucciones.

En la línea

```
  int horas, minutos;
```

se reserva espacio en memoria a las variables horas y minutos.

En las líneas

```
  hora = 3;
  minutos = 60 * horas;
```

se les da valor a dichas variables (al contenido de los espacios reservados). Posteriormente se les puede asignar a estas variables valores diferentes. **=** es el operador de asignación y ***** es el operador de multiplicación.

Otros operadores son: **+** (número positivo usado como operador unario y suma usado como operador binario), **-** (número negativo usado como operador unario y substracción usado como operador binario), **/** (operador de división), **%** (operador módulo, esto es, resto de la división de dos números enteros).

En este momento, se mencionan estos operadores para empezar a hacer pequeños programas. En lecciones ulteriores se verán en detalle todos los operadores.

La línea

```
  printf ("Hay %d minutos en %d horas.", minutos, horas);
```

escribe:

Hay 180 minutos en 2 horas.

Como se ve los dos %d no se han escrito y sin embargo se ha escrito en su lugar los valores de las variables minutos y horas.

El símbolo **%** avisa a la función printf que se va a imprimir una variable en esta posición; la letra **d** informa que la variable a imprimir es entera (digit).

printf significa escritura (print) con formato (format) porque nos permite formatear la salida a nuestro gusto.

La estructura de toda función C es:

```
  {
    declaración_de_variables
    sentencias
  }
```

donde declaración_de_variables es una lista del tipo:

```
  tipo lista_de_variables;
```

y lista_de_variables es uno o más nombres de variables separados por comas.

RECUERDA: La declaración de variables ha de ir al principio de la función, antes de la primera sentencia ejecutable.

Si no has entendido algo en los tres ejemplos vistos, no te preocupes, pues todo lo que hay ellos se va a estudiar en profundidad en lecciones posteriores.

Con estos ejemplos lo que se ha pretendido es empezar a hacer programas completos en C desde un primer momento, intentando ofrecer una visión global de éste.

LECCIÓN 2

INTRODUCCION A LA LECCION 2

En esta lección vamos a hacer un estudio completo sobre lo tipos, operadores y expresiones del lenguaje C. Además profundizaremos un poco más en el estudio de la función printf a medida que vaya siendo necesario.

A modo de introducción vamos a dar unas definiciones muy breves que serán ampliadas a lo largo de toda la lección:

- Las variables y constantes son los objetos básicos que se manipulan en un programa.
- Las declaraciones indican las variables que se van a utilizar y establecen su tipo y, quizá, su valor inicial.
- Los operadores especifican lo que se va a hacer con ellas.
- Las expresiones combinan variables y constantes para producir nuevos valores.

DATOS

Los programas funcionan con datos.

Los **datos** son los números y
los caracteres que contienen la
información a utilizar.
Una primera división de los
datos la podemos hacer en
constantes y variables.
Las **constantes** son datos con
valores fijos que no pueden ser
alterados por el programa.
Las **variables** son datos cuyo
valor se puede cambiar a lo
largo del programa.
Una segunda división de los datos la podemos
hacer según los tipos de que sean.

TIPOS DE DATOS

Existen cinco tipos de datos básicos en C:

Tipo	Descripción	Longitud en bytes	Rango
char	carácter	1	0 a 255
int	entero	2	-32768 a 32767

float coma flotante	4	aproxim. 6 dígitos de precisión
double coma flotante de doble precisión	8	aproxim. 12 dígitos de precisión
void sin valor	0	sin valor

NOTA IMPORTANTE: La longitud en bytes, y por consiguiente, también el rango, de la tabla anterior, dependen de cada tipo de procesador y de cada compilador de C. No obstante, la información reseñada en la tabla es correcta para la mayoría de los ordenadores.

TIPO CARACTER

En C, los caracteres se definen con apóstrofos.

Ejemplos válidos de constantes tipo carácter: 'T', 'l', '1'.
Ejemplos inválidos de constantes tipo carácter: 'TT', l, 1.

'TT' es incorrecto porque hay dos caracteres entre los apóstrofos; l es incorrecto porque el compilador lo interpreta como una variable; el 1 lo interpreta el compilador como un número y no como un carácter.

El valor de una constante carácter es el valor numérico del carácter en el conjunto de caracteres del sistema. Por ejemplo, en el conjunto ASCII, el carácter cero, o '0', es 48, y en EBCDIC '0' es 240, ambos muy diferentes del valor numérico 0.

A lo largo de este tutor se utilizará el código ASCII y no el EBCDIC a fin de utilizar ejemplos concretos.

Ejemplos de asignación de este tipo de datos:

```
char ch1, ch2; /* declaración de las variables ch1 y ch2 */
ch1 = 'A'; /* a la variable ch1 se le asigna el valor ascii de 'A': 65 */
ch2 = 65; /*a la variable ch2 se le asigna el código ASCII 65 que es 'A'*/
```

Las dos asignaciones anteriores son equivalentes pero es preferible la primera asignación porque es más portátil. A ch1 se le asigna en cualquier ordenador el carácter 'A'. Sin embargo, a la variable ch2, en sistemas basados en código ASCII se le asigna el carácter 'A' (el código ASCII de 65 es 'A'), y en sistemas basados en código EBCDIC se le asigna un carácter distinto a 'A'.

Todas las variables en C han de ser declaradas antes de poder ser usadas.

La forma general de declaración es la siguiente:

```
tipo lista_variables;
```

Aquí, tipo debe ser un tipo de datos válido de C y lista_variables puede consistir en uno o más nombres de identificadores separados por comas. Las declaraciones deben estar antes de la primera sentencia ejecutable.

Ejemplos de declaraciones:

```
int i, j, k;
char caracter;
```

Los identificadores en C son los nombres usados para referenciar las variables, las funciones y otros objetos definidos por el usuario. Los nombres de los identificadores están compuestos por letras, dígitos y el carácter de subrayado (_). El número de caracteres significativos de los identificadores depende del compilador. El primer carácter de un

identificador ha de ser letra o el carácter de subrayado.

En Turbo C, el número de caracteres significativos por defecto es 32.

Ciertos caracteres no imprimibles se representan como constantes de carácter mediante secuencias de escape.

En la siguiente tabla se muestran las secuencias de escape del ANSI C:

Código	Significado
\b	retroceso
\f	salto de página
\n	nueva línea
\r	retorno de carro
\t	tabulación horizontal
\"	comillas (")
\'	apóstrofo (')
\0	carácter nulo
****	barra invertida (\)
\v	tabulación vertical
\a	alerta (bell, campanilla)
\ddd	constante octal (ddd son tres dígitos como máximo)
\xddd	constante hexadecimal (ddd son tres dígitos como máximo)

Hay otros caracteres no imprimibles que no tienen correspondencia en la tabla anterior. Estos caracteres se pueden utilizar mediante los códigos \ddd, \xddd o simplemente usando el número del código ASCII.

Ejemplo de asignaciones equivalentes:

```
char ch1, ch2, ch3, ch4; /*declaración de cuatro variables tipo carácter*/
ch1 = '\n';   /* el carácter '\n' es el número 13 en ASCII */
ch2 = 13;     /* 13 decimal <=> 12 octal <=> A hexadecimal */
ch3 = '\12'; /* también sería válido: ch3 = '\012'; */
ch4 = '\xA';  /* también sería válido: ch4 = '\xa'; */
```

La notación preferida es la primera. Aunque para los caracteres no imprimibles que no tienen correspondencia en la tabla anterior, la única solución es una una de las tres últimas asignaciones del ejemplo.

TIPO ENTERO

Es un número sin parte fraccionaria.

Las constantes enteras se pueden escribir de uno de los tres modos siguientes:

- En decimal: escribiendo el número sin empezar por 0 (a excepción de que sea el propio 0). Ejemplos: 1, 0, -2.

- En hexadecimal: empezando el número por 0x. Ejemplos: 0xE, 0x1d, 0x8.

- En octal: empezando el número por 0. Ejemplos: 02, 010.

TIPOS FLOAT Y DOUBLE

Las constantes de este tipo tienen parte real y parte fraccionaria.

El tipo double tiene doble precisión que el tipo float. Por lo demás, los dos tipos son iguales.

La sintaxis correcta de las constantes de estos dos tipos es:

[signo] [dígitos] [.] [dígitos] [exponente [signo] dígitos]

donde

signo es + o -;
dígitos es una secuencia de dígitos;
. es el punto decimal;
exponente es E o e.

Los elementos que están entre [] son opcionales, pero el número no puede empezar por e o E, ya que el compilador lo interpretaría en este caso como un identificador y no como un número.

Algunos ejemplos de constantes de este tipo: 1.0e9, -3E-8, -10.1.

TIPO VOID

Significa sin valor, sin tipo.

Uno de los usos de void se puede observar al comparar estos dos programas:

```
#include <stdio.h>
main ()
{
  printf ("Versión 1.");
  getchar ();
}
```

```
#include <stdio.h>
void main (void)
{
  printf ("Versión 2.");
  getchar ();
}
```

Al poner void entre los paréntesis de la definición de una función, se define a ésta como función que no tiene argumentos. No confundir con llamada a función, en cuyo caso no se puede utilizar el void.

Del mismo modo, al poner void antes del nombre de la función en la definición de ésta, se está declarando como función que no devuelve nada.

La segunda versión es preferible y es la que se utilizará a lo largo de todo el tutor.

PROGRAMA EJEMPLO

```
#include <stdio.h>

void main (void)
{
  int i = 1;
  char c = 'c';
```

```
  float f = 1.0;
  double d = 1e-1;
  printf ("  i = %d\n  c = %c\n  f = %f\n  d = %lf\n", i, c, f, d);
  getchar ();
}
```

La salida de este programa es:

```
  i = 1
  c = c
  f = 1.000000
  d = 0.100000
```

Como se puede observar en el programa, se puede asignar un valor a una variable en el momento de la declaración.

En la lección 1 ya se dijo que **%d** indica a la función printf el lugar en que se ha de escribir una variable de tipo entera. Los códigos **%c, %f** y **%lf** indican a la función printf el lugar en la cadena de caracteres en la que se han de escribir variables de tipo char, float y double respectivamente.

MODIFICADORES

A excepción del tipo void, los tipos de datos básicos pueden tener varios modificadores precediéndolos.

Hay modificadores de tipo y de acceso.

MODIFICADORES DE TIPO

Un **modificador de tipo** se usa para alterar el significado del tipo base para que se ajuste más precisamente a las necesidades de cada momento.

Modificadores de tipo:

Modificador	Descripción	Tipos a los se les puede aplicar el modificador
signed	con signo	int, char
unsigned	sin signo	int, char
long	largo	int, char, double
short	corto	int, char

El uso de signed con enteros es redundante aunque esté permitido, ya que la declaración implícita de entero asume un número con signo.

El estándar ANSI elimina el long float por ser equivalente al double. Sin embargo, como se puede observar en el último ejemplo visto, para escribir un double con la función printf es necesario utilizar el código de formato de printf: %lf; que significa: long float.

Se puede utilizar un modificador de tipo sin tipo; en este caso, el tipo se asume que es int.

La longitud (y por tanto, también el rango) de los tipos dependen del sitema que utilicemos; no obstante, la siguiente tabla es válida para la mayoría de

sistemas:

Tipo	Longitud en bytes	Rango
char	1	Caracteres ASCII
unsigned char	1	0 a 255
signed char	1	-128 a 127
int	2	-32768 a 32767
unsigned int	2	0 a 65535
signed int	2	Igual que int
short int	1	-128 a 127
unsigned short int	1	0 a 255
signed short int	1	Igual que short int
long int	4	-2147483648 a 2147483649
signed long int	4	-2147483648 a 2147483649
unsigned long int	4	0 a 4294967296
float	4	Aproximadamente 6 dígitos de precisión
double	8	Aproximadamente 12 dígitos de precisión
long double	16	Aproximadamente 24 dígitos de precisión

MODIFICADORES DE ACCESO

Acabamos de hablar de los modificadores de tipos y hemos dicho que modifican el tipo básico. También hay otra clase que son los **modificadores de acceso.** Como su propio nombre indica, estos modificadores modifican el acceso a los tipos.

Estos modificadores son:

Modificador	Descripción
const	constante
volatile	volátil

Las variables de tipo **const** son aquéllas a las que se les asigna un valor inicial y este valor no puede ser cambiado a lo largo del programa. Se utilizan para declarar constantes.

Ejemplo de declaración de una constante:

```
const unsigned int hola;
```

Las variables de tipo **volatile** previenen al compilador que dicha variable puede ser cambiada por medios no explícitamente especificados en el programa.

Obsérvese las siguientes setencias C:

```
int v;
v = 1;
v = 2;
```

En estos casos, los compiladores suelen optimizar el código y la primera sentencia de asignación se desecha y no se genera código para ella ya que es redundante.

Si se hubiese declarado la variable v como volatile:

```
volatile v;
```

la optimización descrita no se realizaría sobre la variable v, generándose código para las dos asignaciones.

En C existen tipos derivados.
Los **tipos derivados** son
aquéllos, que como su propio
nombre indica, derivan de los
tipos básicos.
A continuación vamos a hablar de
un tipo derivado muy común en C:
las **cadenas de caracteres**.

CADENAS DE CARACTERES

Una **cadena de caracteres** (también conocido por el nombre de **string**) es una secuencia de caracteres encerrados entre comillas.

Ejemplos de cadenas:

"El amor es como la luna: cuando no crece es que mengua."
"abc"
"a"
"\n a \n b \n c \n"

Las comillas no forman parte de la secuencia. Sirven para especificar el comienzo y final de ésta, al igual que los apóstrofos marcaban los caracteres individuales.

Las cadenas son tratadas por el C como un array de caracteres.

Un **array** (conocido también con el nombre de **vector**) es una secuencia de datos que se encuentran almacenados en memoria de una forma consecutiva.

Un **array de caracteres** es una secuencia de caracteres.

El array:

"abc"

se almacenaría en memoria como:

```
------------------
| a | b | c | \0 |
----Á---Á---Á-----
```

El carácter '\0' se llama carácter nulo, y es utilizado por el C para marcar el final de la cadena en memoria.
El carácter nulo ('\0') no es la cifra 0 (cuyo código ASCII es 48), sino un carácter no imprimible, cuyo código ASCII es 0.

Ejemplo:

```
void main (void)
{
  char ch1, ch2, ch3, ch4;
  ch1 = '\0'; /* en memoria, ch1 tiene el valor 0 que es el valor ASCII
                 correspondiente al carácter '\0'                        */
  ch2 = 0;    /* en memoria, ch2 tiene el valor 0                        */
  ch3 = '0';  /* en memoria, ch3 tiene el valor 48 que es el valor ASCII
                 correspondiente al carácter '0'                         */
  ch4 = 48;   /* en memoria, ch4 tiene el valor 48                       */
}
```

Notad que en este programa no se ha puesto: #include <stdio.h>. Esto se debe a que nuestro programa no utiliza ninguna información contenida en dicha librería.

En las asignaciones anteriores, a ch1 y ch2 se les asigna el carácter nulo, pero a las variables ch3 y ch4 se le asigna el carácter cero.

Teniendo en cuenta el carácter nulo en la cadena "abc", esta cadena es un array de tamaño 4 (tres caracteres más el carácter nulo).

Obsérvese el siguiente programa.

```
#include <stdio.h>

void main (void)
{
  printf ("\nInvertir en conocimientos produce siempre los mejores "
          "intereses.\n(%s)", "Benjamín Franklin");
}
```

La salida de este programa en pantalla es la siguiente:

```
Invertir en conocimientos produce siempre los mejores intereses.
(Benjamín Franklin)
```

En este ejemplo podemos observar la aparición de dos cosas nuevas: la división de una cadena de caracteres en varias líneas y el código de formato %s.

El código **%s** le indica a la función printf que escriba una cadena en su lugar.

Una cadena de caracteres se puede escribir en varias líneas de fichero cerrando las comillas al final de la línea y volviéndolas a abrir en la línea siguiente. Haciéndolo de este modo, el compilador lo interpreta como una sola cadena de caracteres escrita en varias líneas. En C, los finales de instrucciones no se detectan con los finales de línea sino con puntos y comas al final de cada instrucción.

La sentencia:

```
  printf ("\nInvertir en conocimientos produce siempre los mejores "
          "intereses.\n(%s)", "Benjamín Franklin");
```

también se puede escribir, por ejemplo, del siguiente modo:

```
  printf (
           "\nInvertir "
           "en conocimientos produce "
           "siempre los mejores"
           " intereses."
           "\n(%s)",
           "Benjamín "
           "Franklin"
         );
```

Yo, personalmente, no sé porqué!, prefiero la primera versión a la segunda.

Conviene hacer la observación que **'x' es distinto de "x"**. 'x' es una constante carácter. "x" es una cadena de caracteres.

'x' pertenece a un tipo básico (char). "x" es de un tipo derivado (array compuesto de elementos del tipo básico char). "x", en realidad, contiene dos caracteres, a saber, 'x' y '\0'.

El estudio de los array se estudiará en profundidad en lecciones posteriores; pero para comprender un poco mejor el concepto de array de caracteres, vamos a hablar un poco más de ellos.

Una variable array se declara:

```
tipo_de_cada_elemento variable_array [numero_de_elementos_del_array];
```

y a cada elemento del array se accede:

```
variable_array [numero_de_elemento];
```

Es muy importante tener siempre en mente que al primer elemento de un array se accede mediante:

```
variable_array [0];
```

y al segundo elemento:

```
variable_array [1];
```

y así sucesivamente para el acceso al resto de los elementos del array.

Programa ejemplo:

```
#include <stdio.h>

void main (void)
{
  int x[2]; /* se reserva memoria para dos elementos de tipo int */
  x[0] = 10;
  x[1] = 11;
  /* el elemento x[2] no existe, mejor dicho, no se ha reservado memoria */
  printf ("\nx[0] = %d\nx[1] = %d\n", x[0], x[1]);
}
```

Si en el programa anterior se hubiese hecho:

```
   x [2] = 3;
```

el compilador probablemente compilará sin problemas y no nos informará de ningún error. PERO AL EJECUTAR EL PROGRAMA, EL VALOR 3 SE ESCRIBIRA EN UNA POSICION DE MEMORIA NO ASIGNADA; ESTO PUEDE PRODUCIR RESULTADOS INESPERADOS; pensad que el valor 3 se podría escribir sobre el código del sistema operativo o cualquier otro programa que esté en memoria en ese momento.

Al hacer la declaración

```
int x [2];
```

estamos reservando memoria para x[0] y x[1], dicho de otro modo, int x[2] reserva memoria para dos elementos.

Repitimos: hemos hablado de los arrays lo mínimo para poder entender los arrays de caracteres; más adelante, en otra lección, se va hablar a fondo sobre los arrays.
partir

OPERADORES

Un **operador** es un símbolo que realiza una

determinada operación sobre sus operandos.
Un **operando** es el dato que va a
ser manipulado por el operador.

Los operadores en C se pueden dividir en cuatro grupos:

a) Operadores aritméticos.
b) Operadores relacionales y lógicos.
c) Operadores a nivel de bits.
d) Operadores especiales.

EXPRESIONES

Las **expresiones** contienen variables y
constantes para producir nuevos valores.

OPERADORES ARITMETICOS

Los **operadores aritméticos** realizan operaciones aritméticas.

Son los siguientes:

Operador	Acción
-	Resta, también menos monario
+	Suma, también suma monaria
*****	Multiplicación
/	División
%	División en módulo
--	Decremento
++	Incremento

Los operadores de incremento y decremento solo se pueden aplicar a
variables, no constantes. El de incremento añade 1 a su operando y
el de decremento resta 1. En otras palabras,

```
++x; o x++; es lo mismo que x = x + 1;
```

y

```
--x; o x--; es lo mismo que x = x - 1;
```

Los operadores de incremento y decremento pueden preceder o seguir al
operando. Si el operador precede al operando, C lleva a cabo la operación
antes de utilizar el valor del operando. Si el operador sigue al operando,
C utilizará su valor antes de incrementarlo o decrementarlo. Esto se ve
muy bien en los dos ejemplos siguientes:

```
int x, y;
x = 2;
y = ++x;
/* ahora x tiene el valor 3
   e y tiene el valor 3 */
```

```
int x, y;
x = 2;
y = x++;
/* ahora x tiene el valor 3
   e y tiene el valor 2 */
```

La precedencia de los operadores aritméticos es la siguiente:

```
MAYOR        ++  --
             + (más monario)  - (menos monario)
```

```
                        *  /  %
              MENOR     +  -
```

Los operadores del mismo nivel de precedencia son evaluados por el compilador de izquierda a derecha. Se puede alterar el orden de evaluación utilizando paréntesis.

Ejemplo:

```
void main (void)
{
  int x1, x2, x3, x4, x5, x6;
  /* Asignaciones             */ /* Orden de asignaciones                */
  x1 = 2 + 3 * 4;                /* x1 = 14;                             */
  x2 = (2 + 3) * 4;              /* x2 = 20;                             */
  x3 = -4 - (-1);                /* x3 = -3;                             */
  x4 = 10 / 2 % 3;               /* x4 = 2;                              */
  x5 = ++x3 - x4;                /* x3 = -2; x5 = -4;                    */
  x6 = x3++ - x4;                /* x6 = -4; x3 = -1;                    */
  x1 = -x1;                      /* x1 = -14;                            */
  x2 = (x1 + x2) / x3;           /* x2 = -6;                             */
  x3 = ((x1++) + (x2++)) - x3;   /* x3 = -19; x1 = -13; x2 = -5;         */
  x4 = -(-(-x3));                /* x4 = 19;                             */
  x5 = (x6 * x6 + x6 / x6);      /* x5 = 17;                             */
  x6 = (x1++) + (++x2) - (++x6); /* x2 = -4; x6 = -3; x6 = -14; x1 = -12; */
  x1++;                          /* x1 = -11;                            */
  --x2;                          /* x2 = -5;                             */
}
```

OPERADORES RELACIONALES Y LOGICOS

La palabra relacional se refiere a la relación entre unos valores y otros. La palabra lógico se refiere a las formas en que esas relaciones pueden conectarse entre sí.

Los vamos a ver juntos porque ambos operadores se basan en la idea de **cierto** (true en inglés) y **falso** (false en inglés). **En C, cualquier valor distinto de cero es cierto, y el valor 0 es falso.** Las expresiones que son ciertas toman el valor de 1 y las que son falsas toman el valor de 0.

Los operadores son:

Operadores relacionales

Operador	Acción
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
!=	No igual

Operadores lógicos

Operador	Acción
&&	Y
||	O
!	NO

Tabla de verdad para los operadores lógicos:

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	0	1	1	0
1	1	0	1	0

Precedencia de estos operadores:

```
        MAYOR     !
                  >  >=  <  <=
                  ==  !=
                  &&
        MENOR     ||
```

Ejemplo:

```
void main (void)
{
  int x1, x2, x3, x4, x5, x6;
  /* Asignaciones          */ /* Orden de asignaciones */
  x1 = 10 < 12;               /* x1 = 1;               */
  x2 = 10 > 12;               /* x2 = 0;               */
  x3 = -1 && 5;               /* x3 = 1;               */
  x4 = 0 || x3;               /* x4 = 1;               */
  x5 = x1 >= x2 <= x3;        /* x5 = 1;               */
  x6 = x1 == x2 || x3 != x4; /* x6 = 0;               */
  x1 = !x1;                   /* x1 = 0;               */
  x2 = ! (!x1 || x3 <= x3);  /* x2 = 0;               */
  x3 = 1 && 0;                /* x3 = 0;               */
  x4 = 1 || 0;                /* x4 = 1;               */
  x5 = !(-10);                /* x5 = 0;               */
  x6 = !!x4;                  /* x6 = 1;               */
}
```

Una particularidad interesante del C es que la evaluación de una expresión se termina en cuanto se sabe el resultado de dicha expresión. Veámoslo con un ejemplo:

```
  0 && x
  1 || x
```

En las dos expresiones anteriores NO se evalúa x puesto que es superfluo: en la primera expresión al ser uno de los dos operandos 0, el otro no hace falta mirarlo; con la segunda expresión podemos decir lo mismo. Como los operadores && y || se evalúan de izquiera a derecha, podemos asegurar que es el segundo operando (el que contiene la x) el que no se valúa. Si la expresión fuera: x && 0, se valuaría la x, y si ésta es cierta se evaluaría el 0, y si la x fuera falsa, no se evaluaría el 0.

OPERADORES A NIVEL DE BITS

Estos operandos realizan operaciones sobre los bits de un byte o una palabra (dos bytes). Sólo se pueden utilizar con los tipos char e int.

Estos operadores son:

Operador	Acción
&	Y
|	O
^	O exclusiva (XOR)
	Complemento a uno (NOT)
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Las tablas de verdad de los operadores &, | e son las mismas que las tablas de verdad de los operadores &&, || y ! respectivamente; pero los operadores a nivel de bits trabajan bit a bit.
La tabla de verdad para el XOR es:

```
p q   p ^ q
0 0     0
1 0     1
1 1     0
0 1     1
```

Signifiquemos que los operadores relacionales y lógicos siempre producen un resultado que es 0 ó 1, mientras que las operaciones entre bits producen cualquier valor arbitrario de acuerdo con la operación específica. En otras palabras, las operaciones a nivel de bits pueden dar valores distintos de 0 ó 1, pero los operadores lógicos siempre dan 0 ó 1.

Ejemplo:

```
  char x, y, z1, z2;
  x = 2; y = 3; z1 = 2 && 3; z2 = 2 & 3; /* z1 = 1; z2 = 2 */
```

¨Por qué (2 && 3) es 1 y (2 & 3) es 2?
2&&3: el compilador evalúa la expresión 1 && 1 que es 1.
2&3: el compilador evalúa 00000010 & 00000011 que es 00000010 (2 en decimal)

Sintaxis para los operadores de desplazamiento:

expresión >> número de bits a desplazar a la derecha
expresión << número de bits a desplazar a la izquierda

Dos observaciones sobre los operadores de desplazamiento:

1) Un desplazamiento no es una rotación. O sea, los bits que salen por un extremo no se introducen por el otro.

2) Normalmente, a medida que se desplaza los bits hacia un extremo se va rellenando con ceros por el extremo opuesto. PERO NO EN TODOS LOS ORDENADORES ES ASI. Si queremos introducir ceros y que el programa sea portátil lo tenemos que hacer explícitamente con una operación and.

Tabla de precedencia:

```
    MAYOR
            <<  >>
            &
            ^
    MENOR   |
```

Ejemplos:

```
void main (void)
{
  char x, y;
  /* Asignaciones   x en bits   y en bits   valor de x   valor de y */
  /* ------------   ---------   ---------   ----------   ---------- */
  x = 2; y = 3;  /* 0000 0010   0000 0011        2            3     */
  y = y << 1;    /* 0000 0010   0000 0110        2            6     */
  y = x | 9;     /* 0000 0010   0000 1011        2           11     */
  y = y << 3;    /* 0000 0010   0101 1000        2           88     */
  x = x;        /* 1111 1101   0101 1000       -3           88     */
  x = 4 ^ 5 & 6; /* 0000 0000   0101 1000        0           88     */
}
```

OPERADORES ESPECIALES

Bajo este apartado se recogen una serie de operadores no agrupables en ninguno de los grupos anteriores.

Estos operadores son los siguientes:

?
& *
sizeof
,
. ->
() []

Veamos cada uno de estos operadores.

OPERADOR CONDICIONAL (?).

El operador ? tiene la forma general:

expresion_1 ? expresion_2 : expresion_3

donde expresion_1, expresion_2 y expresion_3 son expresiones C.

El operador ? actúa de la siguiente forma: Evalúa expresion_1. Si es cierta, evalúa expresion_2 y toma ese valor para la expresión. Si expresion_1 es falsa, evalúa expresion_3 y toma su valor para la expresión.

Ejemplo:

```
int x, y;
x = 2 < 3 ? 4 : 5; /* a x se le asigna el valor 4: x = 4; */
y = 2 > 3 ? 4 : 5; /* a y se le asigna el valor 5: x = 5; */
x = 1 < 2 ? (4 > 3 ? 2 : 3) : 5; /* a x se le asigna el valor 2: x = 2;
```

OPERADORES DE DIRECCION (&) Y DE CONTENIDO (*).

Estos dos operadores operan con punteros. Los dos son monarios.

Un puntero es una variable que contiene una dirección de memoria.

El significado que tiene en este caso el operador * no tiene absolutamente nada que ver con el que tiene el operador aritmético *. En el código fuente no hay confusión entre uno y otro pues el aritmético es binario y el de punteros es monario. Lo mismo ocurre con el operador &.

En este momento no vamos a decir nada más de estos dos operadores y de punteros. Ya llegará el momento más adelante.

OPERADOR sizeof.

El operador sizeof es un operador monario que toma el valor de la longitud, en bytes, de una expresión o de un tipo; en este último caso, el tipo ha de estar entre paréntesis.

Ejemplo:

```
  double d;
  int longitud_en_bytes_de_la_variable_d, longitud_en_bytes_del_tipo_char;
  longitud_en_bytes_de_la_variable_d = sizeof d;
  longitud_en_bytes_del_tipo_char = sizeof (char);
  /* en la mayoría de los sistemas: sizeof d es 8 y sizeof (char) es 1 */
```

OPERADOR COMA (,).

La coma (,) tiene dos usos muy distintos en C:

1) Para representar una lista de elementos:

Ejemplos:

```
  int a, b, c;
  printf ("%d%d%d", 1, 2, 3);
```

2) Como operador.

Como operador, la coma encadena varias expresiones. Estas expresiones son evaluadas de izquierda a derecha y el valor de la expresión total es el valor de la expresión más a la derecha.

Ejemplo:

```
  int x;
  x = (2, 3, 4); /* a x se le asigna el valor 4: x = 4; */
```

OPERADORES PUNTO (.) Y FLECHA (->).

Estos dos operadores se utilizan con dos tipos compuestos: estruct (estructura) y union (unión).

El significado, tanto de los tipos compuestos struct y union, como de los operadores . y ->, se estudiará en lecciones posteriores.

Aquí se han nombrado estos dos operadores para saber que existen y para hacer una tabla de precedencia con todos los operadores del C un poco más adelante, en esta misma lección.

OPERADORES PARENTESIS () Y CORCHETES [].

Los paréntesis se pueden considerar como operadores que aumentan la precedencia de las operaciones que contienen.

Ejemplo:

```
  int x = 1 - 2 * 3;   /* a x se le asigna el valor -5: x = -5; */
  int y = (1 - 2) * 3; /* a y se le asigna el valor -3: x = -3; */
```

Los paréntesis también se usan, dentro de una expresión, para especificar la llamada a una función.

Ejemplo:

```
  printf ("La inactividad sólo apetece cuando tenemos demasiado que hacer."
          "\n(No%l Coward)"); getchar ();
```

Los corchetes llevan acabo el indexamiento de arrays. Ya hemos hablado anteriormente un poco de ellos y se estudiará en detalle en otras lecciones.

Ejemplo:

```
  float f[3]; /* reserva memoria para tres float: f[0], f[1] y f[2] */
  f[0] = 1.1; f[1] = 2.2; f[2] = 3.3; /* tres asignaciones */
```

SENTENCIAS DE ASIGNACION

Una sentencia de asignación es una sentencia C
en la que se asigna un valor a una variable.
La forma general de la sentencia de asignación es:

nombre_variable operador_de_asignacion expresion;

donde operador_de_asignacion es uno de los operadores siguientes:

=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=.
Con el operador =, el significado
de la asignación es dar el valor
de la expresión de la parte derecha
a la variable que se encuentra en
la parte izquierda del operador =.

Ejemplo:

```
  int a, b, c;
  a = 5;
  b = 4 + a;
  c = (a * b) - 1;
```

Una sentencia de asignación es una expresión. El valor de esta expresión es el valor que se le asigna a la variable. El operador de asignación se evalúa de derecha a izquierda.

Ejemplo:

```
  int x, y;
  x = y = 2; /* el 2 se asigna primero a la y y después a la x */
```

Hemos dicho que el operador de asignación se evalúa de derecha a izquierda, así que primero se hace y = 2 y el valor de esta expresión es 2, este valor se asigna a x y el valor de la expresión total es 2.

También se podía haber hecho:

```
  x = (y = 2);
```

pero en este caso los paréntesis son innecesarios porque no cambian la precedencia.
El resto de operadores de asignación se exponen en la siguiente tabla:

Sentencia de asignación	**Sentencia de asignación equivalente**
x *= y;	**x = x * y;**
x /= y;	**x = x / y;**
x %= y;	**x = x % y;**
x += y;	**x = x + y;**
x -= y;	**x = x - y;**
x <<= y;	**x = x << y;**
x >>= y;	**x = x >> y;**
x &= y;	**x = x & y;**
x ^= y;	**x = x ^ y;**

x |= y; **x = x | y;**

x e y son dos variables.

INICIALIZACIONES DE VARIABLES

La forma de inicialización es:

tipo nombre_variable = expresión;

Ejemplo:

```
  float f = 3.0;
```

También se pueden inicializar varias variables separadas por comas en una misma sentencia.

Ejemplos:

```
  int x = 1; /* declaración e inicialización de x */
  char ch1 = 'a', ch2 = 'b'; /* declaración e inicialización de ch1 y ch2 */
  float f1 = 2.2, f2 = 3e3; /* declaración e inicialización de f1 y f2 */
  int x, y = 3, z;/* declaración de x, y, z pero inicialización sólo de y */
  double d = 1.1 - 2.2; /* declaración e inicialización de d */
  int a = 1 + 2, b = 4, c; /* declaración de a, b, c pero esta última no se
                              inicializa */
```

CONVERSION DE TIPOS

La conversión de tipos se refiere a la situación en la que se mezclan variables de un tipo con variables de otro tipo.

Cuando esto ocurre en una sentencia de asignación, la regla de conversión de tipo es muy fácil: el valor de lado derecho de la expresión se convierte al del lado izquierdo.

Ejemplo:

```
int x = 2.3; /* 2.3 se convierte a 2 */
char ch = 500; /* los bits más significativos de 500 se
                  pierden */
```

El tipo que resulta de aplicar un operador con dos operandos de tipos diferentes, es el tipo de mayor tamaño (mayor longitud en bytes).

Ejemplo:

```
  2 + 3.3; /* el valor de esta expresión es 5.3, o sea,
              un valor de tipo float */
```

Es posible forzar a que una expresión sea de un tipo determinado utilizando una construcción denominada **molde.**

La forma general de un molde es: **(tipo) expresión**

El molde se puede considerar como un operador monario teniendo la misma precedencia que el resto de los operadores monarios.

Ejemplos:

Expresión	Valor de la expresión	Tipo de la expresión

```
--------------  --------------------- --------------------
     3 / 2               int                    1
    3.0 / 2             float                 1.5
(float) (3 / 2)         float                 1.0
 (float) 3 / 2          float                 1.5
```

En la expresión (float) 3 / 2, al ser (float) un operador monario, tiene más prioridad que el operador binario /.

Una constante decimal, octal o hexadecimal seguida por la letra **l** o **L**, se interpreta como una constante long.

Las siguientes expresiones son equivalentes:

```
4l
4L
(long) 4
```

También se puede utilizar el tipo void en los moldes.

Por ejemplo:

```
(2 + 3); /* expresión entera */
(void) (2 + 3); /* expresión sin ningún tipo */
```

PRECEDENCIA DE OPERADORES

```
Mayor     ()  []  ->  .
          !    ++  --  -  (tipo)  *  &  sizeof
          *  /  %
          <<  >>
          <  <=  >  >=
          ==  !=
          &
          ^
          |
          &&
          ||
          ?
          =  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=
Menor     ,
```

Los operadores unarios, el condicional y los de asignación asocian de derecha a izquierda; los demás asocian de izquierda a derecha.
Veamos los pasos que se siguen al evaluar algunas expresiones:

```
Expresión 1: 10 < 5 && 8 >= 3
Paso 1:         0   && 8 >= 3
Paso 2:         0   &&   1
Paso 3:             0
```

```
Expresión 2: x = (y = 3), y
Paso 1: a la variable y se le asigna el valor 3
Paso 2: el valor de la expresión coma es y, o sea, 3
Paso 3: a la variable x se le asigna el valor 3
Paso 4: el valor de toda la expresión es x, es decir, 3
```

```
Expresión 3: 1 + 2 < 3 || 4 + 5 > 2
Paso 1:        3   < 3 || 4 + 5 > 2
```

```
Paso 2:            0   || 4 + 5 > 2
Paso 3:            0   ||   9   > 2
Paso 4:            0   ||       1
Paso 5:                 1

Expresión 4: (-3 < !2) >> (-3)
Paso 1:      (-3 < 0 ) >> (-3)
Paso 2:          1     >> (-3)
Paso 3:          1     >>   2
Paso 4:                0

Expresión 5: 2 < 3 < ((4 - 1) / !3)
Paso 1:        1   < ((4 - 1) / !3)
Paso 2:        1   < (   3    / !3)
Paso 3:        1   < (   3    / 0 )
Paso 4: Error: División por cero.

Expresión 6: (double) (int) (!-3 / (float) 2)
Paso 1:      (double) (int) ( 0  / (float) 2)
Paso 2:      (double) (int) ( 0  / 2.0)
Paso 3:      (double) (int)     0.0
Paso 4:      (double)   0
Paso 5:         0.0
```

LECCIÓN 3

INTRODUCCION A LA LECCION 3

En esta lección vamos a estudiar todas las sentencias que posee el C para cambiar el flujo del programa. Entre ellas están las sentencias condicionales if **y** switch; **las sentencias iterativas** while, for **y** do; **y las sentencias** break, continue, goto **y** return; **La sentencia return sólo se menciona aquí, pues se estudiará en la lección dedicada al estudio de las funciones de C.**

SENTENCIAS C

Una sentencia en C puede ser:

- Una **sentencia simple.**

```
printf ("Filosofía de Murphy: Sonría; mañana puede ser peor.");
y = 4 + 1; /* sentencia de asignación */
; /* sentencia que no hace nada */
```

- Una **sentencia compuesta.**

```
{ /* sentencia compuesta formada por dos sentencias simples */
  --x;
  printf ("Ley de Murphy: Si algo puede salir mal, saldrá mal.");
}
{ } /* sentencia que no hace nada */
```

Las sentencias if, switch, while, for y do son sentencias simples. Todas estas sentencias las veremos en esta lección. **Una sentencia compuesta está formada por ninguna, una o varias sentencias simples delimitadas entre llaves.** A las sentencias compuestas también reciben el nombre de **bloques.**Las sentencias simples son todas aquéllas que no son compuestas.
borrar_ventana

SENTENCIAS DE CONTROL:

La mayoría de las sentencias
de control de cualquier lenguaje
están basadas en condiciones.

Una condición es una expresión cuya
resolución da como resultado cierto
(true) o falso (false).

Muchos lenguajes de programación
incorporan los valores true y false;
en C cualquier valor distinto de cero
es true, y el valor cero es false.

SENTENCIAS CONDICIONALES

C posee dos sentencias condicionales: **if** y **switch**.

Además, el operador **?** es una posible alternativa para
if en ciertas situaciones.

Sentencia if

SINTAXIS

```
if (expresión)
  sentencia
```

o

```
if (expresión)
  sentencia_1
else
  sentencia_2
```

DESCRIPCION

Si expresión es cierta se ejecuta la sentencia
correspondiente al if. Si expresión es falsa se
ejecuta la sentencia correspondiente al else si
lo hay.

En la segunda sintaxis se ejecuta sentencia_1 o
sentencia_2, pero nuna ambas.

EJEMPLOS

```
if (contador < 50)
  contador++;
```

```
if (x < y)
  z = x;
else
  z = y;
```

```
if (ch == '\n')
  {
    numero_de_lineas++;
    numero_de_caracteres++;
  }
```

OBSERVACIONES

1) Lo siguiente es incorrecto:

```
if (expresión)
  {
    sentecias
  };
else
  sentencia
```

puesto que entre el if y el else sólo puede haber
una sentencia y aquí hay dos: { } y ;.
borrar_ventana
2) Al ser la sentencia if una sentencia simple,
la sentencias if se pueden anidar:

```
/*
   a la variable numero_menor se le asigna la
   variable con menor valor entre x, y, z
*/
if (x <= y)
  if (x <= z)
    numero_menor = x;
  else
    numero_menor = z;
else
  if (y <= z)
    numero_menor = y;
  else
    numero_menor = z;
```

borrar_ventana
3) El else siempre está asociado al if más cercano.
Los dos siguientes ejemplos son distintos:

```
/* Ejemplo 1: */
if (n > 0)
  if (a > b)
    z = a;
  else
    z = b;

/* Ejemplo 2: */
if (n > 0)
  {
```

```
      if (a > b)
        z = a;
    }
  else
    z = b;
```

borrar_ventana

4) Un constructor común en programación es la escala if-else-if. Tiene la forma siguiente:

```
if (expresión_1)
  sentencia_1
else if (expresión_2)
  sentencia_2
.
.
.
else
  sentencia_n
```

```
/* Ejemplo: numero_menor
toma el valor de la variable
menor entre x, y, z */
if (x <= y && x <= z)
 numero_menor = x;
else if (y <= z && y <= z)
 numero_menor = y;
else
  numero_menor = z;
```

Las condiciones se evalúan de arriba hacia abajo. Sólo se ejecuta la sentencia correspondiente a la primera expresión que sea cierta, si la hay. Si ninguna expresión es cierta, se ejecuta el else final si existe, sino no se ejecuta ninguna sentencia.

borrar_ventana

5) El operador ? se puede utilizar para reemplazar las sentencias if-else. Este operador ternario se ha visto en la lección 2.

```
/* Ejemplo 1 con if-else */
if (x <= y)
  z = x;
else
  z = y;
```

```
/* Ejemplo 1 con ?: */
z = x <= y ? x : y;
```

```
/* Ejemplo 2 con if-else */
if (n == 1)
  printf ("Mensaje 1");
else
  printf ("Mensaje 2");
```

```
/* Ejemplo 2 con ?: */
n == 1 ?
  printf ("Mensaje 1"):
  printf ("Mensaje 2");
```

Sentencia switch

FORMA GENERAL

```
switch (expresión)
  {
    case expresión_constante_1:
      sentencias_1
      break;
    case expresión_constante_2:
      sentencias_2
      break;
    case expresión_constante_3:
      sentencias_3
      break;
     .
     .
     .
    default:
      sentencias_n
  }
```

DESCRIPCION

En muchas ocasiones es más elegante utilizar la sentencia switch que la escala if-else-if.

Una expresión constante es una expresión en la que todos los operandos son constantes.

El switch evalúa expresión. A continuación evalúa cada una de las expresiones constantes hasta que encuentra una que coincida con expresión. Cuando la encuentra ejecuta las sentencias correspondientes a ese case. Si no hay ninguna expresión case que coincida con expresión, se ejecuta las sentencias correspondientes al default.

EJEMPLO

```
switch (operando)
  {
    case 1:
      x *= y;
      break;
    case 2:
      x /= y;
      break;
    case 3:
      x += y;
      break;
    case 4:
      x -= y;
      break;
    default:
      printf ("ERROR!");
  }
```

OBSERVACIONES

1) La sentencia default es opcional. Si fallan todas las comprobaciones de los case y no hay default, no se ejecuta ninguna acción en el switch. Incluso el default puede ir en cualquier posición y no obligatoriamente al final.

2) La sentencia switch se diferencia de la escala if-else-if en dos cosas:

  1§) En la sentencia switch sólo se puede comprobar la igualdad entre las expresiones constantes y expresión.

2§) Las expresiones de los case han de ser constantes.

borrar_ventana

3) Las sentencias break en el switch son opcionales. El break hace que se produzca una salida inmediata de la instrucción switch. Si no hay una sentencia break en un case, al ejecutarse las sentencias que corresponden a ese case, también se ejecutarían las sentencias correspondientes al siguiente case y así sucesivamente hasta encontrar un break o llegar al final del switch. La sentencia break puede ir en cualquier sitio, no forzosamente al final de las sentencias de los case.

```
case 1:  /* Ejemplo de case sin break  */
  x++;   /* Después de ejecutarse x++, */
case 2:  /* se ejecuta siempre y++     */
  y++;
  break;
```

borrar_ventana

4) Asociado a cada case puede haber ninguna, una o varias sentencias.

```
case 1: /* case sin sentencias directas*/
case 2: /* case con dos sentencias */
  ++x;
  break;

case 3:
  x++;
  break;
  y++;/*esta sentencia nunca se ejecuta*/

case 4: /* case con tres sentencias */
  x++;
  y++;
  break;
```

SENTENCIAS ITERATIVAS

Los bucles o sentencias iterativas permiten que un conjunto de instrucciones sea ejecutado hasta que se alcance una cierta condición.

Las sentencias iterativas son: **while**, **for** y **do**.

Sentencia while

SINTAXIS

```
while (expresión)
  sentencia
```

DESCRIPCION

Se evalúa expresión. Si es cierta, se ejecuta sentencia y se vuelve a evaluar expresión. El ciclo continúa hasta que expresión es falsa, momento en que la ejecución continúa con lo que está después de sentencia.

EJEMPLOS

```
/*                  /*                          /*
    el bucle termina    imprime los cinco pri-      al ; del while no
    cuando encuentra    meros números naturales     se llega nunca ya
    un carácter dis- */                         que el bucle no
    to de blanco en  i = 1;                     nace ninguna itera-
    vector s         while (i <= 5)             ción.
*/                     {                      */
i = 0;                   printf ("\ni = %d", i);    int condicion = 0;
while (s[i] == ' ')      i++;                       while (condicion)
   i++;                 }                             ;
```

Sentencia for

FORMA GENERAL

```
  for (expresión_1; expresión_2; expresión_3)
    sentencia
```

DESCRIPCION

En la sentencia for, a expresión_1 se le llama inicialización, a expresión_2 se le llama condición, y a expresión_3 se le llama incremento.

La forma general descrita de la sentencia for es equivalente a:

```
  expresión_1
  while (expresión_2)
    {
      sentencia
      expresión_3;
    }
```

La forma de actuar del bucle for se observa claramente en su equivalente del while.

Las tres expresiones de la sentencia for son opcionales, aunque los puntos y coma siempre deben aparecer. Si no aparece expresión_2 se asume que es 1.

EJEMPLOS

```
#include <stdio.h>
void main (void)
{
  printf ("Lista de los 100 primeros"
```

```
#include <stdio.h>
void main (void)
{
  int i, s;
```

```
          "números naturales:\n");
  int x;
  for (x = 1; x <= 100; x++)
    printf ("%d ", x);
}
```

```
  for (s = 0, i = 1; i < 100; i++)
    s += i;
  printf ("La suma de los primeros "
    "100 números naturales es %d.",s);
}
```

Nota: (s = 0, i = 1) es una expresión, más concretamente, es una expresión formada por el operador coma, cuyo operandos son expresiones de asignación.

OBSERVACION

La instrucción

```
  for (;;)
    sentencia
```

es un bucle infinito.

Sin embargo, aunque en este bucle no nos podamos salir por la condición del for, nos podemos salir por otros medio (por ejemplo, mediante la sentencia break, que estudiaremos un poco más adelante en esta misma lección).

Sentencia do

SINTAXIS

```
  do
    sentencia
  while (expresión);
```

DESCRIPCION

En la instrucción do, primero se ejecuta sentencia y a continuación se evalúa expresión. En caso de ser cierta, se ejecuta sentencia de nuevo y así sucesivamente. La iteración termina cuando la expresión se convierte en falsa.

EJEMPLO

```
/*
   imprime los 50 primeros
   números naturales
*/
#include <stdio.h>
void main (void)
{
  int i = 1;
  do
```

```
    {
      printf ("%d ", i++);
    } while (i <= 50);
}
```

OBSERVACION

Las llaves en el ejemplo anterior no son necesarias al tratarse de una sola sentencia; pero en el caso de la sentencia do, yo recomiendo que siempre pongáis las llaves para hacer el programa más legible (para el lector, no para el compilador). Obsérvese la misma instrucción sin llaves:

```
  i = 1;
  do
    printf ("%d ", i++);
  while (i <= 50);
```

Cuando un lector ve la línea del while, puede pensar que se trata de la sentencia while, cuando en realidad es la sentencia do-while. partir

Sentencia break

SINTAXIS

```
  break;
```

DESCRIPCION

Esta sentencia provoca la salida inmediata de las sentencias switch, while, for y do. Por lo tanto, su uso sólo es correcto dentro de un bloque de una estas sentencias.

EJEMPLO

```
for (i = 0; i < 10; i++)
  for (j = 0; j < i; j++)
    if (j == 5)
      break;
```

OBSERVACION

Una sentencia break obliga a una salida inmediata del ciclo (o switch) más interior.

En el ejemplo anterior, la ejecución de la sentencia break provoca una salida del bucle for de la variable j, pero no tiene ningún efecto sobre el bucle for de la variable i.

Sentencia continue

SINTAXIS

```
continue;
```

DESCRIPCION

La sentencia continue funciona de una forma algo similar a break. En vez de forzar la terminación, continue fuerza una nueva iteración del bucle y salta cualquier código que exista entre medias.

EJEMPLO

```
/*
  este bucle imprime todos los números no
  negativos del vector vector_de_numeros
*/
for (i = 0; i < 100; i++)
  {
    numero = vector_de_numeros [i];
    if (numero < 0)
      continue;
    printf ("%d ", vector_de_numeros [i]);
  }
```

OBSERVACION

En los ciclos while y do-while, una sentencia continue da lugar a que el control pase directamente a la evaluación de la condición y prosiga el proceso del bucle. En el caso de un for, primero se ejecuta la parte incremento del bucle, a continuación se ejecuta la evaluación de condición, y finalmente el bucle prosigue.

Sentencia goto

SINTAXIS

```
goto etiqueta;
```

DESCRIPCION

El lenguaje C tiene la fatalmente seductora sentencia goto. La utilización de esta sentencia en un programa es una de las mejores formas de hacerlo ilegible y difícilmente modificable.

La sentencia goto provoca un salto a una etiqueta que se encuentra en la misma función. Una etiqueta es un identificador válido de C seguido por dos puntos.

EJEMPLO

```
/*
  Programa que imprime los 100
  primeros números naturales.
*/

#include <stdio.h>
void main (void)
{
  int x = 1;
  bucle: /* etiqueta */
    printf ("%d ", x++);
    if (x <= 100)
      goto bucle; /* salto a etiqueta */
}
```

En esta lección se ha estudiado todas las sentencias de control de programa que posee el C. Entre ellas se incluyen los constructores de bucles while, for y do, las sentencias condicionales if y switch; las sentencias break, continue y goto. Aunque técnicamente también la sentencia return afecta al flujo de control de un programa, no se hará referencia a ella hasta la siguiente lección sobre funciones.

Consejo: No utilices las sentencias goto y continue a no ser que sea absolutamente necesario en un programa; lo mismo digo para la sentencia break siempre que no sea en un switch (en este caso es necesario). La utilización de estas sentencias disminuye la lectura de un programa, además de que no hacen ninguna falta ya que el C es un lenguaje muy rico en sentencias, operadores y funciones. Yo he programado bastante en C y nunca he tenido la necesidad de utilizar estas sentencias; un ejemplo es la programación de este tutor.

LECCIÓN 4

INTRODUCCION A LA LECCION 4

El objetivo de esta lección es describir algunas funciones del C.

Las funciones que detallaremos son:

- Funciones de control de programa: **exit (), _exit (), abort ()** y **assert ().**
- Funciones de E/S: **printf (), scanf (), putchar (), getchar (), puts ()** y **gets ().**
- Funciones de consola: **cprintf (), cscanf (), putch (), getch (), getche (), ungetch (), cputs (), cgets ()** y **kbhit ().**
- Funciones de consola de Turbo C: **lowvideo (), highvideo ()** y **normvideo ().**

También veremos la constante **EOF.**

En Turbo C se verá además dos constantes definidas en stdlib.h (**EXIT_SUCCESS** y **EXIT_FAILURE**) y una variable definida en conio.h (**directvideo**).

FUNCIONES DE CONTROL DE PROGRAMA

En la lección anterior hemos visto las sentencias de control de programa: **if, switch, while, for, do, break, continue** y **goto.** A continuación vamos a ver las funciones que pueden afectar al flujo de control de un programa; estas funciones son: **exit, abort** y **assert.**

FUNCIONES exit () y _exit ()

La función exit, que se encuentra declarada en la biblioteca estándar (stdlib.h), da lugar a la terminación automática de un programa.

Al describir una función siempre hay que especificar los argumentos y el valor que devuelve si los hay. La función exit no devuelve nada pero necesita un argumento que es recogido por el programa llamador (normalmente el sistema operativo). Por convención, el valor 0 indica terminación normal; valores distintos de 0 indican situaciones anormales.

En los programas que manipulan ficheros, éstos han de abrirse, manipularse y cerrarse. Los datos de estos ficheros no se manipulan directamente sino a través de unos almacenamientos temporales llamados buffers. La función exit cierra todos los ficheros abiertos, vacía todos los buffers de salida, y a continuación llama a la función _exit para terminar el programa. **La función _exit provoca la terminación inmediata del programa sin realizar el vaciado de los buffers ni cerrar los ficheros**; si se desea se la puede llamar directamente; el argumento de _exit es el mismo que para exit.

Ejemplo:

```
~#include <stdlib.h> /* para poder utilizar la función exit () */
~
~/* las funciones tarjeta_color () y jugar () han de estar definidas en
```

```
~   algún lado */
~
~void main (void)
~{
~  /* tarjeta_color () es una función que devuelve 0 (falso) si la tarjeta
~     del sistema no es color y 1 (cierto) si lo es */
~  if (tarjeta_color ())
~    exit (1); /* terminación anormal: la tarjeta no es color */
~  jugar (); /* llamada a función para jugar */
~  exit (0); /* terminación normal, esta sentencia no es necesaria */
~}
```

En la librería stdlib de Turbo C hay dos constantes definidas para pasárselas como argumento a la función exit:

EXIT_SUCCESS que tiene valor 0 y **EXIT_FAILURE** que tiene valor 1

De este modo, si nos olvidamos si el argumento 0 de exit significa terminación normal o anormal, utilizamos estas constantes que es más difícil que se nos olvide. Además hace el programa más legible.

Ejemplos de utilización:

```
exit (EXIT_SUCCESS);

exit (EXIT_FAILURE);
```

FUNCION abort ()

La función abort aborta el programa. Es muy similar a la función exit.

Se diferencia de la función exit en dos aspectos fundamentales:

1) La función abort no acepta ningún argumento. Se le llama de la siguiente forma:

```
abort ();
```

2) La función abort no vacía los buffers ni cierra ningún fichero.

El principal uso de abort es prevenir una fuga del programa cerrando los ficheros abiertos.

Esta función se encuentra declarada en el fichero: stdlib.h.

Cuando se desee terminar un programa inmediatamente es preferible intentar utilizar la función exit.

FUNCION assert ()

La función assert no devuelve nada y acepta una expresión como argumento. Esta función testea la expresión dada; si la expresión es cierta no hace nada; si la expresión es falsa escribe un mensaje en la salida de error estándar y termina la ejecución del programa.

El mensaje presentado es dependiente del compilador pero suele tener la siguiente forma:

Expresión fallida: <expresión>, archivo <archivo>, línea <num_línea>

La función assert se suele utilizar para verificar que el programa opera correctamente.

Se encuentra declarada en el fichero: assert.h.

En Turbo C, el mensaje presentado por la función assert es:
Assertion failed: <expresión>, file <fichero>, line <num_línea>
Abnormal program termination

Ejemplo:

```
#include <assert.h> /* para poder utilizar la función assert ()

void main (void)
{
  int x, y;
  x = y = 1;
  assert (x < y); /* la expresión x < y es falsa y el programa termina */
  x++; y++; /* estas dos sentencias nunca se ejecutan */
}
```

FUNCIONES DE E/S

Se llaman funciones de entrada/salida (input/output), abreviado funciones de E/S (I/O), a aquéllas que transportan datos entre el programa y la entrada y salida estándar.

La entrada estándar normalmente es el **teclado** y la salida estándar normalmente es la **consola**. Mientras no se diga lo contrario, en este tutor se considera la entrada estándar como el teclado y la salida estándar como la consola.

El final de la entrada y salida se suele marcar (en el caso de ficheros de texto ocurre siempre) con un carácter especial llamado Fin-De-Fichero y se simboliza "EOF" (End-Of-File). Hay una constante definida en el fichero stdio.h que se llama **EOF** y tiene el valor de -1. El carácter de fin de fichero se suele escribir con CONTROL-Z (código ASCII 26) en el DOS y CONTROL-D en UNIX. Algunas funciones del C (por ejemplo, scanf) devuelven el valor de EOF cuando leen el carácter de marca de fin de fichero.

En las operaciones de E/S, los datos utilizados suelen pasar por buffers. Un **buffer** es una cantidad de memoria utilizada para meter y sacar datos.

Tras estos preliminares ya estamos en condiciones de ver las principales funciones de E/S: **printf**, **scanf**, **putchar**, **getchar**, **puts** y **gets**.

FUNCIONES printf () y scanf ()

La función printf escribe datos formateados en la salida estándar.
La función scanf lee datos formateados de la entrada estándar.

El término "con formato" se refiere al hecho de que estas funciones pueden escribir y leer datos en varios formatos que están bajo su control.

Ambas funciones están declaradas en el fichero stdio.h, y tienen la forma general:

```
printf ("cadena de control", lista de argumentos);
scanf ("cadena de control", lista de argumentos);
```

La cadena de control está formada por caracteres imprimibles y códigos de formato. Debe haber tantos códigos de formato como argumentos.

Los códigos u órdenes de formato son las siguientes:

Código	Formato
%c	Simple carácter
%d	Entero decimal con signo
%i	Entero decimal con signo
%e	Punto flotante en notación científica: [-]d.ddd e [+/-]ddd
%f	Punto flotante en notación no científica: [-]dddd.ddd
%g	Usa %e o %f, el que sea más corto en longitud
%o	Entero octal sin signo
%s	Cadena de caracteres
%u	Entero decimal sin signo
%x	Entero hexadecimal sin signo
%%	Signo de tanto por ciento: %
%p	Puntero
%n	El argumento asociado debe ser un puntero a entero en el que se pone el número de caracteres impresos hasta ese momento

Las órdenes de formato pueden tener modificadores. Estos modificadores van entre el % y la letra identificativa del código. Si el modificador es un **número**, especifica la anchura mínima en la que se escribe ese argumento. Si ese número empieza por **0**, los espacios sobrantes (si los hay) de la anchura mínima se rellenan con 0. Si ese número tiene **parte real y parte fraccionaria**, indica el número de dígitos de la parte real y de la parte fraccionaria a imprimir en el caso de imprimir un número, o indica el número mínimo y máximo a imprimir en el caso de imprimir una cadena de caracteres. Por defecto, la salida se justifica a la derecha en caso de que se especifique anchura mínima; si el modificador es un **número negativo**, la justificación se hará a la izquierda. Otros dos modificadores son las letras **l** y **h**; el primero indica que se va a imprimir un long, y h indica que se va a imprimir un short. En la explicación de los modificadores se ha hablado de imprimir, es decir, hemos hablado del printf. Los modificadores de scanf son los mismos.

Después de esta parrafada veamos unos ejemplos prácticos.

Ejemplos:

Sentencia printf ()	Salida
(":%f:", 123.456)	:123.456001:
(":%e:", 123.456)	:1.234560e+02:
(":%g:", 123.456)	:123.456:
(":%-2.5f:", 123.456)	:123.45600:
(":%-5.2f:", 123.456)	:123.46:
(":%5.5f:", 123.456)	:123.45600:
(":%10s:", "hola")	: hola:
(":%-10s:", "hola")	:hola :
(":%2.3s:", "hola")	:hol:
(":%x:", 15)	:f:
(":%o:", 15)	:17:
(":%05d:", 15)	:00015:
(":abc:%n", &var_int)	:abc: (además la variable var_int toma el valor de 5)

Las órdenes de formato de Turbo C son un poco más rica que en el ANSI C:

```
Especificadores de formato
==========================

% [banderas] [width] [.prec] [F|N|h|l] type


-------------------------------------
Especificador de formato "[bandera]"
=====================================

[bandera] | Acción
=============================================================================
(ninguna) | Justificado a la derecha; rellenos con 0 o blancos a la izq.
     -    | Justificado a la izquierda; rellenos con espacios a la derecha
     +    | Siempre empieza con + o -
  blanco  | Imprime signo para valores negativos solamente
     #    | Convierte usando forma alternativa

Formas alternativas para la bandera #
=====================================
-----------------------------------------------------
c,s,d,i,u| (no tiene efecto)
o        | Los argumentos distintos de 0 empiezan con 0
x o X    | Los argumentos empiezan con 0x o 0X
e, E, f  | Siempre usa punto decimal
g o G    | Igual que e, E, o f pero con ceros al final


-------------------------------------
Especificador de formato "[anchura]"
=====================================

[anchura] | Acción
=============================================================
     n    | Anchura mínima, rellenos con blanco
    0n    | Anchura mínima, rellenos con 0 a la izquierda
     *    | El próximo argumento de la lista es la anchura


-----------------------------------
Especificador de formato "[.prec]"
===================================

  [.prec]  | Acción
=============================================================
(ninguna) | Precisión por defecto
    .0    | (d,i,o,u,x)              Precisión por defecto
          | (e,E,f)                  Sin punto decimal
    .n    | Al manos n caracteres
     *    | El próximo argumento de la lista es la precisión


-------------------------------------
Especificador de formato "[F|N|h|l]"
=====================================

Modificador | Cómo es interpretado el argumento
=================================================================
      F     | ------         el argumento es puntero far
      N     | ------         el argumento es puntero near
      h     | d,i,o,u,x,X    el argumento es short int
      l     | d,i,o,u,x,X    el argumento es long int
      l     | e,E,f,g,G      el argumento es double (sólo scanf)
      L     | e,E,f,g,G      el argumento es long double
```

```
---------------------------------
Especificador de formato "tipo"
=================================
   tipo  | Acción
=============================================================================
    d    | signed   decimal int
    i    | signed   decimal int
    o    | unsigned octal   int
    u    | unsigned decimal int
    x    | En printf = unsigned hexadecimal int;
         | en scanf = hexadecimal int
    X    | En printf = unsigned hexadecimal int;
         | en scanf = hexadecimal long
    f    | Punto flotante [-]dddd.ddd
    e    | Punto flotante [-]d.ddd e [+/-]ddd
    g    | Formato e o f basado en la precisión
    E    | Igual que e excepto E par exponente
    G    | Igual que g excepto E para exponente
    c    | Carácter simple
    s    | Imprime caracteres terminados en '\0' or [.prec]
    %    | El carácter %
    p    | Puntero: near = YYYY; far = XXXX:YYYY
    n    | Almacena número de caracteres escritos en la dirección apuntada
         | por el argumento de entrada
```

Ejemplos:

Sentencia printf ()	Salida
("%x", 2)	2
("%#x", 2)	0x2
("%#X", 2)	0X2
("%f", 1.2)	1.200000
("%g", 1.2)	1.2
("%#g", 1.2)	1.200000
("%*.*f", 5, 4, 1.2)	1.2000

Hay una diferencia muy importante entre los argumentos de printf y scanf. En printf los argumentos son expresiones pero en scanf los argumentos han de ser direcciones de memoria (punteros).

Los punteros se van a estudiar en una lección posterior. No obstante, hablaremos en este momento un poco de ellos para saber utilizar la función scanf.

Los punteros hemos dicho que son direcciones de memoria. Para obtener la dirección de memoria de una variable es necesario aplicar el operador monario & (no confundirlo con el operador binario, que es el and entre bits) de la siguiente forma:

```
& variable
```

Hay un tipo de variable un poco especial a este respecto, que son los vectores (también llamados arrays), cuyo nombre es un puntero que apunta al primer elemento del vector.

Ahora mismo se pretende que entendamos cómo usar la función scanf, no los punteros, que es tema de otra lección; por cierto, el tema de los punteros es, quizás, una de los más difíciles del C.

Ejemplo de cómo usar la función scanf:

```
scanf ("%d %s %f", &variable_entera, cadena_de_caracteres, &variable_float);
```

Programa ejemplo:

```
/*
  Programa que lee números enteros de teclado hasta que se introduce un 0.
  El programa no es muy útil pero sí instructivo.
*/

#include <stdio.h> /* para poder utilizar la función scanf */

void main (void)
{
  int x;
  do
    {
      scanf ("%d", &x);
    } while (x != 0);
}
```

Las llamadas a funciones son expresiones y como el resultado de evaluar una expresión es un valor (a no ser que el resultado de la expresión sea de tipo void), las funciones pueden devolver valores y la llamada a esa función toma el valor devuelto por la función.

Supóngamos que f() es una función que devuelve un entero, entonces, las siguientes expresiones son correctas:

```
  int x, y, z;

  x = f ();
  y = f () * 2;
  f ();
  (void) f ();
  z = f () + f ();
  if (f () < 0)
    printf ("ERROR");
```

La función printf devuelve un valor entero que contiene el número de caracteres escritos. En caso de error, devuelve EOF.

La función scanf devuelve el número de campos que han sido asignados. Si ocurre un error o se detecta el fin de fichero, se devuelve EOF.

Ejemplo:

```
  if (scanf ("%u", &variable_unsigned) == EOF)
    printf ("Error o Fin-De-Fichero al intentar leer un valor unsigned.");
```

Profundicemos un poco más en el estudio de la función scanf.

En la cadena de control de scanf se pueden distinguir tres elementos:

- **Especificadores de formato.**
- **Caracteres con espacios en blanco.**
- **Caracteres sin espacios en blanco.**

Sobre los especificadores de formato ya hemos hablado. Sin embargo, la función scanf() tiene otro especial: Un ***** situado después del % y antes del código de formato lee los datos del tipo especificado pero elimina su asignación. Así, dada la entrada

```
  2 4
```

como respuesta a la función

```
  scanf ("%*d%d", &x);
```

asigna el valor 4 a la x y no el valor 2 que es descartado.

Un espacio en blanco en la cadena de control da lugar a que scanf() salte uno o más espacios en blanco en el flujo de entrada. Un carácter blanco es un espacio, un tabulador o un carácter de nueva línea. Esencialmente, un carácter espacio en blanco en una cadena de control da lugar a que scanf() lea, pero no guarde, cualquier número (incluido 0) de espacios en blanco hasta el primer carácter no blanco.

Un carácter que no sea espacio en blanco lleva a scanf() a leer y eliminar el carácter asociado. Por ejemplo, "%d,%d" da lugar a que scanf() lea primero un entero, entonces lea y descarte la coma, y finalmente lea otro entero. Si el carácter especificado no se encuentra, scanf() termina.

FUNCIONES putchar () y getchar ()

La función **putchar** escribe un carácter en la salida estándar.
La función **getchar** escribe un carácter en la entrada estádar.

La función putchar necesita un argumento que es el carácter a escribir.
La función getchar no recibe ningún argumento.

Ambas funciones devuelven, en caso de éxito, el carácter procesado (escrito o leído), y en caso de error o fin de fichero, EOF.

Las instrucciones

```
  char ch;
  ch = getchar ();
  putchar (ch);
```

hacen lo mismo que las instrucciones

```
  char ch;
  scanf ("%c", &ch);
  printf ("%c", &ch);
```

pero para escribir y leer caracteres simples se prefiere la primera forma.
En notación C, las instrucciones del primer ejemplo se escriben:

```
  putchar (getchar ());
```

puesto que es más eficiente.

Lo mismo se puede decir de las siguientes asignaciones:

```
  x = x + 1;
  x = x + 2;
```

que aunque sean correctas, no es estilo C y además son menos eficientes que sus correspondientes en estilo C:

```
  x++; /* o ++x; */
  x += 2; /* hacer dos veces: x++; x++; ya es menos eficiente que x += 2; */
```

Veamos un programa en C:

```
/* fichero ejemplo.c */

#include <stdio.h> /* para poder utilizar: getchar (), putchar (), EOF */

void main (void)
{
  int ch;
  while ((ch = getchar ()) != EOF)
    putchar (ch);
}
```

Al ser el nombre del programa ejemplo.c, el nombre del fichero ejecutable será ejemplo.exe.

Si se ejecuta el programa de la siguiente forma:

```
  ejemplo
```

se leen caracteres de teclado y se escriben en pantalla. Se leen caracteres hasta que se encuentra el carácter de marca de fin de fichero que en el DOS se escribe con CONTROL-Z.

Si se ejecuta el programa con redirección:

```
  ejemplo < fichero_fuente > fichero_destino
```

se produce una copia del fichero_fuente al fichero_destino.

Otro ejemplo de ejecución:

```
  ejemplo >> fichero_destino
```

en este caso se leen caracteres de teclado y se añaden al fichero_destino.

Este programa no sólo es instructivo sino también útil; pero hay dos cosas que merecen una explicación: la variable ch es int no char, y la condición del while parece un poco extravagante.

Si ch se hubiese declarado como:

```
  char ch;
```

ch es unsigned y sería un error el compararlo con EOF que vale -1.

Sí se podía haber declarado ch como:

```
  signed char ch;
```

pero esto tiene el inconveniente de que ch está en el rango -128 a 127 y ch no podría tomar los valores de los caracteres ASCII entre 128 y 255.

Así que siempre que manejen caracteres con las funciones de E/S es preferible declarar ch como entero:

```
  int ch;
```

El bucle while

```
  while ((ch = getchar ()) != EOF)
    putchar (ch);
```

puede parecer un poco raro pero es la forma de leer caracteres en C hasta que se encuentra el caracter EOF. Los paréntesis son necesarios puesto que los operadores de asignación son los penúltimos con menos precedencia (el operador, es el que tiene menos preferencia) y si no se incluyeran en el while anterior, primero se ejecutaría la comparación != y el resultado de esta comparación (1 ó 0) se asignaría a la variable ch.

FUNCIONES puts () y gets ()

La función **puts** escribe una cadena de caracteres y un carácter de nueva línea al final de la cadena en la salida estándar.
La función **gets** lee una cadena de caracteres de la entrada estándar hasta que se encuentra el carácter '\n', aunque este carácter no es añadido a la cadena.

La función puts acepta como argumento una cadena (sin formato). Si tiene éxito devuelve el último carácter escrito (siempre es '\n'). En otro caso, devuelve EOF.

La llamada a función

```
puts ("Esto es un ejemplo.");
```

es equivalente a:

```
printf ("Esto es un ejemplo.\n");
```

La función gets acepta como argumento un puntero al principio de la cadena, es decir, el nombre de la variable cadena de caracteres; y devuelve dicho puntero si tiene éxito o la constante NULL si falla.

NULL es una constante definida en el fichero stdio.h que tiene valor 0. Esta constante se suele utilizar para denotar que un puntero no apunta a ningún sitio.

Las instrucciones

```
char cadena [100];
gets (cadena);
```

no son equivalentes a

```
char cadena [100];
scanf ("%s", cadena);
```

puesto que gets lee una cadena hasta que encuentre '\n' y scanf hasta que encuentre un carácter blanco (' '), un tabulador ('\t') o un carácter de nueva línea ('\n').

Con las funciones de lectura de cadenas es necesario tener una precaución muy importante: en la declaración de cadena se ha reservado memoria para 100 caracteres; por lo tanto, si la función gets o scanf leen más de 100 caracteres, los caracteres a partir del 100 se están escribiendo en memoria en posiciones no reservadas con lo cual los resultados son impredecibles: pensad que se puede estar escribiendo datos en el código del sistema operativo, del compilador de C, de programas residentes, ...

Este problema se puede solucionar con la función scanf de la siguiente forma:

```
char cadena [100];
```

```
scanf ("%100s", cadena);
```

donde los caracteres introducidos a partir del número 100 son ignorados y no se escriben en la variable cadena. Con la función gets no se puede hacer esto.
partir
FUNCIONES DE CONSOLA
Las funciones de consola que vamos a describir no están definidas en el estándar ANSI, ya que son funciones, que por su propia naturaleza, dependen del entorno fijado y en gran parte no son portables. Sin embargo, estos tipos de funciones tienen una importancia primordial a la hora de crear un software de calidad.

Todo lo que se diga sobre estas funciones es válido para los dos compiladores de C más importantes: el de Microsoft y el de Borland. Si utilizas otro compilador distinto a los anteriores, es probable que los nombres de las funciones coincidan con las que vamos a ver, en caso contrario, estas funciones tendrán otros nombres.

Las características de consola específicas de Borland sólo se estudiará si la opción turbo está on.

Todas las funciones, y en general toda la información, de consola, están recogidas en la librería **<conio.h>** (consola input/output).

En esta lección vamos a ver diez funciones de esta librería: **cprintf**, **cscanf**, **putch**, **getch**, **getche**, **ungetch**, **cputs**, **cgets** y **kbhit**.

Además veremos tres funciones más, que pertenecen a Turbo C: **lowvideo**, **higvideo** y **normvideo**.

La función cprintf es similar a la función printf. Lo mismo sucede con los pares de funciones: cscanf-scanf, cputs-puts, putch-putchar, getch-getchar, getche-getchar, cputs-puts y cgets-gets.

Las diferencias entre estos pares de funciones se muestran en la tabla:

Funciones de E/S estándar	**Funciones de E/S de consola**
- Se escribe en la salida estádar	**- Se escribe en pantalla**
- Se lee de la salida estándar	**- Se lee de teclado**
- Se escribe y lee a través de buffers	**- No utiliza buffers**
- Para pasar a la línea siguiente, es suficiente escribir el carácter de nueva línea: '\n'	**- Para pasar a la línea siguiente hay que escribir los caracteres de nueva línea y el de retorno de carro: '\n' y '\r'**

Consecuencia de estas diferencias:

1) Las funciones de escritura de conio.h siempre escriben en pantalla. Las funciones de escritura de stdio.h normalmente escriben en pantalla, pero si se redirige la salida (esto se puede hacer al ejecutar el programa mediante los símbolos >, >> o |) ya no escribe en pantalla. Lo mismo se puede decir de las funciones de lectura y el teclado.

2) En los programas interactivos no queda bien trabajar con buffers. Si has visto los ejemplos de lecciones anteriores, getchar lee un carácter, pero lo lee una vez que se ha pulsado la tecla RETURN; además lo caracteres leídos hasta pulsar la tecla RETURN se mantienen en el buffer para ser leídos posteriormente. Esto queda muy mal en los programas interactivos. Las funciones getch y getche, que las estudiaremos un poco más adelante, lee caracteres inmediatamente, es decir, tras pulsar la tecla.

3) Las funciones de escritura estándar siempre escriben en pantalla con los colores blanco sobre negro. Las funciones de escritura de consola escriben con cualquier atributo de pantalla.

FUNCIONES cprintf () y cscanf ()

Estas dos funciones son exactamente iguales que sus correspondientes printf y scanf. Sólo se diferencia en dos cosas:

1) La función cprintf escribe en pantalla en la posición actual del cursor y con el atributo de pantalla actual. La función cscanf lee de teclado.

2) En la función cprintf, para pasar a la línea siguiente, es necesario escribir dos caracteres: '\n' y '\r'.

Estas dos diferencias ocurren con todas las funciones de entrada y salida de conio.h.

Ejemplo:

```
cprintf ("Pasando al principio de la línea siguiente\n\r");
```

En la librería conio.h de Turbo C existe una variable de tipo entera que se llama **directvideo**. Esta variable controla la salida: si directvideo tiene el valor 1, la salida va directamente a RAM de vídeo; si tiene el valor 0, la salida se escribe vía llamadas a la ROM BIOS. El valor por defecto es directvideo = 1. Un valor de 1 hace la escritura más rápida pero menos portable.

Ejemplo:

```
#include <conio.h> /* directvideo: sólo Turbo C */
#include <stdio.h> /* puts () */

void main (void)
{
  directvideo = 0;
  puts ("Este mensaje se escribe vía llamadas a la ROM BIOS.");
  directvideo = 1;
  puts ("Este mensaje se escribe directamente a la RAM de vídeo.");
}
```

FUNCIONES cputs () y cgets ()

La función cputs escribe una cadena de caracteres en pantalla. Si recuerdas, las función puts escribía una cadena y además un caracter de nueva línea; la función cputs no pasa a la línea siguiente a no ser que se encuentre en la cadena los caracteres de nueva línea y retorno de carro. La función cputs, al igual que la función puts, devuelve el último caracter escrito.

La función cgets lee una cadena de caracteres de consola. Esta función es un poco diferente a la función gets. El primer elemento de la cadena debe contener la longitud máxima de la cadena a ser leída. La función cgets devuelve en el segundo elemento de la cadena el número de caracteres leídos. La cadena empieza en el tercer elemento del array. Devuelve la dirección del tercer elemento del array, que es donde empieza la cadena leída.

Veamos un ejemplo de la función cgets:

```
~#include <conio.h> /* para poder utilizar las funciones cputs (),
~                     cgets (), cprintf () y getch () */
~
~void main (void)
~{
~  char cadena[83]; /* cadena[0], cadena[1] y 80 caracteres más el
~                     carácter terminador nulo */
~
~  cadena[0] = 81; /* la función cgets almacenará en cadena, como máximo,
~                    80 caracteres más el carácter nulo */
~
~  cputs ("Escribe una cadena de caracteres:\n\r");
~  cgets (cadena); /* lee cadena de consola */
~
~  cprintf ("\n\n\rNúmero de caracteres leídos: %d", cadena[1]);
~  cprintf ("\n\rCadena leída:\n\r %s", &cadena[2]); /* &cadena[2] es la
~                      dirección del tercer elemento del array cadena */
~
~  cputs ("\n\n\rPulsa cualquier tecla para finalizar el programa.");
~  getch ();
~}
```

FUNCIONES putch (), getch (), getche () y ungetch ()

La función putch escribe un carácter en consola. Es similar a putchar.

Las funciones getch y getche leen un carácter de consola, con eco a pantalla (getche) o sin eco (getch). Son similares a getchar.

Las teclas especiales, tales como las teclas de función, están representadas por una secuencia de dos caracteres: un carácter cero seguido del código de exploración de la tecla presionada. Así, para leer un caracter especial, es necesario ejecutar dos veces la función getch (o getche).

La función ungetch devuelve un carácter al teclado. Acepta como parámetro el carácter a devolver y devuelve el propio carácter si hace la operación con éxito o EOF se ha ocurrido un error.

Si hace ungetch, la próxima llamada de getch o cualquier otra función de entrada por teclado, leerá el carácter que devolvió la función ungetch. No se puede devolver más de un carácter consecutivo.

Ejemplo:

```
#include <conio.h>

void main (void)
{
   char ch = 0;

   cprintf ("Entra una cadena: ");
   while (ch != '\r')
     {
       ch = getch ();
       putch (ch);
     }
}
```

FUNCION kbhit ()

La función kbhit devuelve un valor cierto (distinto de cero) si hay una tecla disponible y falso (0) si no la hay.

Ejemplo:

```
#include <conio.h> /* para utilizar: cprintf (), kbhit (), getch () */

void main (void)
{
  float numero = 0;

  cprintf ("Pulsa cualquier tecla para parar la escritura de números.\n\r");
  while (! kbhit ())
    cprintf ("\r%g", ++numero);
  cprintf ("\n\rTecla pulsada: %c.\n\r", getch ());

  cprintf ("\n\rPulsa cualquier tecla para finalizar.");
  getch ();
}
```

FUNCIONES lowvideo (), highvideo () y normvideo ()

Estas tres funciones pertenecen a Turbo C.

La función lowvideo hace que los caracteres que se escriban a partir de la llamada a esta función se escriban en baja intensidad.

La función highvideo hace que los caracteres que se escriban a partir de la llamada a esta función se escriban en alta intensidad.

La función normvideo hace que los caracteres que se escriban a partir de la llamada a esta función se escriban en intensidad normal.

Ejemplo:

```
#include <conio.h> /* para utilizar lowvideo(), hihgvideo() y normvideo() */

void main (void)
{
  normvideo ();
  cputs ("Texto con intensidad normal.");
  lowvideo ();
  cputs ("Texto con baja intensidad.");
  highvideo ();
  cputs ("Texto con alta intensidad.");
}
```

RESUMEN DE LO QUE HEMOS VISTO EN ESTA LECCION

exit ()
_exit ()
abort ()
assert ()
EXIT_SUCCESS
EXIT_FAILURE

EOF
printf ()

scanf ()
códigos de formato
putchar ()
getchar ()
puts ()
gets ()
cprintf ()
cscanf ()
putch ()
getch ()
getche ()
ungetch ()
cputs ()
cgets ()
kbhit ()
directvideo
lowvideo ()
highvideo ()
normvideo ()

LECCIÓN 5

INTRODUCCION A LA LECCION 5

El objetivo de esta lección es hacer un estudio completo en todo lo referente a las variables, funciones, y directivas del preprocesador del C.

Los puntos que detallaremos son:
- Declaración y definición de variables y funciones.
- Tipos de variables según el lugar de la declaración: variables locales, parámetros formales y variables globales.
- Especificadores de clase de almacenamiento: **extern**, **static**, **register** y **auto**.
- Reglas de ámbito de las funciones.
- Argumentos de las funciones en general y de la función main en particular.
- Recursividad.
- Separación de un programa en varios ficheros.
- Número variable de argumentos. Librería **<stdarg.h>**: **va_list**, **va_arg()**, **va_start()** y **va_end()**. Librería **<stdio.h>**: **vprintf()** y **vscanf()**.
- Directivas del preprocesador: **#include**, **#define**, **#undef**, **#error**, **#if**, **#else**, **#elif**, **#endif**, **#ifdef**, **#ifndef**, **#line** y **#pragma**.

FUNCIONES

Los programas en C, al menos que sean muy simples, en cuyo caso estarían compuestos sólo por la función main, están formados por varias funciones.

DECLARACION DE FUNCIONES

La forma general de una función en el estándar ANSI actual es:

```
especificador_de_tipo nombre_de_la_funcion (lista_de_declaraciones_de_param)
{
  cuerpo_de_la_funcion
}
```

y en el C de Kernighan y Ritchie es:

```
especificador_de_tipo nombre_de_la_funcion (lista_de_parametros)
declaracion_de_los_parametros
{
  cuerpo_de_la_funcion
}
```

especificador_de_tipo es el tipo del valor que devuelve la función, si es void, la función no devuelve ningún valor, si no se pone se supone que la función devuelve un entero; lista_de_declaraciones_de_param es una lista de declaraciones separadas por comas donde cada declaración consta de tipo y nombre de parámetro; cuerpo_de_la_funcion es una lista de sentencias C, incluso puede no haber ninguna: 'nada () {}' en cuyo caso la función no hace nada (será útil como un lugar vacío durante el desarrollo de un programa); lista_de_parametros es una lista de nombres de parámetros separados por comas; declaracion_de_los_parametros consiste en especificar el tipo de los parámetros.

La segunda forma aunque está permitida en el ANSI C es considerada como obsoleta. De hecho, las palabras claves void, const y volatile vistas ya, no existen en el viejo C de Kernighan y Ritchie.

Veamos un ejemplo:

```
~  #include <stdio.h>
~
~  /* Función declarada según la forma del estándar ANSI */
~  void funcion_1 (int numero_1_funcion_1, int numero_2_funcion_1)
~  {
~    printf ("\n%d + %d = %d", numero_1_funcion_1, numero_2_funcion_1,
~            numero_1_funcion_1 + numero_2_funcion_1);
~  }
~
~  /* Función declarada según la forma del C de Kernighan y Ritchie */
~  void funcion_2 (numero_1_funcion_2, numero_2_funcion_2)
~  int numero_1_funcion_2, numero_2_funcion_2;
~  {
~    printf ("\n%d - %d = %d", numero_1_funcion_2, numero_2_funcion_2,
~            numero_1_funcion_2 - numero_2_funcion_2);
~  }
~
~  void main (void)
~  {
~    int numero_1_funcion_main = 2, numero_2_funcion_main = 3;
~    funcion_1 (numero_1_funcion_main, numero_2_funcion_main);
~    funcion_1 (numero_1_funcion_main, numero_2_funcion_main);
~  }
```

La salida de este programa es:

```
2 + 3 = 5
2 - 3 = -1
```

El funcionamiento de este ejemplo parace claro. Hay una observación importante que hacer acerca del mismo: Si la función main se hubiera definido antes de las funciones funcion_1 y funcion_2, el compilador al procesar las llamadas a funcion_1 y funcion_2 no sabe de qué tipo son dichas funciones y nos puede informar de un error o asumir que devuelven un entero. Es bueno declarar todas las funciones a utilizar (excepto main) antes de su definición para evitar problemas de este tipo.

El ejemplo anterior se escribiría de la siguiente forma:

```
~  #include <stdio.h>
~
~  void funcion_1 (int numero_1_funcion_1, int numero_2_funcion_1);
~  void funcion_2 (int numero_1_funcion_2, int numero_2_funcion_2);
~
~  void main (void)
~  {
~    int numero_1_funcion_main = 2, numero_2_funcion_main = 3;
~    funcion_1 (numero_1_funcion_main, numero_2_funcion_main);
~    funcion_1 (numero_1_funcion_main, numero_2_funcion_main);
~  }
~
~  /* Función declarada según la forma del estándar ANSI */
~  void funcion_1 (int numero_1_funcion_1, int numero_2_funcion_1)
~  {
~    printf ("\n%d + %d = %d", numero_1_funcion_1, numero_2_funcion_1,
~            numero_1_funcion_1 + numero_2_funcion_1);
~  }
~
~  /* Función declarada según la forma del C de Kernighan y Ritchie */
~  void funcion_2 (numero_1_funcion_2, numero_2_funcion_2)
~  int numero_1_funcion_2, numero_2_funcion_2;
~  {
~    printf ("\n%d - %d = %d", numero_1_funcion_2, numero_2_funcion_2,
~            numero_1_funcion_2 - numero_2_funcion_2);
~  }
```

El compilador, en esta nueva versión, al procesar las llamadas a las funciones funcion_1 y funcion_2 en la función main ya sabe cómo son estas funciones.

En las declaraciones (no en las definiciones que son las descripciones completas de las funciones) es lícito suprimir el nombre de los parámetros, así que también se podría haber hecho:

```
  void funcion_1 (int, int);
  void funcion_2 (int, int);
```

e incluso también:

```
  void funcion_1 (int, int), funcion_2 (int, int);
```

al igual que se declaran las variables.

A las declaraciones de las funciones también se les llama prototipos.

VALORES DEVUELTOS

Todas las funciones, excepto aquéllas del tipo void, devuelven un valor. Este valor se especifica explícitamente en la sentencia return y si no existe ésta, el valor es 0.

La forma general de return es:

return expresión;

Tres observaciones sobre la sentencia return:

1) La sentencia return tiene dos usos importantes. Primero, fuerza a una salida inmediata de la función, esto es, no espera a que se llegue a la última sentencia de la función para acabar. Segundo, se puede utilizar para devolver un valor.

2) return no es una función sino una palabra clave del C, por lo tanto no necesita paréntesis como las funciones, aunque también es correcto:
return (expresión);
pero teniendo en cuenta que los parántesis forman parte de la expresión, no representan una llamada a una función.

3) En las funciones de tipo void se puede hacer:
return;
y de esta forma se provoca la salida inmediata de la función.

Veamos un ejemplo para aclarar todo esto:

```
~#include <stdio.h>
~
~int maximo (int, int);
~long potencia (int, int);
~
~void main (void)
~{
~  int a = 2, b = 3, c = 4, d = 5;
~  printf ("\nEl máximo entre %d y %d es %d.", a, b, maximo (a, b));
~  printf ("\n%d elevado a %d es %d.\n", c, d, potencia (c, d));
~}
~
~int maximo (int ma, int mb)
~{
~  return ma >= mb ? ma : mb;
~}
~
~long potencia (int pa, int pb)
~{
~  int i;
~  long pot = 1;
~  for (i = 1; i <= pb; i++)
~    pot *= pa;
~  return pot;
~}
```

La salida de este programa es:

El máximo de 2 y 3 es 3.
4 elevado a 5 es 1024.

TIPOS DE VARIABLES SEGUN EL LUGAR DE DECLARACION

Existen tres lugares básicos donde se pueden declarar variables: dentro

de funciones, en la definición de parámetros de funciones y fuera de las funciones. Estas variables son, respectivamente, **variables locales**, **parámetros formales** y **variables globales**.

VARIABLES LOCALES

Son aquéllas que se declaran dentro de una función.

Estas variables se declaran al principio de un bloque de código, se destruyen al llegar al final del bloque de código y sólo puede ser utilizada (ya que sólo tiene existencia) dentro de ese bloque. Recuerda que un bloque de código empieza con una llave abierta y termina con una llave cerrada.

Ejemplos:

```
~void f1 (void)
~{
~  int x;   /* se reserva memoria para x */
~  x = 10;  /* se le asigna un valor */
~}          /* se libera la memoria asignada a x */

~void f2 (void)
~{
~  int x = 1;           /* se reserva memoria para x */
~  /* sentencias */
~  {
~    int y = 2;         /* se reserva memoria para y */
~    /* sentencias */
~  }                    /* se libera memoria asignada a y */
~  /* sentencias */     /* en ese punto no existe la variable y */
~}                      /* se libera memoria asignada a x */

~void f3 (void)
~{
~  int x, y; /* se reserva memoria para las variables x e y */
~  x = 1;    /* se asigna 1 a la variable x */
~  y = x;    /* se asigna x (1) a la variable y */
~  {         /* comienzo de un bloque de código */
~    int x;  /* se reserva memoria para x; ésta es distinta a la anterior */
~    x = 2;  /* se asigna 2 a la variable x de este bloque */
~    y = x;  /* se asigna x (2) a la variable y */
~  }         /* se libera memoria asignada a la variable x de este bloque */
~  y = x;    /* se asigna x (1) a la variable y */
~}           /* se libera memoria asignada a las variables x e y */
```

PARAMETROS FORMALES

Si una función va a usar argumentos, entonces debe declarar las variables que van a aceptar los valores de esos argumentos. Estas variables son los parámetros formales de la función. Se comportan como cualquier otra variable local de la función.

Ejemplos:

```
~void f1 (int x)
~{
~  /* x es una variable local a esta función */
~}
```

```
~void f2 (int x)
~{
~  int x; /* ERROR: se ha intentado definir dos variables del mismo
~            nombre en el mismo ámbito */
~}
```

VARIABLES GLOBALES

A diferencia de las variables locales, las variables globales se conocen a lo largo de todo el programa y se pueden usar en cualquier parte de código. Además, mantienen su valor durante toda la ejecución del programa. Las variables globales se crean al declararlas en cualquier parte fuera de una función.

Ejemplo:

```
~void f1 (void), f2 (void), f3 (void);
~
~int x;
~
~void main (void)
~{
~  f1 ();
~  f2 ();
~  f3 ();
~}
~
~void f1 (void)
~{
~  x = 10;
~}
~
~void f2 (void)
~{
~  x = 11;
~}
~
~void f3 (void)
~{
~  int x; /* esta variable x es local, es distinta a la global */
~  x = 12; /* se le asinga x a la variable x local, no a la global */
~}
```

ESPECIFICADORES DE CLASE DE ALMACENAMIENTO

Existen cuatro especificadores de clase de almacenamiento soportados por C. Son: **extern**, **static**, **register** y **auto**.

Se usan para indicar al compilador cómo se debe almacenar la variable que le sigue. El especificador de almacenamiento precede al resto de la declaración de variable. Su forma general es:

especificador_de_almacenamiento tipo nombre_de_variable;

EXTERN

Consideremos los dos siguientes ejemplos:

```
int a;        /* definición de una variable  */
extern int b; /* declaración de una variable */
```

En la primera sentencia estamos definiendo una variable de tipo entero llamada a.
En la segunda sentencia estamos declarando una variable de tipo entero llamada b.

En la definición de una variable se reserva un espacio de memoria para una variable.
La declaración le indica al compilador que esa variable está o será definida en otra parte, pero no reserva memoria para ella.

Así pues, una variable sólo se puede definir una vez pero se puede declarar todas las veces que se desee.

A partir de ahora, cuando se diga declarar nos estaremos refiriendo a la declaración y a la definición, a no ser que se distinga explícitamente entre ambos conceptos, ya que es el término más utilizado en todos los libros y programas.

El principal uso de extern se da cuando un programa está compuesto de varios ficheros y tenemos una variable global a varios de ellos. Obsérvese el siguiente ejemplo:

```
/* Fichero 1 */          /* Fichero 2 */
int x;                   extern int x;
main ()                  f2 ()
{                        {
  ...                      ...
}                        }
f1 ()                    f3 ()
{                        {
  ...                    ...
}                        }
```

En la situación anterior, a la variable x pueden acceder las cuatro funciones, es decir, los dos ficheros.
Si no hubiésemos hecho
```
  extern int x;
```
las funciones del fichero 2 no podrían acceder a la variable x.
Y si hubiésemos puesto
```
  int x;
```
en vez de
```
  extern int x;
```
en el fichero 2, entonces, el compilador daría un error porque se está intentando definir dos veces la misma variable en el mismo ámbito.

STATIC

Las variables globales son variables permanentes.
Tienen dos significados diferentes dependiendo si son locales o globales.

VARIABLES ESTATICAS LOCALES

La diferencia con las variables locales normales es que su contenido no se pierde al salirse de la función, de tal manera que al volver a entrar en la función, la variable estática tiene el mismo valor que el que tenía cuando terminó la función en la llamada anterior. La variable estática sólo es inicializada en la primera llamada a la función.

Ejemplo:

```
#include <stdio.h>

void f1 (void), f2 (void);

void main (void)
{
  f1 ();
  f1 ();
  f2 ();
  f2 ();
}

void f1 (void)
{
  static int x = 1;
  printf ("\nx = %d", x);
  x++;
}

void f2 (void)
{
  int y = 1;
  printf ("\ny = %d", y);
  y++;
}
```

La salida de este programa es:

```
x = 1
x = 2
y = 1
y = 1
```

VARIABLES ESTATICAS GLOBALES

Cuando se aplica el modificador static a una variable global, se indica al compilador que cree una variable global conocida únicamente en el fichero en el que se declara la variable global static. Esto significa que, aunque la variable es global, las rutinas de otros ficheros no la reconocerán ni alternarán su contenido directamente; así, no estará sujeta a efectos secundarios.

REGISTER

El especificador register pide al compilador de C que mantenga el valor de las variables definidas con ese modificador en un registro de la CPU en lugar de en memoria, que es donde se almacenan normalmente las variables.

Varias observaciones:

1) El acceso a los registros de la CPU es mucho más rápido que el acceso a la memoria.

2) Las variables register se almacenan en los registros si se puede, si no, se almacenan en memoria.

3) Las variables register sólo pueden ser de tipo int y char, y además han de ser locales no estáticas o parámetros de función.

4) En la mayoría de los sistemas sólo se permiten una o dos variables register al mismo tiempo. En la práctica, se declaran variables register aquéllas que se utilizan como índices en los bucles.

Ejemplo:

```
/* esta función calcula la potencia
   de dos números enteros */
int pot_ent (int base, int exponente)
{
  register temporal = 1;
  for (; exponente; exponente--)
    temporal *= base;
  return temporal;
}
```

AUTO

Las variables auto (automáticas) son todas aquellas variables locales que no son estáticas.

En la práctica, este especificador de clase de almacenamiento no se utiliza nunca, ya que todas las variables locales que no llevan el especificador static son consideradas auto.

REGLAS DE AMBITO DE LAS FUNCIONES

Las reglas de ámbito de un lenguaje son las reglas que controlan si un fragmento de código conoce o tiene acceso a otro fragmento de código o de datos.

Si queremos que una función f() pueda ser llamada desde dos ficheros distintos, hacemos lo siguiente:

```
/* Fichero 1 */

void f (void);

void main (void)
{
  ...
}

void f (void)
{
```

```
/* Fichero 2 */

void f (void);

void f2 (void)
{
  ...

  f ();

  ...
```

```
    ...                    }
}
```

Como se observa con las declaraciones (al principio de los dos ficheros) y las definiciones (la función f() se define al final de primer fichero) ocurre lo mismo que con las declaraciones y definiciones de las variables globales.

partir

ARGUMENTOS DE LAS FUNCIONES

Si una función va a usar argumentos, debe declarar variables que tomen los valores de las argumentos. Como se dijo antes, estas variables se llaman parámetros formales de la función. Se comportan como otras variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir.

Los argumentos se pueden pasar a las funciones de dos formas:

- Llamada por valor: este método copia el valor del argumento en el parámetro formal.

- LLamada por referencia: este método copia la dirección del argumento (que ha de ser una variable) en el parámetro formal.

Los parámetros de la función printf son pasados por valor:

```
printf ("%d", x); /* pasamos el valor de x */
```

Los parámetros de la función scanf son pasados por referencia:

```
scanf ("%d", &x); /* pasamos la dirección de x */
```

Ejemplo de programa con función con llamadas por referencia:

```
#include <stdio.h>

void intercambiar (int *px, int *py);

void main (void)
{
  int x = 2, y = 3;
  printf ("\nx = %d, y = %d", x, y);
  intercambiar (&x, &y);
  printf ("\nx = %d, y = %d", x, y);
}

void intercambiar (int *px, int *py)
{
  int temporal;
  temporal = *px;
  *px = *py;
  *py = temporal;
}
```

Comentario de este programa:

En lecciones posteriores se van a estudiar los punteros (variables que contienen direcciones de memoria) en profundidad, pero inevitablemente ya nos hemos vistos obligados a hablar algo de ellos en lecciones anteriores, y otra vez vamos a estar obligados a hablar un poco más de

ellos en este momento si queremos completar nuestro estudio sobre las funciones.

La forma general de declarar un puntero es:

```
tipo *nombre_puntero;
```

Una variable puntero es una variable que contiene una dirección de memoria. Al valor de la dirección de memoria apuntada por un puntero se accede de la siguiente manera:

```
*nombre_puntero
```

La dirección de una variable, como se dijo en lecciones anteriores, se obtiene de la siguiente forma:

```
&nombre_variable
```

En este caso, * es el operado de contenido y & es el operador de dirección. Son monarios. No confundir con los operadores binarios de multiplicación (*) y de and a nivel de bits (&).

Después de lo dicho, estamos en condiciones de comprender el programa anterior.

Al hacer 'intercambiar (&x, &y);' estamos pasando a la función intercambiar las direcciones de las variables x e y.

Al hacer la declaración 'void intercambiar (int *px, int *py)' estamos declarando dos variables locales px e py que son punteros a enteros, es decir, que contienen direcciones en las cuales hay valores enteros.

Al hacer '*px' y '*py' estamos accediendo a los valores de tipo entero apuntados por las punteros px y py, es decir, estamos accediendo a los valores de las variables x e y de la función main.

Hay un caso especial de llamada por referencia que es el paso de arrays como argumento de una función. También este caso se discutió cuando se describió la función scanf en la lección anterior. Y también, al igual que los punteros, se desarrollará los arrays en lecciones posteriores (concretamente en la lección siguiente). Pero, no obstante, hay que hacer algunas observaciones en este momento para explicar cómo se pueden pasar como argumento a una función.

Recordemos que el nombre de la variable array es un puntero al primer elemento del array. De esta forma cuando se usa un array como un argumento a una función, sólo se pasa la dirección del array, no una copia del array entero. Cuando se llama a una función con un nombre de array, se pasa a la función un puntero al primer elemento del array.

Existen tres formas de declarar un parámetro que va a recibir un puntero a un array. Veamos con un ejemplo las tres formas.

```
#include <stdio.h>

void funcion_ejemplo_1 (int a[10]);
void funcion_ejemplo_2 (int a[]);
void funcion_ejemplo_3 (int *a);

void main (void)
{
  int array [10];
  register int i;
  for (i = 0; i < 10; i++)
```

```
    array[i] = i;
  funcion_ejemplo_1 (array);
  funcion_ejemplo_2 (array);
  funcion_ejemplo_3 (array);
}

void funcion_ejemplo_1 (int a[10])
{
  register int i;
  for (i = 0; i < 10; i++)
    printf ("%d ", a[i]);
}

void funcion_ejemplo_2 (int a[])
{
  register int i;
  for (i = 0; i < 10; i++)
    printf ("%d ", a[i]);
}

void funcion_ejemplo_3 (int *a)
{
  register int i;
  for (i = 0; i < 10; i++)
    printf ("%d ", a[i]);
}
```

En la función funcion_ejemplo_1(), el parámetro a se declara como un array de enteros de diez elementos, el compilador de C automáticamente lo convierte a un puntero a entero. Esto es necesario porque ningún parámetro puede recibir un array de enteros; de esta manera sólo se pasa un puntero a un array. Así, debe haber en las funciones un parámetro de tipo puntero para recibirlo.

En la función funcion_ejemplo_2(), el parámetro a se declara como un array de enteros de tamaño desconocido. Ya que el C no comprueba los límites de los arrays, el tamaño real del array es irrelevante al parámetro (pero no al programa, por supuesto). Además, este método de declaración define a como un puntero a entero.

En la función funcion_ejemplo_3(), el parámetro a se declara como un puntero a entero. Esta es la forma más común en los programas escritos profesionalmente en C. Esto se permite porque cualquier puntero se puede indexar usando [] como si fuese un array. (En realidad, los arrays y los punteros están muy relacionados).

Los tres métodos de declarar un parámetro de tipo array llevan al mismo resultado: un puntero.

Otro ejemplo:

```
#include <stdio.h>

void funcion_ejemplo (int numero);

void main (void)
{
  int array [10];
  register int i;
  for (i = 0; i < 10; i++)
    funcion_ejemplo (array[i]);
}

void funcion_ejemplo (int numero)
```

```
{
  printf ("%d ", numero);
}
```

ARGUMENTOS DE MAIN

Algunas veces es útil pasar información al programa cuando se va a ejecutar. Los argumentos en la línea de órdenes son las informaciones que siguen al nombre del programa en la línea de órdenes del sistema operativo.

En este caso, la función **main** se declara para recibir dos parámetros especiales, **argc** y **argv**, que se utilizan para recibir los argumentos de la línea de órdenes.

El parámetro argc contiene el número de argumentos de la línea de órdenes y es un entero. Siempre vale 1, por lo menos, ya que el nombre del programa cuenta como el primer agumento.

El parámetro argv es un array donde cada elemento es una cadena de caracteres, la cual contiene la información suministrada al programa a través de la línea de órdenes del sistema operativo. Dicho de otro modo, un array donde cada elemento es un puntero al primer elemento de la cadena correspondiente.

Ejemplo:

```
/*
   Este programa acepta un nombre en la línea de
   órdenes tras el nombre del programa e imprime
   un mensaje de salutación
*/

#include <stdio.h>

void main (int argc, char *argv[])
{
  if (argc != 2)
    printf ("El número de argumentos es incorrecto.\n");
  else
    printf ("Hola %s.\n", argv[1]);
}
```

En Turbo C, también podemos utilizar un tercer parámetro en la función main, **arge**, con la misma estructura que el parámetro argv, y que contiene las cadenas del entorno.

El último elemento del vector arge (al igual que en argv) contiene el valor NULL; esto indica que el puntero del último elemento de arge no apunta a ningún sitio. Recordamos que NULL es una constante definida en las librerías <stdio.h> y <stdlib.h> entre otras, y equivale al valor 0.

Los nombres argc, argv y arge son una convención ya que pueden tener cualquier nombre válido de un identificador en C.

Ejemplo:

```
/*
  Este programa acepta una serie de argumentos
```

```
    en la línea de órdenes del sistema operativo
    y los imprime
  */

  #include <stdio.h> /* printf (), NULL */

  void main (int argc, char *argv[], char *arge[])
  {
    register int i;
    printf ("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
      printf ("argv[%d] = \"%s\"\n", i, argv[i]);
    for (i = 0; arge[i] != NULL; i++)
      printf ("arge[%d] = \"%s\"\n", i, arge[i]);
  }
```

RECURSIVIDAD

Las funciones en C se pueden llamar a sí mismas. Si una expresión en el cuerpo de una función se llama a sí misma, la función es recursiva.

Vamos a escribir dos versiones del cálculo del factorial de un número:

```
unsigned long fact1 (int n)
{
  register int i;
  unsigned long factorial = 1;
  for (i = 1; i <= n; i++)
    factorial *= i;
  return (factorial);
}
```

```
unsigned long fact2 (int n)
{
  unsigned long factorial;
  if (n <= 1)
    factorial = 1;
  else
    factorial = n * fact2 (n-1);
  return (factorial);
}
```

La función fact1 es la versión iterativa y fact2 es la versión recursiva del cálculo del factorial de un número.

La versión iterativa tiene dos ventajas sobre la versión recursiva:

1. Es más rápida.
2. Consume menos memoria.

La versión recursiva es preferible en aquellos casos en los que la lógica es mucho más clara y sencilla que en la versión iterativa.

En el ejemplo anterior es preferible la versión iterativa a la recursiva ya que la dificultad en la lógica es similar en ambos casos.

Cuando se escriben funciones recursivas se debe tener una expresión if en algún sitio que fuerce a la función a volver sin que se ejecute la llamada recursiva. Si no se hace así, la función nunca devolverá el control una vez que se le ha llamado; esto produce un desbordamiento de la pila.

SEPARACION DE UN PROGRAMA EN VARIOS FICHEROS

En programas grandes es conveniente dividir el código fuente en varios ficheros y compilarlos por separados.

Esto tiene dos grandes ventajas:

1. Mayor modularidad.
2. Tiempos de compilación más cortos, ya que cuando modificamos un fichero con código fuente, sólo tenemos que compilar ese fichero y enlazar el programa.

La forma de realizar la compilación separada depende del compilador.

En Turbo C, la compilación separada se consigue creando un fichero proyecto (con extensión .PRJ) en el cual se listan todos los ficheros fuentes que componen el proyecto. En todas las versiones hay una opción para manejar los proyectos: abrirlos, cerrarlos, obtener ayuda acerca de ellos, etcétera.

NUMERO VARIABLE DE ARGUMENTOS

En la librería **<stdarg.h>** nos encontramos toda la información necesaria para declarar funciones en las que el número y el tipo de parámetros es variable.

Vamos a mostrar inmediatamente un ejemplo y vamos a basar la explicación de este apartado sobre el ejemplo.

```
#include <stdio.h>  /* vprintf () */
#include <stdarg.h> /* va_list, va_start (), va_end () */

void imprimir_cadena_con_formato (char *cadena_de_formato, ...);

void main (void)
{
  imprimir_cadena_con_formato ("%d %f %s", 5, 2.3, "cadena");
}

void imprimir_cadena_con_formato (char *cadena_de_formato, ...)
{
  va_list puntero_a_los_argumentos;
  va_start (puntero_a_los_argumentos, cadena_de_formato);
  vprintf (cadena_de_formato, puntero_a_los_argumentos);
  va_end (puntero_a_los_argumentos);
}
```

Si nos fijamos en la declaración de la función

```
void imprimir_cadena_con_formato (char *cadena_de_formato, ...)
```

observamos que el primer argumento es una cadena (un puntero al primer elemento de la cadena, es decir, un puntero al primer carácter) y a continuación tenemos puntos suspensivos (...) que simbolizan que el número y tipo de argumentos después del primero es variable.

En la primera línea de la función

```
va_list puntero_a_los_argumentos;
```

nos encontramos la definición de una variable (puntero_a_los_argumentos) que es del tipo va_list, es decir, esta variable es un puntero a uno de los argumentos. El tipo va_list es un tipo definido en la librería <stdarg.h>. En otras lecciones veremos cómo podemos definir nuevos tipos a partir de los tipos básicos del lenguaje.

La siguiente línea de la función en cuestión

```
va_start (puntero_a_los_argumentos, cadena_de_formato);
```

es una llamada a la función va_start(). Esta función está declarada en la librería <stdarg.h> y acepta dos parámetros. El primero es de tipo va_list, es decir, es el puntero a los argumentos, y el segundo parámetro ha de ser el parámetro conocido situado más a la derecha en la función en que estamos, que en este caso es la cadena de formato. La misión de la función va_start() es inicializar el puntero a los argumentos.

A continuación nos encontramos la sentencia

```
vprintf (cadena_de_formato, puntero_a_los_argumentos);
```

que es una llamada a la función vprintf(). La función vprintf() hace lo mismo que la función printf() con la diferencia que está basada en una lista de argumentos variables. Lo mismo ocurre con las funciones vscanf() y scanf().

La última línea de la función en estudio

```
va_end (puntero_a_los_argumentos);
```

es una llamada a la función va_end(). Esta función está declarada en la librería <stdarg.h> y acepta un parámetro que es de tipo va_list y debe ser el puntero a los argumentos. La misión de la función va_end() es finalizar el acceso del puntero a los argumentos. Es importante llamar a esta función cuando ya no es necesario utilizar el puntero a lo argumentos.

En la librería <stdarg.h> tenemos sólamente cuatro elementos: un tipo (va_list) y tres funciones (va_start(), va_end() y va_arg()). La función va_arg() que es la que nos queda por explicar la vamos a ver, al igual que las otras, con un ejemplo.

```
/* Ejemplo de argumento de longitud variable: suma de una serie */


#include <stdio.h>  /* printf () */
#include <stdarg.h> /* va_list, va_start (), va_arg (), va_end () */


void main (void)
{
  double d, suma_serie ();

  d = suma_serie (5, 0.5, 0.25, 0.125, 0.0625, 0.03125);

  printf ("La suma de la serie es %f\n", d);
}
```

```
double suma_serie (int num)
{
  double suma = 0.0, t;
  va_list ptrarg;

  va_start (ptrarg, num);

  for (; num; num--)
    {
      t = va_arg (ptrarg, double);
      suma += t;
    }

  va_end (ptrarg);
  return suma;
}
```

La novedad en este programa es la aparición de la función va_arg(). Esta función está declarada en la librería <stdarg.h>. Acepta dos parámetros: el primero ha de ser el puntero a los argumentos y el segundo es el parámetro conocido más a la derecha en la función en la que estamos. La función va_arg() devuelve el argumento al cual el puntero a los argumentos apunta actualmente.

EL PREPROCESADOR C

El preprocesador le da ciertas órdenes al compilador. Conceptualmente, el preprocesador procesa el código fuente antes que el compilador. A las instrucciones del preprocesador se les llama directivas de preprocesamiento. Todas empiezan con el símbolo #. Las directivas del ANSI C son: **#include, #define,#undef,#error,#if,#else,#elif,#endif,#ifdef,#ifndef,#line y #pragma.**

DIRECTIVA #include

Cuando el preprocesador encuentra un comando #include busca el fichero que atiende por el nombre que está situado detrás y lo incluye en el fichero actual.

El nombre del fichero puede venir de dos formas:

```
#include <stdio.h>
#include "mifichero.h"
```

El primer #include busca el fichero stdio.h en los directorios del sistema. El segundo #include busca el fichero en el directorio en el que se está trabajando. Si se incluye path, lo busca en el directorio especificado:

```
#include "c:\programa\prog.h"
```

Ejemplo:

```
/* Fichero p.h */

void f (void);


/* Fichero p.c */
```

```
#include "p.h"

void main (void)
{
  f ();
}

void f (void)
{
}
```

Conceptualmente, una vez que este programa (compuesto de dos ficheros) ha sido procesado por el preprocesador, el compilador encuentra un solo fichero:

```
/* Fichero p.c */

/* Fichero p.h */

void f (void);

void main (void)
{
  f ();
}

void f (void)
{
}
```

DIRECTIVA #define

La directiva #define sustituye en el código fuente un determinado identificador por la cadena especificada.

Tiene dos formas. La primera de ellas es:

```
#define identificador cadena
```

A este identificador se le llama nombre de macro o simplemente macro.

Esta directiva hace que a partir de la definición de la macro, cada vez que el preprocesador encuentre identificador en el código fuente, lo sustituya por cadena.

Las directivas no son sentencias C, por eso no llevan puntos y coma al final.

La cadena abarca desde el primer carácter no blanco después de identificador hasta el final de la línea.

Ejemplo:

```
#include <stdio.h>

#define BOOLEAN short
#define TRUE    1
#define FALSE   0

void main (void)
```

```
{
  BOOLEAN v = TRUE;
  printf ("%d %d", v, FALSE);
}
```

Cuando el preprocesador procesa este programa, el compilador se encuentra:

```
#include <stdio.h>

void main (void)
{
  short v = 1;
  printf ("%d %d", v, 0);
}
```

En realidad, en este ejemplo, el procesador también sustituiría la línea del #include por el contenido del fichero stdio.h, pero lo hemos obviado porque en este momento estamos explicando la directiva #define y además el contenido del fichero stdio.h depende de cada implementación de compilador de C.

Por convención y para distinguir un lector las macros de los demás identificadores del C, éstas se suelen escribir en mayúsculas.

Las macros pueden ser utilizadas como parte de la definición de otras macros. Por ejemplo:

```
#define UNO  1
#define DOS  UNO + UNO
#define TRES UNO + DOS
```

Es importante comprender que la sustitución es simplemente el reemplazamiento de un identificador por su cadena asociada. Así se puede hacer:

```
#define ERROR "Se ha producido un error"
printf (ERROR);
```

RECOMENDACION IMPORTANTE: siempre que se pueda es conveniente encerrar la definición de la macro entre paréntesis.

Obsérvese en el siguiente ejemplo lo que puede ocurrir de no hacerse así:

```
#define CUATRO 2+2
/* ... */
x = CUATRO * CUATRO;
```

El preprocesador expande la asignación de x a:

```
x = 2+2 * 2+2;
```

Si se hubiese definido la macro CUATRO de la forma:

```
#define CUATRO (2+2)
```

la expansión de la asignación de x es:

```
x = (2+2) * (2+2);
```

Como se ve, en el primer caso se asigna el valor 8 a la variable x, mientras que en el segundo se asigna el valor 16 a la variable x que es lo que realmente se desea.

Si el identificador aparece dentro de una cadena no se lleva a cabo sustituciones. Por ejemplo:

```
#define XYZ esto es una prueba
/* ... */
printf ("XYZ");
```

El código anterior no imprime esto es una prueba ; imprime XYZ .

Cuando la cadena es más larga de una línea, se puede continuar en la siguiente poniendo una barra invertida al final de la línea. Por ejemplo:

```
#define MACRO \
  if (x < y)  \
    f1 ();    \
  else        \
    f2 ();
```

La segunda forma de la directiva #define es:

```
#define id1(id2,...) cadena
```

El paréntesis ha de ir inmediatamente después de id1 ya que si no pertenecería a la cadena.

Todas las instancias de id2 en cadena serán reemplazadas con el texto actual definido por id2 cuando id1 es referenciado en el código fuente.

Ejemplo:

```
#define SUMA(a,b)  a + b
/* ... */
x = SUMA (5, 8);
```

Esto expande a:

```
x = 5 + 8;
```

Observar que en el #define es obligatorio que el paréntesis vaya junto a SUMA; sin embargo, cuando se llama a la macro no es obligatorio que el paréntesis abierto vaya junto a SUMA.

RECOMENDACION IMPORTANTE: se recomienda que todos los argumentos de la macro en la cadena vayan entre paréntesis para prevenir casos como el siguiente:

```
#define PRODUCTO(a,b) (a * b)
/* ... */
x = PRODUCTO (2+3, 4);
```

Esto expande la asignación de la x a:

```
x = (2+3 * 4);
```

Si se hubiese definido la macro de la forma:

```
#define PRODUCTO(a,b)  ((a) * (b))
```

entonces la asignación de la x se expandiría a:

```
x = ((2+3) * (4));
```

Como se ve, en el primer caso se asigna el valor 14 a la variable x, mientras que en el segundo se asigna el valor 20 a la variable x que es lo que realmente se desea.

Otros ejemplos de macros:

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define ESTA_ENTRE(x,x1,x2) ((x) >= (x1) && (x) <= (x2))
```

A las macros sin argumentos también se les llama constantes simbólicas e incluso, en algunos sitios, simplemente constantes. Constantes simbólicas ya vistas son EOF y NULL. Las funciones va_arg(), va_start() y va_end() vistas anteriormente, son macros en algunas implementaciones, pero esto es transparente para el usuario.

Ejemplo:

```
/* Fichero prog_ej.h */
#include <stdio.h>
#define funcion_principal main
#define principio {
#define final }
#define numero_entero int
#define escribir_cadena(cadena) printf (cadena)
#define escribir_numero_entero(numero) printf ("%d", numero);
#define leer_numero_entero(numero) scanf ("%d", νmero);

/* Fichero prog_ej.c */
#include "prog_ej.h"
void funcion_principal (void)
principio
  numero_entero x;
  escribir_cadena ("Introduce un número: ");
  leer_numero_entero (x);
  escribir_cadena ("x = ");
  escribir_numero_entero (x);
final
```

El uso de macros con argumentos en lugar de funciones reales tiene principalmente una ventaja: las sustituciones incrementan la velocidad del código porque no se gasta tiempo en llamar a la función. Sin embargo, hay que pagar un precio por el aumento de velocidad: el tamaño del programa aumenta debido a la duplicación del código.

DIRECTIVA #undef

La directiva #undef elimina una definición anterior de la macro que le sigue.

La forma general es:

```
#undef identificador
```

Ejemplo:

```
#define TAMANIO_ARRAY 100
int array [TAMANIO_ARRAY];
#undef TAMANIO_ARRAY
/* aquí no existe la macro TAMANIO_ARRAY */
```

DIRECTIVA #error

La directiva del preprocesador #error fuerza al compilador a parar la compilación y escribir un mensaje de error.

Tiene la forma general:

```
#error mensaje
```

Ver un ejemplo de esta directiva al final de la descripción de las directivas #if, #else, #elif y #endif.

DIRECTIVAS #if, #else, #elif y #endif

Estas directivas permiten compilar selectivamente parte del código fuente de un programa. Este proceso se llama compilación condicional.

La forma general del bloque #if es:

```
#if expresion_constante
  secuencia de sentencias
#endif
```

y la forma del bloque #if-#else es:

```
#if expresion_constante
  secuencia de sentencias
#else
  secuencia de sentencias
#endif
```

La expresión que sigue a #if se evalúa en tiempo de compilación, por eso ha de ser una expresión constante (que no contiene variables). Si la evaluación de la expresión es cierta, es decir, es 1, entonces se compilan la secuencia de sentencias que le siguen; en caso contrario, se compila la secuencia de sentencias siguientes al #else si existe, o las sentencias siguientes al #endif si no hay #else.

Ejemplo del bloque #if:

```
#if SISTEMA == IBM
  #include "ibm.h"
#endif
```

Ejemplo del bloque #if-#else:

```
#include <stdio.h>
#define MAX 10
main ()
{
  #if MAX > 100
    puts ("Compilación 1");
  #else
    puts ("Compilación 2");
  #endif
}
```

La directiva #elif quiere decir else if y establece una escala del tipo if-else-if para opciones de compilación múltiples. La directiva #elif, al igual que la #if, va seguida de una expresión constante. Si la expresión es cierta, ese bloque de código se compila y no se comprueba ninguna expresión #elif más. En otro caso, se comprueba el bloque siguiente de la serie.

La forma general de una secuencia #elif es:

```
#if expresión
  secuencia de sentencias
#elif expresión 1
  secuencia de sentencias 1
#elif expresión 2
  secuencia de sentencias 2
#elif expresión 3
  secuencia de sentencias 3
.
.
.
#elif expresión N
  secuencia de sentencias N
#endif
```

Las directivas #if y #elif se pueden anidar.

Ejemplo:

```
#if X > Y
  #if X > Z
    int xyz = 1;
  #elif
    int xyz = 2;
  #endif
#else
  int xyz = 3;
#endif
```

La expresión constante puede contener también el operador sizeof.

En Turbo C, existe un operador, **defined**, que puede ser utilizada en las expresiones de estas directivas.

La forma general es:

```
defined identificador
o
defined (identificador)
```

La expresión formada por este operador es 1 si identificador ha sido previamente definido (usando #define) y no ha sido después indefinido (usando #undef). En otro caso, este operador devuelve 0.

Este operador sólo puede ser utilizado en las directivas #if y #elif.

Ejemplo 1:

```
#if defined(X)
  v = 1;
#elif defined(Y)
  v = 2;
#else
  #error Ni X ni Y han sido definidos.
#endif
```

Ejemplo 2:

```
#if !defined(MODELO)
  #error Modelo no definido
#endif
```

DIRECTIVAS #ifdef y #ifndef

Estas directivas, al igual que las #if y #elif, son directivas de compilación condicional.

La forma general de #ifdef es:

```
#ifdef identificador
  secuencia de sentencias
#endif
```

#ifdef significa si definido . Si se ha definido previamente identificador en una sentencia #define, se compila el bloque de código que sigue a la sentencia.

Ejemplo:

```
#ifdef DEBUG
  printf ("Espacio total %d\n", espacio);
#endif
```

La forma general es #ifndef es:

```
#ifndef identificador
  secuencia de sentencias
#endif
```

#ifndef significa si no definido . Si no se ha definido previamente identificador mediante una sentencia #define, se compila el bloque de código.

Ejemplo:

```
#ifndef ABC
  printf ("ABC no definido");
#endif
```

DIRECTIVA #line

La directiva #line hace que el compilador crea que el número de línea de la próxima línea a la que está esta directiva, sea la dada por <constante>, y el fichero de entrada corriente sea el dado por <identificador>.

La forma general es:

```
#line <constante> [<identificador>]
```

Si <identificador> no es un nombre correcto de fichero, el nombre del fichero corriente no es cambiado.

Ejemplo:

```
#line 100                       /* reinicializa el contador de líneas */
main ()                         /* línea 100 */
{                               /* línea 101 */
  printf ("%d\n", __LINEA__); /* línea 102 */
}                               /* línea 103 */
```

__LINE__ es una macro predefinida que contiene el número de la línea de código fuente actual.

__FILE__ es una macro predefinida que contiene el nombre del fichero fuente actual.

La macro #line lo que hace es cambiar los contenidos de estas dos macros.

DIRECTIVA #pragma

La directiva #pragma es una directiva definida por la implementación que permite que se den varias instrucciones al compilador.

La forma general es:

```
#pragma <nombre_de_directiva>
```

Las instrucciones que se dan con esta directiva son dependientes del compilador.

Esto también ocurre con cada versión de Turbo C. Comprueba el manual de usuario o la ayuda del entorno integrado de los distintos compiladores de Turbo C para los detalles y opciones de esta directiva.

NOMBRES DE MACROS PREDEFINIDAS

El estándar ANSI propone cinco macros predefinidas:

Macro	Tipo	Qué contiene
__DATE__	Literal string	Contiene una cadena de la forma mes/día/año que en la fecha de conversión del código fuente a código fuente
__FILE__	Literal string	Contiene una cadena con el nombre del fichero fuente actual
__LINE__	Constante decimal	Contiene el número de la línea actual del fichero fuente que está siendo procesado
__STDC__	Constante	Contiene un 1 si la implementación es estándar y cualquier otro número si la implementación varía de la estándar
__TIME__	Literal string	Contiene una cadena de la forma horas:minutos:segundos que es la hora de conversión del código fuente al código objeto

Ejemplo:

```
/*
  Fichero EJ.C: Programa ejemplo para
  probar las constantes predefinidas
*/

void main (void)
{
  #include <stdio.h>

  printf ("\n__DATE__ : %s", __DATE__);
  printf ("\n__FILE__ : %s", __FILE__);
  printf ("\n__LINE__ : %d", __LINE__);
  printf ("\n__TIME__ : %s", __TIME__);

  #ifdef __STDC__
    printf ("\n__STDC__ : %d", __STDC__);
  #endif
}
```

La salida de este programa en mi implementación es:

```
__DATE__ : Feb 23 1993
__FILE__ : EJ.C
__LINE__ : 12
__TIME__ : 16:20:59
```

Las versiones de Turbo C tienen bastante más constantes simbólicas predefinidas pero como varían en cada sección, no son muy utilizadas por el usuario medio y requieren conocimientos más profundos del sistema que hasta aquí citados, no las vamos a ver. Consulta el manual de usuario o la ayuda del entorno integrado del compilador de Turbo C que usas.

Aunque hay una en todas las versiones de Turbo C que sí es útil: **__TURBOC__**. El valor de esta macro es distinto para cada versión de Turbo C, por lo tanto, tiene dos usos principales: 1) Para saber si estamos compilando en Turbo C o en cualquier otro compilador. 2) Para saber con qué versión de Turbo C estamos compilando.

Ejemplo:

```
#ifdef __TURBOC__
  #include <conio.h> /* highvideo (), cputs () */
  #define alta_intensidad() highvideo()
  #define escribir(cadena) cputs(cadena)
#else
  #include <stdio.h> /* puts () */
  #define alta_intensidad()
  #define escribir(cadena) puts(cadena)
#endif

void main (void)
{
  alta_intensidad (); /* se expande a nada si no Turbo C */
  escribir ("Este mensaje está en alta intensidad ");
  escribir ("si está compilado en Turbo C.");
}
```

El ejemplo anterior se puede compilar en cualquier compilador de C, pero si se compila en uno de Borland el mensaje aparecerá en alta intensidad y si se compila en cualquier otro aparecerá con intensidad normal.

LECCIÓN 6

INTRODUCCION A LA LECCION 6

El objetivo de esta lección es hacer un estudio completo en todo lo referente a la declaración, utilización e inicialización de arrays, tanto unidimensionales, bidimensionales como multidimensionales.

Los puntos que detallaremos son:

- Definición del concepto de array.
- Arrays unidimensionales (vectores). Cadenas de caracteres.
- Arrays bidimensionales (matrices). Arrays de cadenas.
- Arrays multidimensionales.
- Arrays y punteros.
- Inicialización de arrays.

ARRAYS

Un array es una colección de variables del mismo tipo que se referencia por un nombre común.

A un elemento específico de un array se accede mediante un índice. En C todos los arrays constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento. Los arrays pueden tener de una a varias dimensiones.

ARRAYS UNIDIMENSIONALES

La forma general de declaración de un array unidimensional es:

```
especificador_de_tipo  nombre_variable  [tamaño];
```

donde especificador_de_tipo es el tipo base, es decir, el tipo de cada elemento y tamaño es el número de elementos del array.

La forma general de acceder a un elemento del array es:

```
nombre_variable [indice]
```

La longitud en bytes de un array se calcula mediante la fórmula:

```
total_de_bytes = sizeof (tipo) * numero_de_elementos
```

INDICES DE LOS ARRAYS UNIDIMENSIONALES

En C todas los arrays tienen el cero como índice de su primer elemento. Por tanto, cuando se escribe

```
char v[10];
```

se está declarando un array de 10 elementos de tipo entero y el array va de **v[0] a v[9]**.

C no comprueba los límites de los arrays. Esto quiere decir que si hacemos v[20] para el array anterior, el C no nos va a informar de ningún error. Es responsabilidad del programador el indexamiento correcto de un array.

PASO DE ARRAYS UNIDIMENSIONALES COMO PARAMETROS

Hay tres formas de pasar un array unidimensional como parámetro a una función. Consultar el apartado 'ARGUMENTOS DE LAS FUNCIONES' de la lección anterior para recordar cuáles son las tres formas.

UTILIZACION DE ARRAYS UNIDIMENSIONALES COMO CADENAS

El uso más común de los arrays unidimensionales es, con mucho, como cadena de caracteres. **En C una cadena se define como un array de caracteres que termina en un carácter nulo**. Un carácter nulo se especifica como '\0' y generalmente es un cero. Por esta razón, para declarar arrays de caracteres es necesario que sean de un carácter más que la cadena más larga que pueda contener.

Por ejemplo, si se desea declarar un array s para contener una cadena de **10** caracteres, se escribirá

```
char s[11];
```

UTILIZACION DE ARRAYS UNIDIMENSIONALES COMO CADENAS

En C, todo lo que esté encerrado entre comillas es una constante de cadena. Por ejemplo:

```
"cadena"
```

En las constantes de cadenas no es necesario añadir explícitamente el carácter nulo, pues el compilador de C lo hace autómaticamente.

ARRAYS BIDIMENSIONALES

Un array bidimensional es, en realidad, un array unidimensional donde cada elemento es otro array unidimensional. Los arrays bidimensionales son un caso particular de los arrays multidimensionales.

Así como a los arrays unidimensionales se les suele llamar **vectores**, a los arrays bidimensionales se les suele llamar **matrices**.

La forma general de declaración es:

especificador_de_tipo nombre_variable [tamaño_1] [tamaño_2];

y se accede a los elementos del array:

nombre_variable [indice_1] [indice_2]

EJEMPLO DE ARRAY BIDIMENSIONAL

```
void main (void)
{
  #include <stdio.h>
  #define num_filas    4
  #define num_columnas 7
  int i, j, matriz [num_filas] [num_columnas];
  for (i = 0; i < num_filas; i++)
    for (j = 0; j < num_columnas; j++)
      matriz[i][j] = i + j;
  for (i = 0; i < num_filas; i++)
    {
      for (j = 0; j < num_columnas; j++)
        printf ("%2d ", matriz[i][j]);
      putchar ('\n');
    }
}
```

SALIDA DEL EJEMPLO

```
0  1  2  3  4  5  6
1  2  3  4  5  6  7
2  3  4  5  6  7  8
3  4  5  6  7  8  9
```

TAMA¥O EN BYTES DE UN ARRAY BIDIMENSIONAL

El tamaño en bytes de una matriz se calcula mediante la fórmula

bytes_de_memoria = fila * columna * sizeof (tipo)

PASO DE UN ARRAY BIDIMENSIONAL COMO ARGUMENTO A UNA FUNCION

El nombre de un array bidimensional es un puntero al primer elemento del array ([0][0]). Para pasar un array bidimensional como argumento a una función se pasa el puntero al primer elemento. Sin embargo, la función que recibe un array bidimensional como parámetro tiene que definir al menos la longitud de la segunda dimensión. Esto es necesario debido a que el compilador de C necesita conocer la longitud de cada fila para indexar el array correctamente.

MISMO EJEMPLO DE ANTES PERO CON ARRAYS COMO ARGUMENTOS

```
#include <stdio.h>

#define num_filas 4
#define num_columnas 7

void rellenar_matriz (int m[][]), imprimir_matriz (int m[][]);

int i, j;

void main (void)
{
  int matriz [num_filas] [num_columnas];

  rellenar_matriz (matriz);
  imprimir_matriz (matriz);
}
```

MISMO EJEMPLO DE ANTES PERO CON ARRAYS COMO ARGUMENTOS

```
void rellenar_matriz (int m[][num_columnas])
{
  for (i = 0; i < num_filas; i++)
    for (j = 0; j < num_columnas; j++)
      m[i][j] = i + j;
}

void imprimir_matriz (int m[][num_columnas])
{
  for (i = 0; i < num_filas; i++)
    {
      for (j = 0; j < num_columnas; j++)
        printf ("%2d ", m[i][j]);
      putchar ('\n');
    }
}
```

ARRAY DE CADENAS

En C es necesario a veces la utilización de un array de cadenas. Recordar el argv de la función main visto en la lección anterior.

Ejemplo de declaración de un array de 100 elementos en el que cada elemento va a contener una cadena de caracteres de 50 caracteres como máximo:

```
char array_de_cadena [100] [51];
```

El acceso a una cadena del ejemplo anterior se hace:

array_de_cadena[indice]

o

&array_de_cadena[indice][0]

Y el acceso a un carácter de una de las cadenas:

array_de_cadena[indice_1][indice_2]

ARRAY DE CADENAS

```
void main (void)
{
  #include <stdio.h>
  #define NUM_CADENAS 3
  #define LONG_MAX_CADENA 81
  register int i;
  char cadenas [NUM_CADENAS] [LONG_MAX_CADENA];
  puts ("\nINTRODUCCION DE CADENAS:\n");
  for ( i = 0; i < NUM_CADENAS; i++)
    printf ("Cadena %d: ", i), gets (cadenas[i]);
  puts ("LISTADO DE CADENAS INTRODUCIDAS:\n");
  for (i = 0; i < NUM_CADENAS; i++)
    printf ("\nCadena %d: %s", i, cadenas[i]);
}
```

ARRAYS MULTIDIMENSIONALES

C permite arrays de más de dos dimensiones. El límite exacto, si lo hay, viene determinado por el compilador.

La forma general de declaración de un array multidimensional es:

especificador_de_tipo nombre_array [tam_1] [tam_2] ... [tam_n];

NOTA: tam_1, tam_2, ..., tam_n han de ser expresiones constantes.

La forma general de acceso es:

nombre_array [ind_1] [ind_2] ... [ind_n]

PASO DE ARRAYS MULTIDIMENSIONALES COMO ARGUMENTOS A FUNCIONES

Cuando se pasan arrays multidimensionales a funciones, se tiene que declarar todo excepto la primera dimensión. Por ejemplo, si se declara am como

```
int am [4] [3] [6] [5];
```

entonces la función que reciba am podría ser como esta:

```
void func (int a [] [3] [6] [5])
{
```

```
    /* ... */
  }
```

Por supuesto, si se quiere se puede incluir la primera dimensión.

A R R A Y S Y P U N T E R O S

En C los punteros y los arrays están estrechamente relacionados. Por ejemplo, un nombre de array es un puntero al primer elemento del array.

Ejemplo:

```
void main (void)
{
  #include <stdio.h>
  char p[10];
  printf ("p == &p[0] : %d", p == &p[0]);
}
```

La salida por pantalla de este programa es:

```
p == &p[0] : 1
```

INICIALIZACION DE ARRAYS

La forma general de inicialización de un array en la que aparece a continuación:

```
especificador_de_tipo nombre_array [tamaño_1] ... [tamaño_N] =
  { lista_de_valores };
```

Como se ve, la inicialización de arrays es similar a la inicialización de variables.

La lista_de_valores es una lista de constantes, separadas por comas, cuyo tipo es compatible con especificador_de_tipo.

Después de } ha de haber un ;.

Ejemplo de inicialización de un vector:

```
int v[5] = { 1, 2, 3, 4, 5 };
```

La inicialización de cadenas se puede hacer de dos formas:

```
char cadena[4] = "abc";   char cadena[4] = { 'a', 'b', 'c', '\0' };
```

Hay dos formas de inicializar arrays multidimensionales:

```
int m [3] [4]
  {
    1, 2, 3, 4,
    5, 6, 7, 8,
```

```
int m [3] [4]
  {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
```

```
      9, 10, 11, 12            { 9, 10, 11, 12 }
    };                       };
```

No es necesario que estén todos los elementos en las inicializaciones de arrays. Los elementos que falten se inicializan a 0 o quedan sin valor fijo, según el compilador. Por ejemplo:

```
   int v[5] = { 1, 2 };
```

Observar que no se asigna los mismos valores a m1 que a m2 en el siguiente ejemplo:

```
    int m1 [3] [2]     int m2 [3] [2]    El valor 5 se asigna
      {                  {               en el primer caso a
        2, 3,              { 2, 3 },     m1[1][1] y en el se-
        4,                 { 4 },        gundo caso a m2[2][0].
        5, 6               { 5, 6 }
      };                 };
```

La forma más común de inicialización de arrays es sin especificar el tamaño. Por ejemplo:

```
  int v [] = { 2, 3, 4 };
```

En este ejemplo, el compilador reserva memoria para los elementos de v: v[0], v[1] y v[2], y les asigna los valores 2, 3 y 4 respectivamente.

Otro ejemplo:

```
  char cadena [] = "esto es una cadena de prueba";
```

En los arrays multidimensionales es necesario especificar el tamaño de todas las dimensiones excepto el de la primera que es opcional.

Ejemplo:

```
  int m [] [4] =
  {
     { 1, 2, 3, 4 },
     { 5, 6, 7, 8 }
  };
```

La ventaja de las inicializaciones de arrays sin especificar tamaño es que se puede insertar y quitar elementos sin modificar las dimensiones del array. Otra ventaja es que nosotros no tenemos que contar todos los elementos del array para dimensionarlo, así que dejamos que esta tarea la realice el compilador.

EJEMPLO

```
#include <stdio.h>
void main (void)
{
  /*
    si en tu compilador da problema las dos inicializaciones de arrays
    que se van a definir, lo puede solucionar de dos formas: o bien
    haces estos dos arrays globales o los hace locales estáticos
  */
  char meses[12][11] =
    {
      "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
      "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"
    };
```

```
  int dias[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  register short i;
  for (i = 0; i < 12; i++)
    printf ("El mes de %-10s tiene %d días.\n", meses[i], dias[i]);
}
```

SALIDA DE LA EJECUCION DE ESTE EJEMPLO

```
  El mes de Enero      tiene 31 días.
  El mes de Febrero    tiene 28 días.
  El mes de Marzo      tiene 31 días.
  El mes de Abril      tiene 30 días.
  El mes de Mayo       tiene 31 días.
  El mes de Junio      tiene 30 días.
  El mes de Julio      tiene 31 días.
  El mes de Agosto     tiene 31 días.
  El mes de Septiembre tiene 30 días.
  El mes de Octubre    tiene 31 días.
  El mes de Noviembre  tiene 30 días.
  El mes de Diciembre  tiene 31 días.
```

LECCIÓN 7

INTRODUCCION A LA LECCION 7

El objetivo de esta lección es hacer un estudio completo en todo lo referente a la declaración, utilización e inicialización de punteros.

Los puntos que detallaremos son:
- Definición del concepto de puntero.
- Declaración de punteros.
- Operadores de punteros: **&** y *****.
- Aritmética de punteros.
- Asignación dinámica de memoria.
- Punteros y arrays.
- Inicializaciones de punteros.
- Punteros a funciones.
- Funciones **qsort()** y **bsearch()**.
- Funciones de Turbo C: **lfind()** y **lsearch()**.

- Tipo void y punteros.
- Modificador de acceso const y punteros.
- Declaraciones curiosas.

PUNTEROS

El concepto de puntero es importantísimo en la programación en C.

Un puntero contiene una dirección de memoria.
Cuando una variable contiene la dirección de otra variable se dice que la primera variable apunta a la segunda.

VARIABLES PUNTEROS

La forma general para declarar una variable puntero es:

tipo *nombre;

donde tipo es cualquier tipo válido de C (también llamado tipo base) y nombre es el nombre de la variable puntero.

OPERADORES DE PUNTEROS

Existen dos operadores especiales de punteros: **&** y *****. Estos dos operados son monarios y no tienen nada que ver con los operadores binarios de multiplicación (*) y de and a nivel de bits (&).

OPERADOR &

& es un operador monario que devuelve la dirección de memoria de su operando.

Ejemplo:

```
#include <stdio.h>
void main (void)
{
  int x = 10;
  printf (" x = %d\n &x = %p\n",
          x, &x);
}
```

Salida del ejemplo anterior:

```
x = 10
&x = 8FBC:0FFE
```

NOTA: El valor y formato de las direcciones de las variables que se imprimen en esta lección (con el código de formato %p) son dependientes de la implementación. En mi caso las direcciones se escriben en formato segmento:offset, y el valor de la dirección de una variable es distinto según cuándo y desde dónde se ejecute el programa.

*OPERADOR **

El operador * es el complemento de &. Es un operador monario que devuelve el valor de la variable localizada en la dirección que sigue.

Ejemplo 1:

```
#include <stdio.h>
void main (void)
{
  int x = 10;
  printf ("  x = %d\n", x);
  printf ("  *&x = %d", *&x);
}
```

Salida de ejemplo 1:

```
x = 10
*&x = 10
```

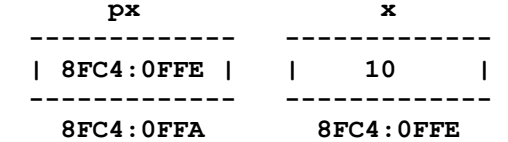
Ejemplo 2:

```
#include <stdio.h>
void main (void)
{
  int x = 10;
  int *px;
  px = &x;
  printf ("  x = %d\n", x);
  printf ("  &x = %p\n", &x);
  printf ("  px = %p\n", px);
  printf ("  &px = %p\n", &px);
  printf ("  *px = %d", *px);
}
```

Salida de ejemplo 2:

```
x = 10
&x = 8FC4:0FFE
px = 8FC4:0FFE
&px = 8FC4:0FFA
*px = 10
```

Gráficamente:



En el ejemplo anterior se observa que hay tres valores asociados a los punteros:

- Dirección en la que se encuentra.
- Dirección a la que apunta.
- Valor contenido en la dirección apuntada.

OJO!! Al ejecutar el siguiente programa los resultados pueden ser inesperados ya que estamos asignando el valor 10 en una posición de memoria no reservada:

```
void main (void)
{
   int *p;
   *p = 10;
}
```

Sería correcto:

```
void main (void)
{
   int x;/*se reserva memoria para x*/
   int *p;/*se reserva memoria para la
       variable p, no para la posición
       de memoria a la que apunta p */
   p = &x; /* p apunta a valor de x */
   *p = 10;/* equivalente a: x = 10 */
}
```

ARITMETICA DE PUNTEROS

Existen 4 operadores que realizan operaciones aritméticas que pueden utilizarse con punteros:

+, -, ++, --

```
/* EJEMPLO DE ARITMETICA DE PUNTEROS */
#include <stdio.h>

void main (void)
{
  int *p;
  #define imprimir_p printf ("\np = %p", p);

  imprimir_p;
  printf ("\tsizeof(*p) = %d", sizeof(*p));
  p++; imprimir_p;
  p -= 5; imprimir_p;
}

/* SALIDA DEL EJEMPLO:
  p = 7203:8D51        sizeof(*p) = 2
  p = 7203:8D53
  p = 7203:8D49
*/
```

En el ejemplo anterior se aprecia que si hacemos p++; no aumenta el valor de p en 1 sino que aumenta en 2, que es el tamaño en bytes de un int, es decir, el tamaño del objeto al que apunta.

Por lo tanto, la sentencia p++ hay que interpretarla

de la siguiente forma: p apunta al siguiente elemento del tipo base . Lo mismo se puede decir de los demás operadores aritméticos aplicados a los punteros.

Toda la aritmética de punteros está en relación con el tipo base del puntero por lo que el puntero está siempre apuntando al elemento apropiado del tipo base.

ASIGNACION DINAMICA DE MEMORIA

Supóngase que queremos hacer un programa que lea n valores enteros introducidos por teclado por el usuario, los almacene en un vector y los imprima en orden inverso.

Una solución es:

```
#include <stdio.h>"

void main (void)
{
  #define NMAX 100 /* número máximo de elementos */
  int v[NMAX]; /* vector */
  int n = 0; /* número de elementos introducidos */
  int varaux; /* variable auxiliar */
  register int i; /* índice */

  do
    {
      printf ("\nIntroduce número de valores a leer (1-%d): ", NMAX);
      scanf ("%d", &n);
    } while (n < 1 || n > NMAX);

  for (i = 0; i <= n - 1; i++)
    {
      printf ("Introduce valor %d: ", i);
      scanf ("%d", &varaux);
      v[i] = varaux;
    }

  printf ("\n\nValores en orden inverso:\n");
  for (i = n - 1; i >= 0; i--)
    printf ("%d ", v[i]);
}
```

Si el usuario introduce como valor de n, el valor 10, estaremos desperdiciando, si un int ocupa 2 bytes, 90*2 bytes de memoria. Además, el usuario no puede introducir más de NMAX valores. Estas restricciones vienen impuestas poque el tamaño de un array en la declaración ha de ser una expresión constante. La asignación de memoria en este caso se dice que es estática porque se determina en el momento de la compilación. Cuando la asignación de memoria se determina en tiempo de ejecución se dice que es asignación dinámica.

Veamos cómo se haría el programa anterior con asignación dinámica y luego pasamos a explicarlo:

```
#include <stdio.h> /* printf (), scanf () */
#include <alloc.h> /* malloc (), free () */

void main (void)
{
  int *v; /* vector */
  int n = 0; /* número de elementos introducidos */
  int varaux; /* variable auxiliar */
  register int i; /* índice */

  printf ("\nIntroduce número de valores a leer: ");
  scanf ("%d", &n);

  v = (int *) malloc (n * sizeof (int));
  if (v == NULL)
    printf ("Memoria insuficiente.");
  else
    {
      for (i = 0; i <= n - 1; i++)
        {
          printf ("Introduce valor %d: ", i);
          scanf ("%d", &varaux);
          v[i] = varaux;
        }

      printf ("\n\nValores en orden inverso:\n");
      for (i = n - 1; i >= 0; i--)
        printf ("%d ", v[i]);

      free (v);
    }
}
```

La primera sentencia de main() es:

```
int *v;
```

En esta declaración estamos declarando v como un puntero a entero.

La siguiente línea extraña para nostros es:

```
v = (int *) malloc (n * sizeof (int));
```

La función malloc reserva memoria; acepta como argumento los bytes de memoria a reservar y devuelve un puntero al primer byte de la zona de memoria reservada; los bytes de memoria solicitados los reserva en un espacio de memoria contiguo. Si no hay suficiente memoria, devuelve NULL. Un puntero que tiene el valor NULL es un puntero que no apunta a ningún sitio.

El prototipo de esta función se encuentra en el fichero malloc.h (de ahí el incluir este fichero en nuestro ejemplo) y es el siguiente:

```
void *malloc (unsigned int bytes);
```

Vemos que devuelve un puntero a void; esto quiere decir que devuelve un puntero que apunta a cualquier tipo base, o dicho de otro modo, un puntero que apunta a una dirección de memoria sin tener tipo base.

El C de Kernighan y Ritchie, como no tiene tipo void, el prototipo de esta función es:

```
char *malloc (unsigned int bytes);
```

En ambos casos, el tratamiento por parte del usuario de esta función es exactamente el mismo.

Veamos otra vez la llamada a esta función en nuestro ejemplo:

```
v = (int *) malloc (n * sizeof (int));
```

Al valor devuelto por la función malloc (puntero a void o puntero a char) siempre se le realiza un moldeado (recordad que esto se hacía con: (tipo)) para adecuarlo al tipo base de nuestro puntero que va a apuntar a esa zona de memoria reservada. En nuestro caso el molde es:

```
(int *) /* puntero a entero */
```

El argumento que le pasamos a malloc ha de ser el número de bytes de memoria a reservar. Esto siempre se hace siguiendo la fórmula:

```
numero_de_elementos * sizeof (tipo_de_cada_elemento)
```

que traducido a nuestro caso queda:

```
n * sizeof (int)
```

Otra forma de hacer lo mismo es:

```
n * sizeof (*v)
```

que suele ser muy corriente. Las dos formas son equivalentes.

La memoria asignada por malloc se desasigna con la función free(). Esta memoria asignada no se desasigna al salir del bloque de código en que fue asignada como ocurre con las variables locales sino con la función free (liberar) o al terminar el programa. Por lo tanto, siempre que asignemos memoria con malloc, tenemos que desasignarla con free cuando ya no nos sea necesaria.

El prototipo de la función free se encuentra en el fichero malloc.h y es el siguiente:

```
void free (void *p);
```

Un puntero a void como parámetro indica que acepta cualquier puntero, independientemente del tipo base al que apunta. El puntero que se le pasa a free como argumento ha de ser un puntero que apunta al principio de una zona reservad anteriormente por malloc; si no es así, se puede caer el sistema.

El resto del ejemplo no tiene ya ninguna dificultad.

Hay otra función, en la librería <alloc.h>, equivalente a malloc, que es la función calloc cuyo prototipo es el siguiente:

```
void *calloc (unsigned numero_de_elementos_a_reservar,
              unsigned tamanio_en_bytes_de_cada_elemento);
```

Esta función es igual que malloc con la única diferencia de sus parámetros.

En Turbo C, los prototipos de las funciones malloc(), calloc() y free(), además de estar en el fichero alloc.h, también están en el

fichero stdlib.h.

PUNTEROS Y ARRAYS

Exite una estrecha relación entre los punteros y los arrays. Ya hemos dicho en varias ocasiones en lecciones anteriores que el nombre de un array es un puntero al primer elemento del array. A cualquier elemento de un array podemos acceder mediante la aritmética de punteros y viceversa, cualquier puntero lo podemos indexar con los []:

```
Arrays unidimensionales:
  p[i] == *(p+i)
Arrays bidimensionales:
  p[i][j] == *(p+(i*longitud_fila)+k) == *(*(p+i)+j)
Arrays muldimensionaes:
  se sigue cualquiera de los dos procedimientos
  ejemplarizados en los arrays bidimensionales

  /*
    EJEMPLO DE ACCESO A UN ARRAY CON UN PUNTERO
  */

  #include <stdio.h>

  void main (void)
  {
    float v[3] = { 1.1, 2.2, 3.3 };
    printf ("v[1] = %g; *(v+1) = %g", v[1], *(v+1));
  }

  /*
    SALIDA POR PANTALLA: v[1] = 2.2; *(v+1) = 2.2
  */
  /* EJEMPLO DE ACCESO A ELEMENTOS INDEXANDO UN PUNTERO CON [] */
  #include <stdio.h> /* printf (), NULL */
  #include <alloc.h> /* malloc (), free () */
  void main (void)
  {
    float *p;
    if ((p = (float *) malloc (3 * sizeof (float))) == NULL)
      printf ("\nERROR: Memoria Insuficiente.");
    else
      {
        *p = 1.1; *(p+1) = 2.2; *(p+3) = 3.3;
        printf ("*(p+1) = %g p[1] = %g", *(p+1), p[1]);
        free (p);
      }
  }
  /* SALIDA: *(p+1) = 2.2; p[1] = 2.2 */
```

Los programadores profesionales de C suelen utilizar la notación puntero en vez de la notación array porque es bastante más rápido y más cómodo, aunque para los no acostrumbrados a esta notación se ve un poco extraño al principio. Pensad que con notación array, para acceder a un determinado elemento, el compilador tiene que hacer una serie de cálculos para averiguar en qué posición está, mientras que en la notación puntero basta con una simple suma. No obstante, cuando el código queda más claro en la notación array que con la notación puntero es preferible la primera notación.

Cuando se trabaja con cadenas de caracteres sí se sebe utilizar la notación puntero, no ya sólo por eficiencia sino también por convención. Una estructura común en C es el array de punteros. Recordar que el argumento argv de la función main() es un array de punteros a caracteres.

Hay tres formas equivalentes de declarar el argumento argv en la función main():

```
  main (int argc, char argv[][]);
  main (int argc, char *argv[]);
  main (int argc, char **argv);
```

Habrás observado que en la primera declaración no se ha especificado el tamaño de la segunda dimensión de argv cuando habíamos dicho antes que era necesario, esto sólo está permitido hacerlo en la función main(). La declaración para un array de 10 punteros a int es:

```
  int *x[10];
```

Para asignar la dirección de una variable entera llamada var al tercer elemento del array de punteros, se ecribe:

```
  x[2] = &var;
```

Para encontrar el valor de var, se escribe:

```
  *x[2]
/* EJEMPLO DE ARRAY DE CADENAS DE CARACTERES */
void error (int numero_de_error)
{
  char *errores [] =
    {
      "error al intentar abrir fichero",
      "error al intentar cerrar fichero",
      "error al intentar leer de fichero",
      "error al intentar escribir en fichero"
    };
  printf ("%s", errores[numero_de_error]);
  exit (1);
}
/*
  Un array de punteros es lo mismo que punteros a punteros.
  Este ejemplo comprueba dicha afirmación.
*/
#include <stdio.h>
void main (void)
{
  int x, *p, **q;
  x = 10; p = &x; q = &p;
  printf ("x = %d; *p = %d; **q = %d", x, *p, **q);
}
/* Salida: x = 10; *p = 10; **q = 10 */
```

A continuación vamos a mostrar dos formas de implementar la siguiente función: la función a implementar acepta como argumentos una matriz de enteros y un elemento, y devuelve 1 si ese elemento se encuentra en la matriz o 0 si dicho elemento no se encuentra en la matriz.

```
  /*
    Las funciones buscar_en_matriz_version_1() y
    buscar_en_matriz_version_2() son equivalentes
    pero la segunda es mucho más eficiente.
  */
```

```
#include <stdio.h>

#define N 3

int buscar_en_matriz_version_1 (int m[N][N], int x)
{
  register int i, j;
  int encontrado = 0;

  for (i = 0; ! encontrado && i < N; i++)
    for (j = 0; ! encontrado && j < N; j++)
      if (m[i][j] == x)
        encontrado = 1;

  return (encontrado);
}

int buscar_en_matriz_version_2 (int m[N][N], int x)
{
  register int i;
  int encontrado = 0;
  int *pm = m; /* declara pm como puntero a int y lo hace apuntar a m */

  for (i = 1; ! encontrado && i <= N*N; i++)
    if (*pm == x)
      encontrado = 1;
    else
      pm++;

  return (encontrado);
}

void main (void)
{
  int matriz [N][N] = { { 1, 2, 3 }, { -1, -2, -3 }, { 5, 6, 7 } };
  int resultado_1 = buscar_en_matriz_version_1 (matriz, 6);
  int resultado_2 = buscar_en_matriz_version_1 (matriz, 8);
  int resultado_3 = buscar_en_matriz_version_2 (matriz, 6);
  int resultado_4 = buscar_en_matriz_version_2 (matriz, 8);

  printf ("\nresultado_1 = %d", resultado_1);
  printf ("\nresultado_2 = %d", resultado_2);
  printf ("\nresultado_3 = %d", resultado_3);
  printf ("\nresultado_4 = %d", resultado_4);
}

/*
  SALIDA:

  resultado_1 = 1
  resultado_2 = 0
  resultado_3 = 1
  resultado_4 = 0
*/
```

INICIALIZACIONES DE PUNTEROS

Un puntero que tiene el valor NULL es un
puntero que no apunta a ningún sitio.

Una inicialización muy común en C se ilustra
con el siguiente ejemplo:

```
  char *p = "cadena\n";
```

En este caso el compilador guarda "cadena\n"
en memoria y hace que p apunte al principio
de la cadena, es decir, el carácter 'c'.

EJEMPLOS DE INCIALIZACIONES EQUIVALENTES

```
int x = 10;                    int x = 10;
int *p = &x;          <==>     int *p;
                               p = &x;


                               int x;
int x, *p, y;         <==>     int *p;
                               int y;


                                  int *p;
int *p, *q, r = 10;       <==>    int *q;
                                  int r = 10;


                                  int v[2] = { 1, 2 };
int v[2] = { 1, 2 }, f (void), *p, x = 3; <==> int f (void);
                                  int *p;
                                  int x = 3;
```

partir

PUNTEROS A FUNCIONES

Una característica algo confusa pero muy útil de C es el puntero a
función. La confusión surge porque una función tiene una posición
física en memoria que puede asignarse a un puntero aunque la función
no es una variable. La dirección de la función es el punto de entrada
de la función; por tanto, un puntero a función puede utilizarse para
llamar a la función.

Ejemplo:

```
~#include <stdio.h> /* printf () */
~
~void main (void)
~{
~  /* escribir es una función que acepta un int como argumento
~     y no devuelve nada */
~  void escribir (int);
~
~  /* pf es un puntero a una función que acepta un int como argumento
~     y no devuelve nada */
~  void (*pf) (int);
~
~  pf = escribir;
```

```
~  escribir (1); /* llama a la función escribir */
~  (*pf) (2); /* llama a la función escribir a través de un puntero */
~}
~
~void escribir (int numero)
~{
~  printf ("%d", numero);
~}
```

La salida de este programa por pantalla es:
12

Una utilidad de los punteros a funciones la tenemos en las funciones **qsort()** y **bsearch()** cuyos prototipos están en el fichero stdlib.h.

qsort ()

El prototipo de la función qsort() está en el fichero stdlib.h y es:

```
  void qsort (void *base, unsigned int num, unsigned int tam,
              int (*compara) (void *arg1, void *arg2));
```

La función qsort() ordena el array apuntado por base utilizando el método de ordenación de C.A.R. Hoare (este método se ha explicado en el ejemplo 3 de la lección 5). El número de elementos en el array se especifica mediante num, y el tamaño en bytes de cada elemento está descrito por tam.

La función compara se utiliza para comparar un elemento del array con la clave. La comparación debe ser:

```
  int nombre_func (void *arg1, void *arg2);
```

Debe devolver los siguientes valores:

```
  Si arg1 es menor que arg2, devuelve un valor menor que 0.
  Si arg1 es igual a arg2 devuelve 0.
  Si arg1 es mayor que arg2, devuelve un valor mayor que 0.
```

El array es ordenado en orden ascendente con la dirección más pequeña conteniendo el menor elemento.

En Turbo C, el prototipo de la función qsort() es ligeramente diferente:

```
  void qsort (void *base, size_t num, size_t tam,
              int (*compara) (const void *, const void *));
```

size_t es un tipo definido en el fichero stdlib.h y suele ser unsigned int; (const void *) no es lo mismo que (void *), la diferencia entre ellos se va a estudiar unas tres o cuatro ventanas más adelante, pero podemos intuirla: en (const void *) el objeto apuntado es constante, es decir, no se puede modificar, en (void *) el objeto apuntado por el puntero sí se puede modificar.

Veamos un ejemplo de la utilización de esta función, donde podemos apreciar además, dos formas posibles de declaración y utilización de la función de comparación requerida por la función qsort().

```
~#include <stdio.h>  /* printf () */
~#include <stdlib.h> /* qsort () */
```

```
~
~void main (void)
~{
~  int num[10] = { 3, 2, 8, 9, 2, 2, 1, -2, 3, 2 };
~  register int i;
~  int comparar_creciente (const void *elem1, const void *elem2);
~  int comparar_decreciente (const int *elem1, const int *elem2);
~
~  printf ("\nArray desordenado: ");
~  for (i = 0; i < 10; i++)
~    printf ("%d ", num[i]);
~
~  qsort (num, 10, sizeof (int), comparar_creciente);
~
~  printf ("\nArray ordenado en orden creciente: ");
~  for (i = 0; i < 10; i++)
~    printf ("%d ", num[i]);
~
~  /*
~     el molde del cuarto argumento convierte el tipo
~       (int (*) (const int *, const int *))
~     al tipo
~       (int (*) (const void *, const void *))
~     que es el que requiere la función qsort
~  */
~  qsort (num, 10, sizeof (int),
~         (int (*) (const void *, const void *)) comparar_decreciente);
~
~  printf ("\nArray ordenado en orden decreciente: ");
~  for (i = 0; i < 10; i++)
~    printf ("%d ", num[i]);
~}
~
~int comparar_creciente (const void *elem1, const void *elem2)
~{
~  /* para acceder al contenido de un puntero del tipo (void *)
~     necesitamos moldearlo a un tipo base que no sea void */
~  return (*(int *)elem1 - *(int *)elem2);
~}
~
~int comparar_decreciente (const int *elem1, const int *elem2)
~{
~  return (*elem2 - *elem1);
~}
```

La salida de este programa por pantalla es:

```
  Array desordenado: 3 2 8 9 2 2 1 -2 3 2
  Array ordenado en orden creciente: -2 1 2 2 2 2 3 3 8 9
  Array ordenado en orden decreciente: 9 8 3 3 2 2 2 2 1 -2
```

bsearch ()

El prototipo de la función bsearch() se encuentra en el fichero stdlib.h y es el siguiente:

```
  void *bsearch (void *clave, void *base, unsigned int num,
    unsigned int tam, int (*compara) (void *arg1, void *arg2));
```

La función bsearch() realiza una búsqueda binaria en el array ordenado apuntado por base y devuelve un puntero al primer elemento que se corresponde con la clave apuntada por clave. El número de elementos en el array está especificado por num y el tamaño (en bytes) de cada elemento está descrito por tam.

La función apuntada por compara se utiliza para comparar un elemento del array con la clave. La forma de la función de comparación debe ser:

```
  nombre_func (void *arg1, void *arg2);
```

Debe devolver los siguientes valores:

```
  Si arg1 es menor que arg2, devuelve un valor menor que 0.
  Si arg1 es igual que arg2, devuelve 0.
  Si arg1 es mayor que arg2, devuelve un valor mayor que 0.
```

El array debe estar ordenado en orden ascendente con la menor dirección conteniendo el elemento más pequeño. Si el array no contiene la clave, se devuelve un puntero nulo.

Esta función está implementada en uno de los ejemplos de la lección 3.

En Turbo C, el prototipo de la función bsearch() es ligeramente diferente:

```
  void *bsearch (const void *clave, const void *base, unsigned int *num,
    unsigned int tam, int (*compara) (const void *arg1, const void *arg2));
```

Los tipos size_t y (const void *) se han explicado en la ventana anterior: qsort().

Ejemplo:

```
~#include <stdlib.h>
~#include <stdio.h>
~
~#define NUM_ELEMENTOS(array) (sizeof(array) / sizeof(array[0]))
~
~int array_de_numeros[] = { 123, 145, 512, 627, 800, 933, 333, 1000 };
~
~int comparacion_de_numeros (const int *p1, const int *p2)
~{
~  return (*p1 - *p2);
~}
~
~int buscar (int clave)
~{
~   int *puntero_a_elemento;
~
~   /* El molde (int (*) (const void *, const void *)) es necesario para
~      evitar un error de tipo distinto en tiempo de compilación. Sin
~      embargo, no es necesario: puntero_a_elemento = (int *) bsearch (...
~      debido a que en este caso es el compilador el que realiza la
~      conversión de tipos */
~   puntero_a_elemento = bsearch (&clave, array_de_numeros,
~     NUM_ELEMENTOS (array_de_numeros), sizeof (int),
~     (int (*) (const void *, const void *)) comparacion_de_numeros);
~
~   return (puntero_a_elemento != NULL);
~}
~
~int main (void)
~{
```

```
~  if (buscar (800))
~    printf ("800 está en la tabla.\n");
~  else
~    printf ("800 no está en la tabla.\n");
~
~  return 0;
~}
```

La salida de este programa por pantalla es:

```
  800 está en la tabla.
```

lfind () y lsearch ()

Estas dos funciones no pertenecen al ANSI C actual pero sí están incluidas en las versiones de Turbo C.

Ambas funciones realizan una búsqueda lineal.

Sus prototipos están en el fichero stdlib.h y son los siguientes:

```
void *lfind (const void *clave, const void *base,
             size_t *num, size_t anchura,
             int (*func_de_comp) (const void *elem1, const void *elem2));

void *lsearch (const void *clave, void *base, size_t *num, size_t anchura,
               int (*func_de_comp) (const void *elem1, const void *elem2));
```

Estas funciones utilizan una rutina definida por el usuario (func_de_comp) para la búsqueda de la clave, en un array de elementos secuenciales.

El array tiene num elementos, cada uno de tamaño anchura bytes y comienza en la dirección de memoria apuntada por base.

Devuelve la dirección de la primera entrada en la tabla que coincida con la clave buscada. Si la clave buscada no se encuentra, lsearch la añade a la lista; lfind devuelve 0.

La rutina *func_de_comp debe devolver cero si *elem1 == *elem2, y un valor distinto de cero en caso contrario.

Ejemplo de la función lfind:

```
~#include <stdio.h>
~#include <stdlib.h>
~
~int comparar (int *x, int *y)
~{
~  return (*x - *y);
~}
~
~int main (void)
~{
~  int array[5] = { 5, -1, 100, 99, 10 };
~  size_t nelem = 5;
~  int clave;
~  int *resultado;
~
```

```
~  clave = 99;
~  resultado = lfind(&clave, array, ≠lem, sizeof (int),
~              (int (*) (const void *, const void *)) comparar);
~  if (resultado)
~    printf ("Número %d encontrado\n", clave);
~  else
~    printf ("Número %d no encontrado.\n", clave);
~
~  return 0;
~}
```

La salida de este programa es:

```
  Número 99 encontrado.
```

Ejemplo de la función lsearch:

```
~#include <stdlib.h>
~#include <stdio.h>
~
~int numeros[10] = { 3, 5, 1 };
~int nnumeros = 3;
~
~int comparar_numeros (int *num1, int *num2)
~{
~  return (*num1 - *num2);
~}
~
~int aniadir_elemento (int numero_clave)
~{
~  int viejo_nnumeros = nnumeros;
~
~  lsearch ((void *) νmero_clave, numeros,
~           (size_t *) &nnumeros, sizeof (int),
~           (int (*) (const void *, const void *)) comparar_numeros);
~
~  return (nnumeros == viejo_nnumeros);
~}
~
~int main (void)
~{
~  register int i;
~  int clave = 2;
~
~  if (aniadir_elemento (clave))
~    printf ("%d está ya en la tabla de números.\n", clave);
~  else
~    printf ("%d está añadido a la tabla de números.\n", clave);
~
~  printf ("Números en tabla:\n");
~  for (i = 0; i < nnumeros; i++)
~    printf ("%d\n", numeros[i]);
~
~  return 0;
~}
```

La salida de este programa es:

```
  2 está añadido a la tabla de números.
  Números en tabla
  3
  5
  1
  2
```

TIPO void Y PUNTEROS

Hasta esta lección hemos visto que el tipo void tiene dos usos:

1.- Para indicar que una función no devuelve nada. Ejemplo:
void hola (char *nombre) { printf ("Hola, %s", nombre); }

2.- Para indicar que una función no acepta ningún argumento.
Ejemplo: **int numero_1 (void) { return 1; }**

Aplicados a punteros tiene otro uso: los punteros a void son punteros genéricos que pueden apuntar a cualquier objeto. Pero los punteros a void no pueden ser referenciados (utilizando *) sin utilizar moldes, puesto que el compilador no puede determinar el tamaño del objeto al que apunta el puntero. Ejemplo:

```
int x; float f;
void *p = &x; /* p apunta a x */
int main (void)
  {
    *(int *) p = 2;
    p = &r;  /* p apunta a f */
    *(float *) p = 1.1;
  }
```

MODIFICADOR DE ACCESO const Y PUNTEROS

El modificador de acceso const aplicado a punteros ofrece varias posibilidades. Veámosla en los siguientes ejemplos:

```
void main (void)
{
  char *p1 = "abc";              /* puntero */
  const char *p2 = "abc";        /* puntero a constante */
  char *const p3 = "abc";        /* puntero constante */
  const char *const p4 = "abc"; /* puntero constante a constante */

  *p1 = 'd'; /* correcto */
  *p2 = 'd'; /* error */
  *p3 = 'd'; /* correcto */
  *p4 = 'd'; /* error */

  p1++; /* correcto */
  p2++; /* correcto */
  p3++; /* error */
  p4++; /* error */

  p1 = p2; /* warning */
  p1 = p3; /* correcto */
  p1 = p4; /* warning */

  p2 = p1; /* correcto */
  p2 = p3; /* correcto */
  p2 = p4; /* correcto */

  p3 = p1; /* error */
```

```
    p3 = p2; /* error */
    p3 = p4; /* error */

    p4 = p1; /* error */
    p4 = p2; /* error */
    p4 = p3; /* error */
  }
```

Las líneas que contienen el mensaje de error provocan un error de compilación. Las líneas que contienen el mensaje de warning provoca en algunos compiladores un mensaje de conversión de puntero sospechosa que se puede solucionar haciendo la conversión de una forma explícita:

```
  p1 = (char *) p2;
  p1 = (char *) p4;
```

Si ahora hacemos:

```
  *p1 = 'd';
```

estamos modificando los valores apuntados por p2 y p4; es decir, los valores apuntados por p2 y p4 no pueden ser modificados por estos punteros pero sí pueden ser modificados indirectamente por otro puntero.

Otro ejemplo de cómo se puede modificar el valor de una constante indirectamente a través de un puntero:

```
  const int x = 10; /* x es una variable constante */
  x = 20; /* esto provoca un error en compilación */
  *(int *)&x = 20; /* esto es correcto: obtenemos su dirección, que
                      es del tipo (const int *) y la moldeamos al tipo
                      (int *), una vez hecho esto accedemos al valor de
                      esa dirección con el operador de contenido * */
```

DECLARACIONES CURIOSAS

El C permite la creación de formas de datos muy elaboradas.

Cuando se hace una declaración, el nombre (o "identificador") que usamos se puede modificar añadiéndole uno o varios modificadores:

Modificador	Significado
*	indica un puntero
( )	indica una función
[ ]	indica un array

La clave para desentrañar las declaraciones que mostraremos a continuación es averiguar el orden en que se aplican los modificadores. Para ello se siguen tres reglas:

1. La prioridad de un modificador es tanto mayor cuanto más próximo esté el identificador.

2. Los modificadores [] y () tienen mayor prioridad que *.

3. Se pueden usar paréntesis para agrupar parte de la expresión otorgándole la máxima prioridad.

Ejemplos:

```
  void main (void)
  {
    /* array de arrays de int */
    int x1[8][8];

    /* puntero a puntero a int */
    int **x2;

    /* array de 10 punteros a int */
    int *x3[10];

    /* puntero a array de 10 int */
    int (*x4)[10];

    /* array de 3 punteros a array de 4 int */
    int *x5[3][4];

    /* puntero a array de 3x4 int */
    int (*x6)[3][4];

    /* función que devuelve un puntero a int */
    int *x7(void);

    /* puntero a función que devuelve un tipo int */
    int (*x8)(void);

    /* función que acepta un puntero a char como argumento y que
       devuelve un puntero a un puntero a una función que devuelve
       un carácter */
    char (*(*x11(char*)))(void);

    /* puntero a función que devuelve un puntero a puntero a carácter
       y acepta dos argumentos: el primero es un puntero a puntero a
       puntero a carácter, y el segundo es un array de 10 punteros a
       carácter */
    char ** (*x12) (char ***, char *[10]);

    /* función que acepta un puntero a puntero a puntero a constante y
       devuelve una puntero constante a constante */
    const void * const x13 (const void ***);

    /* función que no devuelve nada y acepta como argumento un puntero
       a función que no devuelve nada y acepta como argumento un puntero
       a función que no devuelve nada y no acepta ningún argumento */
    void x14 (void (*) (void (*) (void)));

    /* función que acepta un int como argumento y devuelve un puntero
       a una función que acepta un int como argumento y que devuelve
       un int */
    int (* (x15 (int))) (int);
  }
```

MODIFICADORES DE TIPO near, far, huge Y MODELOS DE MEMORIA EN TURBO C

En MS-DOS se suele direccionar memoria en la forma:

```
  segmento:offset
```

donde cada segmento son 64K y offset es el desplazamiento dentro de cada segmento.

Los modificadores de tipo near, far y huge, que sólo se pueden aplicar a punteros y funciones, están relacionados con el direccionamiento de memoria.

El modificador near (cerca) fuerza que el valor de un puntero o una función estén en un sólo segmento mientras que far (lejos) y huge (enorme) hacen que un puntero pueda apuntar a cualquier dirección de la memoria y una función puede estar en culquier lugar de la memoria disponible.

A continuación exponemos la descripción de los modificadores near, far y huge así como los distintos modelos de compilación en Turbo C:

---
near (modificador de tipo)
==========

Fuerza a los punteros a ser near (estar cerca, un mismo segmento), genera código de funciones para llamadas near y devuelve un valor near.

```
<tipo> near <definicion-puntero> ;
       o
<tipo> near <definicion-funcion>
```

La primera versión de near declara un puntero de tamaño una palabra (2 bytes) con un rango de 64K. Este modificador de tipo es usado normalmente cuando se compila en los modelos de memoria medium, large y huge para forzar a que los punteros sean near (ya que por defecto, son far, en los modelos de memoria mencionados).

Ejemplo
=======

```
char near *s;
int (near *pi)[10];
```

Cuando near es usado con una declaración de función, Turbo C genera código de función para una llamada near y devuelve near.

Ejemplo
=======

```
int near mi_funcion () {}
```

---
far (modificador de tipo)
==========

Fuerza a los punteros a ser far (estar lejos, cualquier segmento), genera código de funciones para llamadas far y devuelve un valor far.

```
<tipo> far <definicion-puntero> ;
       o
<tipo> far <definicion-funcion>
```

La primera versión de far declara un puntero de tamaño dos palabras (4 bytes) con un rango de 1 megabyte. Este modificador de tipo es usado normalmente cuando se compila en los modelos de memoria tiny, small y compact para forzar a que los punteros sean far (ya que por defecto, son near, en los modelos de memoria mencionados).

Ejemplo
=======

```
char far *s;
```

```
  void * far * p;
```

Cuando far es usado con una declaración de función, Turbo C genera código de función para una llamada far y devuelve far.

Ejemplo
=======

```
  int far my_func() {}
```

---

huge (modificador de tipo)

Es similar al modificador de tipo far

```
  <tipo> huge <definicion-puntero> ;
```

El modificador huge es similar al modificador far pero tiene dos características adicionales:

þ Su segmento es normalizado durante la aritmética de punteros así que las comparaciones de punteros son más precisas.

þ Los punteros huge pueden ser incrementados sin el temor de que sobrepase un segmento.

En resumen, la utilización de punteros huge es más segura que la utilización de punteros far, pero su uso es más lento porque requieren más comprobaciones.

Opciones de modelo

Las opciones de modelo especifican qué modelo de memoria se quiere usar. El modelo de memoria elegido determina el método de direccionamiento de memoria por defecto.

El modelo por defecto es small.

Modelo	Segmentos: Código	Segmentos: Datos	Segmentos: Pila	Punteros: Código	Punteros: Datos
Tiny	64 K			near	near
Small	64 K	64 K		near	near
Medium	1 MB	64 K		far	near
Compact	64 K	1 MB		near	far
Large	1 MB	1 MB		far	far
Huge	1 MB	64 K cada uno	64 K pila	far	far

Tiny

Usa el modelo tiny (muy pequeño) cuando la cantidad de memoria a consumir es muy importante y ha de ser la menor posible.

Los cuatro registros de segmentos (CS, DS, ES, SS) toman la misma dirección, de este modo tenemos un total de 64K para código, datos y pila. Siempre se usan punteros near.

Los programas de modelo tiny pueden ser convertidos a formato .COM.

```
-------------
|   Small   |
-------------
```

Usa el modelo small (pequeño) para aplicaciones de tamaño medio.

Los segmentos de código y datos son diferentes, así que tenemos 64K de código y 64K de datos y pila.

Siempre son usados los punteros near.

```
------------
|  Medium  |
------------
```

El modelo medium (medio) es mejor para grandes programas que no guardan muchos datos en memoria.

Los punteros far son usados para código pero no para datos. Esto da como resultado que los datos y la pila juntos están limitados a 64K, pero el código puede ocupar hasta 1MB.

```
-----------
| Compact |
-----------
```

Usa el modelo compact (compacto) si tu código es pequeño pero necesitas direccionar una gran cantidad de datos.

El modelo compact es el contrario al modelo medium: los punteros far son usados para los datos pero no para el código; el código está limitado entonces a 64K, mientras que los datos tienen un rango de 1Mb.

Todas las funciones son near por defecto y todos los punteros de datos son far por defecto.

```
-------------
|   Large   |
-------------
```

Usa el modelo large (grande) para aplicaciones muy grandes solamente.

Los punteros far son usados para código y datos, dando un rango de 1Mb para ambos. Todas las funciones y punteros de datos son far por defecto.

```
--------------
|    Huge    |
--------------
```

Usa el modelo huge (muy grande) para aplicaciones muy grandes solamente. Los

punteros far son usados para el código y los datos.

Turbo C limita normalmente el tamaño de los datos a 64K; el modelo de memoria huge aparta ese límite permitiendo a los datos ocupar más de 64K.

El modelo huge permite múltiples segmentos de datos (cada uno de 64K de tamaño), hasta 1MB para el código, y 64K para la pila.

Todas las funciones y punteros de datos se asumen como far.

LECCIÓN 8

INTRODUCCION A LA LECCION 8

El objetivo de esta lección es hacer un estudio completo en todo lo referente a la declaración, utilización e inicialización de tipos compuestos de datos y explicar cómo se pueden crear nuevos tipos de datos a partir de los ya existentes.

El lenguaje C proporciona cinco maneras diferentes de crear tipos de datos:

- Estructuras (**struct**).
- Campos de bits.
- Uniones (**union**).
- Enumeraciones (**enum**).
- Tipos definidos por el usuario (**typedef**).

Al terminar la lección se presenta una tabla de precedencia de todos los operadores del lenguaje C.

ESTRUCTURAS

En el lenguaje C una estructura es un conjunto de variables que se referencia bajo un mismo nombre, proporcionando un medio conveniente de mantener junta información relacionada.

Las estructuras se denominan registros en otros lenguajes; por ejemplo, en Pascal.

A los elementos de la estructura se les suele llamar miembros o campos.

DECLARACION DE ESTRUCTURAS

Una estructura se declara con la palabra **struct** seguida de

una lista de declaraciones en-
cerradas entre llaves. Ejemplo:

```
  struct
  {
     int dia;
     int mes;
     int anio;
  };
```

Después de la palabra clave struct va op-
cionalmente el nombre de la estructura y
se puede emplear en declaraciones poste-
riores como una abreviatura de la entrada.

```
  struct fecha
    {
      int dia;
      int mes;
      int anio;
    };
```

En los ejemplos anteriores no se han declarado realmente variables,
sino que sólo se han definido la forma de los datos.
Para declarar una variable de un
tipo estructura sin nombre se hace:

```
  struct
    {
      int dia, mes, anio;
    } fecha_creacion;
```

Si la estructura tiene nombre se
puede hacer de la siguiente forma:

```
struct fecha { int dia, mes, anio; };
struct fecha fecha_creacion;
```

De la misma forma se pueden declarar
varias variables separadas por comas:

```
  struct
    {
      int dia, mes, anio;
    } fecha1, fecha2, fecha3;
```

o bien:

```
struct fecha { int dia, mes, anio; };
struct fecha fecha1, fecha2, fecha3;
```

También se puede crear un tipo
estructura y una variable de tipo
estructura al mismo tiempo:

```
  struct
    {
      int dia, mes, anio;
    } fecha1, fecha2, fecha3;
```

La forma general de definición de un estructura es:

```
  struct nombre_tipo_estructura
    {
      tipo_1 nombre_variable_1;
      tipo_2 nombre_variable_2;
      ...
      tipo_n nombre_variable_n;
    } nombres_variables_estructura;
```

donde tanto los nombres nombre_tipo_estructura como
nombres_variables_estructura pueden omitirse.

INICIALIZACION DE ESTRUCTURAS

Una estructura se inicializa añadiendo a su definición la lista de inicializaciones de las componentes.

```
struct fecha fecha_creacion = { 22, 7, 90 };
```

TAMA¥O DE UNA ESTRUCTURA

El tamaño en bytes de un tipo estructura en memoria es la suma de los tamaños en bytes de cada uno de los tipos de sus componentes:

```
sizeof (struct fecha) == sizeof (int) + sizeof (int) + sizef (int)
```

REFERENCIA A LOS ELEMENTOS DE LA ESTRUCTURA

Los elementos individuales de la estructura se referencia utilizando el operador punto (.). La forma general es:

```
  fecha_creacion.dia = 3;
  printf ("%d", fecha_creacion.dia); /* muestra el valor 3 */
```

Un elemento de una estructura, el nombre de una estructura y una variable ordinaria pueden tener el mismo nombre; siempre se pueden distinguir a través del contexto. Por supuesto que estas cosas se deben evitar en lo posible si disminuyen la legibilidad del código. Por ejemplo:

```
  struct x { int x; } x;
```

ALGUNOS TIPOS USUALES QUE SE PUEDEN HACER CON LAS ESTRUCTURAS

ARRAYS DE ESTRUCTURAS
==========

Declaración:

```
  struct x
    {
      char *s;
      int d;
    } y[] =
        {
          "s1", 4,
          "s2", 3,
          "s3", 10
        };
```

o también

```
struct x
  {
     char *s;
     int d;
  } y[] =
      {
        { "s1", 4 },
        { "s2", 3 },
        { "s3", 10 }
       };
```

Utilización:

```
   printf ("%s %d", y[1].s, y1[1].d); /* imprime s2 3 */
```

ARRAYS DENTRO DE ESTRUCTURAS

Declaración:

```
  struct x
    {
      int a[10];
      float b;
    } y;
```

Utilización:

```
  y.a[1] = 2;
```

ESTRUCTURAS DENTRO DE ESTRUCTURAS (ESTRUTURAS ANIDADAS)

Declaración:

```
  struct persona
    {
      char nombre[TAMNOMBRE], direccion[TAMDIRECCION];
      struct fecha fecha_nacimiento;
    } empleado;
```

Utilización:

```
  empleado.fecha_nacimiento.dia = 10;
```

PUNTEROS A ESTRUCTURAS

Declaración:

```
  struct fecha fecha_creacion, *pfecha;
  pfecha = &fecha_creacion;
```

Utilización:

Hay dos formas equivalentes de acceder a los elementos de una estructura mediante un puntero a la estructura:

1) `(*pfecha).dia = 20;`

   Los paréntesis son necesarios porque el operador . tiene más prioridad que el operador *.

2) `pfecha->dia = 20;`

   El operador -> se utiliza para acceder a un elemento de una estructura a través de un puntero.

   Esta forma es mucho más común que la anterior, es más, el primer método se considera arcaico en los estándares actuales.

Debido a que este tipo de operación es tan común, C proporciona un operador específico para realizar esta tarea: el operador flecha (->).

ESTRUCTURAS Y FUNCIONES

Con respecto a las estructuras, a una función le podemos pasar como argumento:

- Un elemento de una estructura.
- Una estructura completa.
- La dirección de una estructura.

PASO DE ELEMENTOS A ESTRUCTURAS A FUNCIONES

Cuando se pasa un elemento de una variable de estructura a una función, se está realmente pasando el valor de ese elemento a la función. Además, si ese elemento es de tipo simple, se está pasando una variable simple.

Ejemplo:

```
~struct x
~  {
~    int i;
~    char c;
~    char s[10];
~  } y = { 2, 'd', "string" };
~
~/* pasando argumentos por valor: */
~
~func1 (y.i);    /* se pasa variable entera (2) */
~func2 (y.c);    /* se pasa variable carácter ('d') */
~func3 (y.s);    /* se pasa variable string (dirección de la cadena s) */
~func4 (y.s[1]); /* se pasa variable carácter ('t') */
~
~/* pasando argumentos por referencia: */
~
~func5 (&y.i);   /* se pasa la dirección de la variable entera y.i */
~func6 (&y.c);   /* se pasa la dirección de la variable carácter y.c */
~func7 (y.s);    /* se pasa la dirección del string y.s */
~func8 (&y.s[1]);/* se pasa la dirección de la variable carácter y.s[1] */
```

Observar que el operador & precede al nombre de la estructura y no al nombre del elemento individual.

PASO DE ESTRUCTURAS A FUNCIONES

Cuando se utiliza una estructura como argumento de una función, se pasa la estructura íntegra mediante el uso del método estándar de llamada por valor. Esto significa, por supuesto, que todos los cambios realizados en los contenidos de la estructura, dentro de la función a la que se pasa, no afectan a la estructura utilizada como argumento.

Ejemplo 1: Estructura sin nombre.

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

```
~
~void f (); /* los paréntesis vacíos quieren decir que no decimos
~              nada acerca de sus parámetros */
~
~void main (void)
~{
~  struct { int a, b; } arg;
~  arg.a = 10;
~  f (arg); /* imprime 10 */
~}
~
~void f (param)
~struct { int x, y; } param;
~{
~  printf ("%d", param.x);
~}
```

Ejemplo 2: Estructura con nombre.

```
~#include <stdio.h>
~
~struct st { int a, b; }; /* si se declarase dentro de la función main, esta
~                            estructura sólo se conocería en esa función */
~
~void f (); /* los paréntesis vacíos quieren decir que no decimos
~              nada acerca de sus parámetros */
~
~void main (void)
~{
~  struct st arg;
~  arg.a = 10;
~  f (arg); /* imprime 10 */
~}
~
~void f (param)
~struct st param;
~{
~  printf ("%d", param.a);
~}
```

PASO DE DIRECCIONES DE ESTRUCTURAS A FUNCIONES

En la práctica nunca se pasan las estructuras completas a funciones porque ese procedimiento es tremendamente ineficiente (en cuanto a tiempo y memoria consumida). Lo que se hace en lugar de ello es pasar la dirección de la variable estructura como argumento y declarar el parámetro de la función como un puntero a esta estructura.

Ejemplo: Paso de dirección de variable estructura.

```
~#include <stdio.h>
~
~struct st { int a, b; }; /* si se declarase dentro de la función main, esta
~                            estructura sólo se conocería en esa función */
~
~void f (); /* los paréntesis vacíos quieren decir que no decimos
~              nada acerca de sus parámetros */
~
~void main (void)
~{
~  struct st arg;
~  arg.a = 10;
```

```
~  f (&arg);  /* imprime 10 */
~}
~
~void f (param)
~struct st *param;
~{
~  printf ("%d", param->a);
~}
```

Si consideramos que sizeof(struct st *)==2 y sizeof(int)==2, entonces sizeof(struct st)==4. Por lo tanto, al pasar la estructura íntegra pasamos 4 bytes a la función y al pasar el puntero a la estructura sólo pasamos 2 bytes a la función. En el ejemplo visto la diferencia no parece muy significativa, pero si sizeof(struct st)==1000, al pasar la estructura íntegra pasamos 1000 bytes mientras que con su dirección pasamos 2 bytes.

Para los usuarios que tengan la opción turbo on, recordarán que los punteros near ocupan 2 bytes (sólo contienen el desplazamiento dentro del segmento de memoria) y los punteros far 4 bytes (contienen segmento y desplazamiento dentro del segmento). Al suponer en el párrafo anterior que sizeof(struct st *)==2, estamos suponiendo que el puntero es near.

CAMPO DE BITS

A difererencia de la mayoría de los lenguajes de computadoras, el lenguaje C tiene un método incorporado para acceder a un bit individual dentro de un byte. Esto se puede hacer con los operadores de bits que vimos en la lección 2 pero también se puede hacer con un tipo especia de estructura llamada campo de bits.

DECLARACION

La forma general de definición de un campo de bits es:

```
struct nombre_estructura
  {
    tipo1 nombre1 : longitud1;
    tipo2 nombre2 : longitud2;
    ...
  };
```

Un campo de bits tiene que declararse como int, unsigned, o signed. Los campos de bits de longitud 1 deben declararse como unsigned debido a que un bit individual no puede tener signo. Los nombres de los campos son opcionales.

EJEMPLO

```
struct operacion
  {
    unsigned leer: 1;
    unsigned escribir: 1;
    unsigned abrir: 1;
    unsigned cerrar: 1;
    unsigned: 2;
    unsigned error: 1;
  } operacion_fichero;
```

A los campos de bits se accede de la misma forma que a los campos de cualquier estructura:

```
operacion_fichero.abrir = 1;
```

Los campos de bits tienen algunas restricciones: no se puede tomar la dirección de una variable de campos de bits; en algunas máquinas, los campos se disponen de izquierda a derecha y en otras de derecha a izquierda, esto implica código menos portable.
Se puede mezclar elementos normales de estructura con elementos de campos de bits. Por ejemplo:

```
  struct st
    {
      char ch;
      unsigned c1: 1;
      unsigned c2: 3;
    };
```

El tamaño de esta estructura, sizeof (struct st), es 2: 1 byte para ch y 1 byte para c1 y c2.

UNIONES

Las uniones son similares a las estructuras. Su declaración y utilización es exactamente igual que el de las estructuras con la excepción que utiliza la palabra clave union en vez de struct.

La diferencia que hay entre una variable estructura y una variable unión es que es esta última todos sus elementos ocupan la misma posición de memoria.
Ejemplo:

```
  #include <stdio.h>
  void main (void)
  {
    union { int x, y; } u;
    u.x = 10; /* también estamos haciendo u.y = 10, puesto
       que x e y comparten la misma posición de memoria */
    printf ("%d", u.y); /* imprime 10 */
  }
```

El tamaño del tipo union es el tamaño del elemento que necesita más memoria.

Ejemplo:

```
#include <stdio.h>
void main (void)
{
  union
    {
      char ch;
      int i;
    } u;
  u.i = 257;
  printf ("%d %d", u.ch, (char) u.i); /* imprime:  1 1  */
}
```

ENUMERACIONES

Una enumeración es un conjunto de constantes enteras
con nombre que especifica todos los valores válidos
que una variable de ese tipo puede tomar.
La forma general de declaración es:

enum nombre_tipo_enum { lista_de_enumeracion } lista_variables;

Aquí, al igual que con las estructuras, tanto el nombre de la enumeración nombre_tipo_enum como lista_variables son opcionales.
Ejemplo:

```
enum colores { azul, rojo, amarillo };
enum colores color;
```

Dada esta declaración y definición la variable color sólo puede tomar tres valores posibles: azul, rojo o amarillo.

Ejemplo:

```
color = rojo;
if (color == azul)
  printf ("el color es azul");
```

Cada uno de los símblos del conjunto de enumeración es un valor entero, tomando el primero el valor 0, el segundo el valor 1 y así sucesivamente.

Ejemplo:

```
printf ("%d %d %d", azul, rojo, amarillo); /* muestra  0 1 2  */
```

Podríamos haber dado otros valores numéricos a los símbolos si hubiésemos declarado colores, por ejemplo, del siguiente modo:

```
enum colores { azul, rojo = 10, amarillo };
```

Ahora la sentencia printf anterior mostraría 0 10 11 .

Como se ve, los símbolos no inicializados toman el valor numérico siguiente al del símbolo anterior, excepto el primero que toma el valor 0 si no es inicializado.

TIPOS DEFINIDOS POR EL USUARIO: typedef

El programador puede crear (dar nombre en realidad) tipos a partir

de los tipos ya definidos mediante la palabra clave **typedef**.
La forma general de la sentencia typedef es:

typedef tipo nombre;

donde tipo es cualquier tipo de datos permitido
y nombre es el nuevo nombre para ese tipo.
El uso de typedef hace más fácil de leer el código y más fácil
de transportar a una nueva máquina. Pero recuerda: NO se crea
ningún tipo de datos nuevo, sólo se da otro nombre.

Ejemplos:

```
void main (void)
{
   typedef int boolean;
   boolean b; /* equivalente a: int b; */

   typedef int vector[100];
   vector v; /* equivalente a: int v[100]; */

   typedef char *pc;
   pc string; /* equivalente a: char *string; */

   typedef void (*pf) (void);
   pf f; /* equivalente a: void (*f) (void); */

   typedef struct { int x, y; } st;
   st estructura1;
               /* equivalente a: struct { int x, y; } estructura; */

   struct st { int x, y; };
   typedef struct st s;
   s estructura2; /* equivalente a: struct st estructura; */

   typedef int entero;
   typedef const entero * const puntero_constante_a_constante_entera;
   puntero_constante_a_constante_entera pcace;
                        /* equivalente a: const int * const pcace; */
}
```

PRECEDENCIA DE OPERADORES

Con los operadores punto (.) y flecha (->) hemos completado el estudio
de todos los operadores de C. A continuación mostramos la tabla completa
de precedencia de operadores.

Precedencia de Operadores
==========

En la siguiente tabla de precedencia de operadores, los operadores son
divididos en 15 categorías.

La categoría #1 tiene la precedencia más alta; la categoría #2 (operadores
unarios) toma la segunda precedencia, y así hasta el operador coma, el
cual tiene la precedencia más baja.

Los operadores que están dentro de una misma categoría tienen igual
precedencia.

Los operadores unarios (categoría #2), condicional (categoría #13), y
de asignación (categoría #14) se asocian de derecha a izquierda; todos

los demás operadores se asocian de izquierda a derecha.

#	Categoría	Operador	Qué es (o hace)
1.	Más alto	()	Llamada a función
		[]	Indexamiento de array
		->	Selector de componente indirecta
		.	Selector de componente directa
2.	Unario	!	Negación Lógica (NO)
		~	Complemento a 1
		+	Más unario
		-	Menos unario
		++	Preincremento o postincremento
		--	Predecremento o postdecremento
		&	Dirección
		*	Contenido (indirección)
		sizeof	(devuelve tamaño de operando, en bytes)
3.	Multiplicativo	*	Multiplica
		/	Divide
		%	Resto (módulo)
4.	Aditivo	+	Más binario
		-	Menos binario
5.	Desplazamiento	<<	Desplazamiento a la izquierda
		>>	Desplazamiento a la derecha
6.	Relacional	<	Menor que
		<=	Menor o igual que
		>	Mayor que
		>=	Mayor o igual que
7.	Igualdad	==	Igual a
		!=	Distinto a
8.		&	AND entre bits
9.		^	XOR entre bits
10.		|	OR entre bits
11.		&&	AND lógico
12.		||	OR lógico
13.	Condicional	?:	(a ? x : y significa "si a entonces x, si no y")
14.	Asignación	=	Asignación simple
		*=	Asignar producto
		/=	Asignar cociente
		%=	Asignar resto (módulo)
		+=	Asignar suma
		-=	Asignar diferncia
		&=	Asignar AND entre bits
		^=	Asignar XOR entre bits
		|=	Asignar OR entre bits
		<<=	Asignar desplazamiento hacia la izquierda
		>>=	Asignar desplazamiento hacia la derecha
15.	Coma	,	Evaluar

LECCIÓN 9

INTRODUCCION A LA LECCION 9

El objetivo de esta lección es hacer un estudio completo en todo lo referente a la entrada y, salida (E/S) en C, estudiando también los dos sistemas de ficheros que existen en este lenguaje.

Los puntos que se estudian en esta lección son:
- E/S estándar.
- Flujos y ficheros.
- Tipos de flujos: flujos de texto y flujos binarios.
- Flujos predefinidos: **stdin**, **stdout** y **stderr**.
  (Turbo C añade además: **stdaux** y **stdprn**)

- Pasos a realizar para manipular un fichero: declarar variable de fichero, abrirlo, operar con él y cerrarlo.
- Estructura **FILE**.
- Estudio completo del fichero **stdio.h**.
  (tanto del ANSI C como de Turbo C)

- Sistema de ficheros tipo UNIX.
- Descriptores de ficheros.
- Estudio completo del fichero **io.h** en Turbo C.

ENTRADA Y SALIDA

Las operaciones de entrada y salida (abreviadamente **E/S**) no forman parte del lenguaje C propiamente dicho, sino que están en una biblioteca o librería: **<stdio.h>**. Todo programa que utilice funciones de entrada y salida estándar deberá contener la línea:

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

E/S estándar

Por defecto, la entrada estándar es el teclado y la salida estándar es la pantalla o monitor. Hay dos formas básicas de cambiar la entrada y la salida estándar:

1. Con los símbolos de redirección (<, >, <<, >>) o de tubería (|) del sistema operativo al ejecutar el programa desde la línea de órdenes.

2. Con determinadas funciones y variables que se encuentran en la librería <stdio.h> en el código

fuente del programa.

FLUJOS Y FICHEROS

Hay dos conceptos muy importantes en C relacionados con la E/S: flujos (streams, en inglés) y ficheros. Los **flujos** son sucesiones de caracteres a través de los cuales realizamos las operaciones de E/S. Para el programador todos los flujos son iguales. Para el C (en general para el sistema operativo) un **fichero** es un concepto lógico que puede ser aplicado a cualquier cosa desde ficheros de discos a terminales. A cada fichero se le asigna un flujo al realizar la operación de apertura sobre él. Para el programador un fichero es un dispositivo externo capaz de una E/S. Todos los ficheros no son iguales pero todos los flujos sí. Esto supone una gran simplificación para el usuario, ya que sólo tiene que pensar en términos de flujo y no de dispositivos concretos. Por ejemplo, si el usuario hace: printf ("mensaje"); sabe que mensaje se escribe en el flujo estándar de salida, ya sea la pantalla, un fichero de disco, una cinta, ...

TIPOS DE FLUJOS

Cuando hemos dicho que todos los flujos son iguales, es cierto que lo son en su utilización por parte del programador, pero en realidad, podemos distinguir dos tipos:

- **Flujos de texto**: son una sucesión de caracteres originado en líneas que finalizan con un carácter de nueva-línea. En estos flujos puede no haber una relación de uno a uno entre los caracteres que son escritos (leídos) y los del dispositivo externo, por ejemplo, una nueva-línea puede transformarse en un par de caracteres (un retorno de carro y un carácter de salto de línea).
- **Flujos binarios**: son flujos de bytes que tienen una correspondencia uno a uno con los que están almacenados en el dispositivo externo. Esto es, no se presentan desplazamientos de caracteres. Además el número de bytes escritos (leídos) es el mismo que los almacenados en el dispositivo externo.

Esta diferencia de flujos es importante tenerla en cuenta al leer ficheros de discos. Supongamos que tenemos un fichero de disco con 7 caracteres donde el cuarto carácter es el carácter fin de fichero (en sistema operativo DOS es el carácter con código ASCII 26). Si abrimos el fichero en modo texto, sólo podemos leer los 3 primeros caracteres, sin embargo, si lo abrimos en modo binario, leeremos los 7 caracteres ya que el carácter con código ASCII 26 es un carácter como cualquier otro.

PROGRAMAS C CON FLUJOS

Al principio de la ejecución de un programa C se abren tres flujos de tipo texto predefinidos:

stdin : dispositivo de entrada estándar
stdout: dispositivo de salida estándar
stderr: dispositivo de salidad de error estándar

Al finalizar el programa, bien volviendo de la función main al sistema operativo o bien por una llamada a exit(), todos los ficheros se cierran automáticamente. No se cerrarán si el programa termina a través de una llamada a abort() o abortando el programa.

Estos tres ficheros no pueden abrirse ni cerrarse explicítamente.

En Turbo C, además de abrirse los tres flujos anteriores se abren otros dos flujos de texto predefinidos:

stdaux: dispositivo auxiliar estándar
stdprn: impresora estándar

RESUMEN DE TODO LO DICHO HASTA EL MOMENTO EN ESTA LECCION

Como todo lo que acabamos de decir puede resultar un poco confuso a las personas que tienen poca experiencia en C, vamos a hacer un pequeño resumen en términos generales:

1. En C, cualquier cosa externa de la que podemos leer o en la que podemos escribir datos es un fichero.

2. El programador escribe (lee) datos en estos ficheros a través de los flujos de cada fichero. De esta forma el programador escribe (lee) los datos de la misma forma en todos los tipos de ficheros independientemente del tipo de fichero que sea.

3. Aunque conceptualmente todos los flujos son iguales, en realidad hay dos tipos: flujos de texto y flujos binarios.

4. Hay tres flujos de texto predefinidos que se abren automáticamente al principio del programa: stdin, stdout y stderr. Estos tres flujos se cierran automáticamente al final del programa.

PASOS PARA OPERAR CON UN FICHERO

Los pasos a realizar para realizar operaciones con un fichero son los siguientes:

1) Crear un nombre interno de fichero. Esto se hace en C declarando un puntero de fichero (o puntero a fichero). Un puntero de fichero es una variable puntero que apunta a una estructura llamada FILE. Esta estructura está definida en el fichero stdio.h y contiene toda la información necesaria para poder trabajar con un fichero. El contenido de esta estructura es dependiente de la implementación de C y del sistema, y no es interesante saberlo para el programador.

Ejemplo:

```
FILE *pf; /* pf es un puntero de fichero */
```

2) Abrir el fichero. Esto se hace con la función fopen() cuyo prototipo se encuentra en el fichero stdio.h y es:

```
FILE *fopen (char *nombre_fichero, char *modo);
```

Si el fichero con nombre nombre_fichero no se puede abrir devuelve NULL.

El parámetro nombre_fichero puede contener la ruta completa de fichero pero teniendo en cuenta que la barra invertida (\) hay que repetirla en

una cadena de caracteres.

Los valores válidos para el parámeto modo son:

Modo	Interpretación
"r"	Abrir un fichero texto para lectura
"w"	Crear un fichero texto para escritura
"a"	Añadir a un fichero texto
"rb"	Abrir un fichero binario para lectura
"wb"	Crear un fichero binario para escritura
"ab"	Añadir a un fichero binario
"r+"	Abrir un fichero texto para lectura/escritura
"w+"	Crear un fichero texto para lectura/escritura
"a+"	Abrir un fichero texto para lectura/escritura
"rb+"	Abrir un fichero binario para lectura/escritura
"wb+"	Crear un fichero binario para lectura/escritura
"ab+"	Abrir un fichero binario para lectura/escritura

Si se utiliza fopen() para abrir un fichero para escritura, entonces cualquier fichero que exista con ese nombre es borrado y se comienza con un fichero nuevo. Si no existe un fichero con ese nombre, entonces se crea uno. Si lo que se quiere es añadir al final del fichero, se debe utilizar el modo "a". Si no existe el fichero, devuelve error. Abrir un fichero para operaciones de lectura necesita que el fichero exista. Si no existe devuelve error. Finalmente, si se abre un fichero para operaciones de lectura/escritura, no se borra en caso de existir. Sin embargo, si no existe se crea.

Ejemplo:

```
FILE *pf;
pf = fopen ("c:\\autoexec.bat", "r");
if (pf == NULL) /* siempre se debe hacer esta comprobación */
  {
    puts ("No se puede abrir fichero.");
    exit (1);
  }
```

3) Realizar las operaciones deseadas con el fichero como pueden ser la escritura en él y la lectura de él. Las funciones que disponemos para hacer esto las veremos un poco más adelante.

4) Cerrar el fichero. Aunque el C cierra automáticamente todos los ficheros abiertos al terminar el programa, es muy aconsejable cerrarlos explícitamente. Esto se hace con la función fclose() cuyo prototipo es:

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

La función fclose() cierra el fichero asociado con el flujo pf y vuelca su buffer.

Si fclose() se ejecuta correctamente devuelve el valor 0. La comproba ción del valor devuelto no se hace muchas veces porque no suele fallar.

Ejemplo:

```
FILE *pf;
if ((pf = fopen ("prueba", "rb")) == NULL)
  {
    puts ("Error al intentar abrir el fichero.");
    exit (1);
  }
/* ... */
```

```
  if (fclose (pf) != 0)
    {
      puts ("Error al intentar cerrar el fichero.");
      exit (1);
    }
```

RESUMEN DE LOS PASOS PARA MANIPULAR UN FICHERO

Resumen de los 4 pasos anteriores para la manipulación de un fichero:

1) Declarar un puntero de fichero.

```
  FILE *pf;
```

2) Abrirlo el fichero.

```
  if ((pf = fopen ("nombre_fichero", "modo_apertura")) == NULL)
    error ();
  else
    /* ... */
```

3) Realizar las operaciones deseadas con el fichero.

```
  /* En las siguientes ventanas veremos las
     funciones que tenemos para ello. */
```

4) Cerrar el fichero.

```
  if (fclose (pf) != 0)
    error ();
  else
    /* ... */
```

FUNCIONES DEL ANSI C EN FICHERO DE CABECERA STDIO.H

GLOSARIO:

```
--------
  fclose    Cierra un flujo.
========
------
  feof    Macro que devuelve un valor distinto de cero si se se ha detectado
======   el fin de fichero en un flujo.
--------
  ferror    Macro que devuelve un valor distinto de cero si ha ocurrido algún
========   error en el flujo.
--------
  fflush    Vuelca un flujo.
========
-------
  fgetc    Obtiene un carácter de un flujo.
=======
----------
  fgetchar    Obtiene un carácter de stdin.
==========
-------
  fgets    Obtiene una cadena de caracteres de un flujo.
=======
-------
  fopen    Abre un flujo.
=======
---------
```

fprintf	Envía salida formateada a un flujo.
fputc	Escribe un carácter en un flujo.
fputchar	Escribe un carácter en stdout.
fputs	Escribe una cadena de caracteres en un flujo.
fread	Lee datos de un flujo.
freopen	Asocia un nuevo fichero con un flujo abierto.
fscanf	Ejecuta entrada formateada de un flujo.
fseek	Posiciona el puntero de fichero de un flujo.
ftell	Devuelve la posición actual del puntero de fichero.
fwrite	Escribe en un flujo.
getc	Macro que obtiene un carácter de un flujo.
getchar	Macro que obtiene un carácer de stdin.
gets	Obtiene una cadena de caracteres de stdin.
perror	Mensajes de error del sistema.
printf	Escribe con formateo a stdout.
putc	Escribe un carácter en un flujo.
putchar	Escribe un carácter en stdout.
puts	Escribe un string en stdout (y añade un carácter de nueva línea).
remove	Función que borra un fichero.
rename	Renombra un fichero.
rewind	Reposiciona el puntero de fichero al comienzo del flujo.
scanf	Ejecuta entrada formateada de stdin.

```
=======
--------
  setbuf    Asigna un buffer a un flujo.
========
---------
  setvbuf    Asigna un buffer a un flujo.
=========
---------
  sprintf    Envía salida formateada a un string.
=========
--------
  sscanf    Ejecuta entrada formateada de string.
========
---------
  tmpfile    Abre un fichero temporal en modo binario.
=========
--------
  tmpnam    Crea un nombre de fichero único.
========
--------
  ungetc    Devuelve un carácter al flujo de entrada.
========
----------
  vfprintf    Envía salida formateada a un flujo usando una lista de
==========   argumentos.
---------
  vfscanf    Ejecuta entrada formateada de un flujo usando una lista de
=========   argumentos.
---------
  vprintf    Envía salida formateada a stdout usando una lista de
=========   argumentos.
--------
  vscanf    Ejecuta entrada formateada de stdin usando una lista de
========   argumentos.
----------
  vsprintf    Envía salida formateada a un string usando una lista de
==========   argumentos.
---------
  vsscanf    Ejecuta entrada formateada de un string usando una lista de
=========   argumentos.


ESTUDIO DE LAS FUNCIONES EXPUESTAS EN EL GLOSARIO:

--------
  fclose    Cierra un flujo.
========
Sintaxis:
   int fclose (FILE *flujo);

Devuelve 0 si tiene éxito; devuelve EOF si se detecta algún error.

Un error puede ocurrir por ejemplo cuando se intenta cerrar un fichero que
ha sido ya cerrado.

Ejemplo:

  FILE *pf;

  if ((pf = fopen ("prueba", "r")) == NULL)
    error ();

  /* ... */
```

```
  if (fclose (pf))
    error ();
```

---

feof — Macro que devuelve un valor distinto de cero si se se ha detectado el fin de fichero en un flujo.

Sintaxis:

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

Una vez alcanzado el final del fichero, las operaciones posteriores de lectura devuelven EOF hasta que se cambie la posición del puntero del fichero con funciones como rewind() y fseek().

La función feof() es particularmente útil cuando se trabaja con ficheros binarios porque la marca de fin de fichero es también un entero binario válido.

Ejemplo:

```
  /* supone que pf se ha abierto como fichero
     binario para operaciones de lectura */

  while (! feof (pf))
    getc (pf);
```

---

ferror — Macro que devuelve un valor distinto de cero si ha ocurrido algún error en el flujo.

Sintaxis:

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

Los indicadores de error asociados al flujo permanecen activos hasta que se cierra el fichero, se llama a rewind() o a perror().

Ejemplo:

```
  /* supone que pf apunta a un flujo abierto
     para operaciones de escritura */

  putc (informacion, pf);
  if (ferror (pf))
    error ();
```

---

fflush — Vuelca un flujo.

Sintaxis:

```
   int fflush (FILE *flujo);
```

Si el flujo está asociado a un fichero para escritura, una llamada a fflush() da lugar a que el contenido del buffer de salida se escriba en el fichero. Si flujo apunta a un fichero de entrada, entonces el contenido del buffer de entrada se vacía. En ambos casos el fichero permanece abierto.

Devuelve EOF si se detecta algún error.

Ejemplo:

```
  /* supone que pf está asociado
     con un fichero de salida */

  fwrite (buffer, sizeof (tipo_de_dato), n, pf);
  fflush (pf);
```

---
  fgetc   Obtiene un carácter de un flujo.
=======

Sintaxis:
```
   int fgetc (FILE *flujo);
```

La función fgetc() devuelve el siguiente carácter desde el flujo de entrada e incrementa el indicador de posición del fichero. El carácter se lee como un unsigned char que se convierte a entero.

Si se alcanza el final del fichero, fgetc() devuelve EOF. Recuerda que EOF es un valor entero. Por tanto, cuando trabajes con ficheros binarios debes utilizar feof() para comprobar el final del fichero. Si fgetc() encuentra un error, devuelve EOF también. En consecuencia, si trabajas con ficheros binarios debe utilizar ferror() para comprobar los errores del fichero.

Ejemplo:

```
  /* supone que pf está asociado
     con un fichero de entrada */

  while ((ch = fgetc (pf)) != EOF)
    printf ("%c", ch);
```

---
  fgetchar   Obtiene un carácter de stdin.
==========

Sintaxis:
```
   int fgetchar (void);
```

Si tiene éxito, getchar() devuelve el carácter leído, después de convertirlo a int sin extensión de signo. En caso de fin de fichero o error, devuelve EOF.

Ejemplo:

```
  ch = getchar ();
```

---
  fgets   Obtiene una cadena de caracteres de un flujo.
=======

Sintaxis:
```
   char *fgets (char *s, int n, FILE *flujo);
```

La función fgets() lee hasta n-1 caracteres desde el flujo y los sitúa en el array apuntado por s. Los caracteres se leen hasta que se recibe un carácter de nueva línea o un EOF o hasta que se llega al límite especificado. Después de leídos los caracteres, se sitúa en el array un carácter nulo inmediatamente después del último carácter leído. Se guarda un carácter de nueva línea y forma parte de s.

Si fgets() tiene éxito devuelve la dirección de s; se devuelve un puntero nulo cuando se produce un error. Ya que se devuelve un puntero nulo cuando se produce un error o cuando se alcanza el final del fichero, utiliza feof() o ferror() para identificar lo que ocurre realmente.

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de entrada */

  while (! feof (pf))
    if (fgets (s, 126, pf))
      printf ("%s", s);
```

---
  fopen   Abre un flujo.
=======

Sintaxis:
```
   FILE *fopen (const char *nombre_fichero, const char *modo_apertura);
```

Devuelve un puntero al flujo abierto si tiene éxito; en otro caso devuelve NULL.

Esta ha sido explicada más extensa en ventanas anteriores.

Ejemplo:

```
  FILE *pf;

  if ((pf = fopen ("prueba", "r")) == NULL)
    error ();
```

---
  fprintf   Envía salida formateada a un flujo.
=========

Sintaxis:
```
   int fprintf (FILE *flujo, const char *formato[, argumento, ...]);
```

Esta función es idéntica a la función printf() con la excepción que printf() escribe en la salida estándar (flujo stdout) y la función fprintf() escribe en la salida especificada (flujo indicado en su primer argumento).

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  fprintf (pf, "esto es una prueba %d %f", 5, 2.3);
```

---
  fputc   Escribe un carácter en un flujo.
=======

Sintaxis:
```
   int fputc (int c, FILE *flujo);
```

La función fputc() escribe un carácter c en el flujo especificado a partir de la posición actual del fichero y entonces incrementa el indicador de

posición del fichero. Aunque ch tradicionalmente se declare de tipo int, es convertido por fputc() en unsigned char. Puesto que todos los argumentos de tipo carácter son pasados a enteros en el momento de la llamada, se seguirán viendo variables de tipo carácter como argumentos. Si se utilizara un entero, simplemente se eliminaría el byte más significativo.

El valor devuelto por fputc() es el valor de número de carácter escrito. Si se produce un error, se devuelve EOF. Para los ficheros abiertos en operaciones binarias, EOF puede ser un carácter válido. En estos casos, para determinar si realmente se ha producido un error utiliza la función ferror().

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  fputc ('a', pf);
```

---
  fputchar   Escribe un carácter en stdout.
==========

Sintaxis:

```
   int fputchar (int c);
```

Una llamada a fputchar() es funcionalmente equivalente a fputc(c,stdout).

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  fputchar ('a');
```

---
  fputs   Escribe una cadena de caracteres en un flujo.
=======

Sintaxis:

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

La función fputs() escribe el contenido de la cadena de caracteres apuntada por s en el flujo especificado. El carácter nulo de terminación no se escribe.

La función devuelve 0 cuando tiene éxito, y un valor no nulo bajo condición de error.

Si se abre el flujo en modo texto, tienen lugar ciertas transformaciones de caracteres. Esto supone que puede ser que no haya una correspondencia uno a uno de la cadena frente al fichero. Sin embargo, si se abre en modo binario, no se producen transformaciones de caracteres y se establece una correspondencia uno a uno entre la cadena y el fichero.

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  fputs ("abc", pf);
```

```
-------
  fread    Lee datos de un flujo.
=======

Sintaxis:
   int fread (void *buf, int tam, int n, FILE *flujo);

Lee n elementos de tam bytes cada uno. Devuelve el número de elementos (no
bytes) realmente leídos. Si se han leído menos caracteres de los pedidos en
la llamada, es que se ha producido un error o es que se ha alcanzado el
final del fichero. Utiliza feof() o ferror() para determinar lo que ha
tenido lugar.

Si el flujo se abre para operaciones de texto, el flujo de retorno de carro
y salto de línea se transforma automáticamente en un carácter de nueva línea.

Ejemplo:

  /* supone que pf está asociado con
     un fichero de entrada */

  float buf[10];

  if (fread (buf, sizeof (float), 10, pf) != 10)
    if (feof (pf))
      printf ("fin de fichero inesperado");
    else
      printf ("error de lectura en el fichero");


---------
  freopen    Asocia un nuevo fichero con un flujo abierto.
=========

Sintaxis:
   FILE *freopen (const char *nombre_fichero, const char *modo, FILE *flujo);

El flujo es cerrado. El fichero nombre_fichero es abierto y asociado con el
flujo. Devuelve flujo si tiene éxito o NULL si falló.

Ejemplo:

  #include <stdio.h>

  void main (void)
  {
    FILE *pf;

    printf ("Esto se escribe en pantalla.\n");
    if ((pf = freopen ("salida.doc", "w", stdout)) == NULL)
      printf ("Error: No se puede abrir el fichero salida.doc\n");
    else
      printf ("Esto se escribe en el fichero salida.doc");
  }


--------
  fscanf    Ejecuta entrada formateada de un flujo.
========

Sintaxis:
   int fscanf (FILE *flujo, const char *formato[, direccion, ...]);

Esta función es idéntica a la función scanf() con la excepción que scanf()
```

lee de la entrada estándar (flujo stdin) y la función fscanf () lee de la entrada especificada (flujo indicado en su primer argumento).

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de entrada */

  int d;
  float f;
  fscanf (pf, "%d %f", &d, %f);
```

---

fseek Posiciona el puntero de fichero de un flujo.

Sintaxis:

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

La función fseek() sitúa el indicador de posición del fichero asociado a flujo de acuerdo con los valores de desplazamiento y origen. Su objetivo principal es soportar operaciones de E/S aleatorias; desplazamiento es el número de bytes desde el origen elegido a la posición seleccionada. El origen es 0, 1 ó 2 (0 es el principio del fichero, 1 es la posición actual y 2 es el final del fichero). El estándar ANSI fija los siguientes nombres para los orígenes:

Origen	Nombre
Comienzo del fichero	SEEK_SET
Posición actual	SEEK_CUR
Final del fichero	SEEK_END

Si se devuelve el valor 0, se supone que fseek() se ha ejecutado correctamente. Un valor distinto de 0 indica fallo.

En la mayor parte de las implementaciones y en el estándar ANSI, desplazamiento debe ser un long int para soportar ficheros de más de 64K bytes.

Ejemplo:

```
  #include <stdio.h>

  long tamanio_fichero (FILE *flujo);

  int main (void)
  {
     FILE *flujo;

     flujo = fopen ("MIFICH.TXT", "w+");
     fprintf (flujo, "Esto es un test.");
     printf ("El tamaño del fichero MIFICH.TXT es %ld bytes,\n",
             tamanio_fichero (flujo));
     fclose (flujo);
     return 0;
  }

  long tamanio_fichero (FILE *flujo)
  {
     long posicion_corriente, longitud;

     posicion_corriente = ftell (flujo);
     fseek (flujo, 0L, SEEK_END);
```

```
     longitud = ftell (flujo);
     fseek (flujo, posicion_corriente, SEEK_SET);
     return flujo;
  }
```

---
ftell Devuelve la posición actual del puntero de fichero.
=======

Sintaxis:
```
   long ftell (FILE *flujo);
```

Devuelve el valor actual del indicador de posición del fichero para el flujo especificado si tiene éxito o -1L en caso de error.

Ejemplo:

```
  /* ver ejemplo de la función fseek */
```

---
fwrite Escribe en un flujo.
========

Sintaxis:
```
   int fwrite (const void *buf, int tam, int n, FILE *flujo);
```

Escribe n elementos de tam bytes cada uno. Devuelve el número de elementos (no bytes) escritos realmente.

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  float f = 1.2;

  fwrite (&f, sizeof (float), 1, pf);
```

---
getc Macro que obtiene un carácter de un flujo.
======

Sintaxis:
```
   int getc (FILE *flujo);
```

Devuelve el carácter leído en caso de éxito o EOF en caso de error o que se detecte el fin de fichero.

Las funciones getc() y fgetc() son idénticas, y en la mayor parte de las implementaciones getc() está definida por la siguiente macro:

```
#define getc(pf) fgetc(pf)
```

Ejemplo:

```
  /* supone que pf está asociado
     con un fichero de entrada */

  while ((ch = getc (pf)) != EOF)
    printf ("%c", ch);
```

----
  getchar   Macro que obtiene un carácer de stdin.
=========

Sintaxis:
   int getchar (void);

Si tiene éxito, getchar() devuelve el carácter leído, después de convertirlo a int sin extensión de signo. En caso de fin de fichero o error, devuelve EOF.

Las funciones getchar() y fgetchar() son idénticas, y en la mayor parte de las implementaciones getchar() está simplemente definida como la siguiente macro:

#define getchar(pf) fgetchar(pf)

Ejemplo:

```
  ch = getchar ();
```

------
  gets   Obtiene una cadena de caracteres de stdin.
======

Sintaxis:
   char *gets (char *string);

Lee caracteres de stdin hasta que un carácter de nueva línea (\n) es encontrado. El carácter \n no es colocado en el string. Devuelve un puntero al argumento string.

Ejemplo:

```
  char nombre_fichero[128];

  gets (nombre_fichero);
```

----
  perror   Mensajes de error del sistema.
========

Sintaxis:
   void perror (const char *s);

Imprime un mensaje de error en stderr. Primero se imprime el argumento de string s, después se escriben dos puntos, a continuación se escribe un mensaje de error acorde al valor corriente de la variable errno, y por último se escribe una nueva línea.

errno es una variable global que contiene el tipo de error. Siempre que ocurre un error en una llamada al sistema, a errno se le asigna un valor que indica el tipo de error.

En Turbo C, la variable errno está declarada en los ficheros errno.h, stddef.h y stdlib.h, siendo su declaración: int errno;
Los posibles valores que puede tomar esta variable no interesa en este momento, así que se dirán cuáles son cuando se estudie la librería <errno.h>.

Ejemplo:

```
  /*
    Este programa imprime:
     No es posible abir fichero para lectura: No such file or directory
  */

  #include <stdio.h>

  int main (void)
  {
    FILE *fp;

    fp = fopen ("perror.dat", "r");
    if (! fp)
      perror ("No es posible abir fichero para lectura");
    return 0;
  }
```

---
  printf   Escribe con formateo a stdout.
========

Sintaxis:
```
   int printf (const char *formato [, argumento, ...]);
```

Esta función se explicó completamente en la lección 4.

---
  putc   Escribe un carácter en un flujo.
======

Sintaxis:
```
   int putc (int c, FILE *flujo);
```

Si tiene éxito, putc() devuelve el carácter c. En caso de error, devuelve EOF.

Una de las funciones putc() y fputc() se implementa como macro de la otra. Las dos son funcionalmente equivalentes.

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  putc ('a', pf);
```

---
  putchar   Escribe un carácter en stdout.
=========

Sintaxis:
```
   int putchar (int c);
```

Si tiene éxito, putc() devuelve el carácter c. En caso de error, devuelve EOF.

Una de las funciones putchar() y fputchar() se implementa como macro de la otra. Las dos son funcionalmente equivalentes.

Ejemplo:

```
  /* supone que pf está asociado con
     un fichero de salida */

  putchar ('a');
```

---
  puts   Escribe un string en stdout (y añade un carácter de nueva línea).
======

Sintaxis:
```
   int puts (const char *s);
```

Si la escritura tiene éxito, puts() devuelve el último carácter escrito. En otro caso, devuelve EOF.

Esta función se discutió completamente en la lección 4.

---
  remove   Función que borra un fichero.
========

Sintaxis:
```
   int remove (const char *nombre_fichero);
```

La función remove() borra el fichero especificado por nombre_fichero. Devuelve 0 si el fichero ha sido correctamente borrado y -1 si se ha producido un error.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    char fichero[80];

    printf ("Fichero para borrar: ");
    gets (fichero);

    if (remove (fichero) == 0)
      printf ("Fichero %s borrado.\n", fichero);
    else
      printf ("No se ha podido borrar el fichero %s.\n", fichero);

    return 0;
  }
```

---
  rename   Renombra un fichero.
========

Sintaxis:
```
   int rename (const char *viejo_nombre, const char *nuevo_nombre);
```

La función rename() cambia el nombre del fichero especificado por viejo_nombre a nuevo_nombre. El nuevo_nombre no debe estar asociado a ningún otro en el directorio de entrada. La función rename() devuelve 0 si tiene éxito y un valor no nulo si se produce un error.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    char viejo_nombre[80], nuevo_nombre[80];

    printf ("Fichero a renombrar: ");
    gets (viejo_nombre);
    printf ("Nuevo nombre: ");
    gets (nuevo_nombre);

    if (rename (viejo_nombre, nuevo_nombre) == 0)
      printf ("Fichero %s renombrado a %s.\n", viejo_nombre, nuevo_nombre);
    else
      printf ("No se ha podido renombrar el fichero %s.\n", viejo_nombre);

    return 0;
  }
```

---
rewind Reposiciona el puntero de fichero al comienzo del flujo.
========

Sintaxis:

```
  void rewind (FILE *flujo);
```

La función rewind() mueve el indicador de posición del fichero al principio del flujo especificado. También inicializa los indicadores de error y fin de fichero asociados con flujo. No devuelve valor.

Ejemplo:

```
  void releer (FILE *pf)
  {
    /* lee una vez */
    while (! feof (pf))
      putchar (getc (pf));

    rewind (pf);

    /* leer otra vez */
    while (! feof (pf))
      putchar (getc (pf));
  }
```

---
scanf Ejecuta entrada formateada de stdin.
=======

Sintaxis:

```
   int scanf (const char *formato [, ...]);
```

Esta función se explicó completamente en la lección 4.

---
setbuf Asigna un buffer a un flujo.
========

Sintaxis:

```
   void setbuf (FILE *flujo, char *buf);
```

La función setbuf() se utiliza para determinar el buffer del flujo especificado que se utilizará o -si se llama con buf a nulo- para desactivar el buffer. Si un buffer va a ser definido por el programador, entonces debe ser de BUFSIZ caracteres. BUFSIZ está definido en stdio.h. La función setbuf() no devuelve valor.

Ejemplo:

```
#include <stdio.h>

/* BUFSIZ está definido en stdio.h */
char outbuf[BUFSIZ];

int main (void)
{
  /* añade un buffer al flujo de salida estándar */
  setbuf (stdout, outbuf);

  /* pone algunos caracteres dentro del buffer */
  puts ("Esto es un test de salida con buffer.\n\n");
  puts ("Esta salida irá a outbuf\n");
  puts ("y no aparecerá hasta que el buffer\n");
  puts ("esté lleno o volquemos el flujo.\n");

  /* vuelca el buffer de salida */
  fflush (stdout);

  return 0;
}
```

---

setvbuf — Asigna un buffer a un flujo.

Sintaxis:

```
int setvbuf (FILE *flujo, char *buf, int tipo, int tam);
```

La función setvbuf() permite al programador especificar el buffer, su tamaño y su modo para el flujo especificado. El array de caracteres apuntado por buf se utiliza como buffer de flujo para las operaciones de E/S. El tamaño del buffer está fijado por tam, y tipo determina como se usará. Si buf es nulo, no tiene lugar ninguna operación sobre el buffer.

Los valores legales de tipo son _IOFBF, _IONBF y _IOLBF. Están definidos en stdio.h. Cuando se activa el modo _IOFBF se produce una operación de buffer completa. Este es el modo por defecto. Cuando se activa _IONBF, el flujo no utiliza buffer independientemente del valor de buf. Si el modo es _IOLBF, el flujo utiliza buffer por líneas, lo que supone que el buffer es volcado en el fichero cada vez que se escribe un carácter de salto de línea para los flujos de salida; para los flujos de entrada lee todos los caracteres hasta un carácter de salto de línea. En cualquier caso, el buffer es volcado en el fichero cuando se llena.

El valor de tam debe ser mayor que 0. La función setvbuf() devuelve 0 en caso de éxito; en caso de fallo devuelve un valor distinto de cero.

Ejemplo:

```
setvbuf (pf, buffer, _IOLBF, 128);
```

---
  sprintf   Envía salida formateada a un string.
=========

Sintaxis:
   int sprintf (char *buffer, const char *formato [, argumento, ...]);

Esta función es igual que la función printf() con la diferencia de que la salida de la función printf() va al flujo stdout y la salida de la función sprintf() va al string buffer.

Devuelve el número de bytes escritos. En caso de error, sprintf() devuelve EOF.

Ejemplo:

```
  char cadena[80];
  sprintf (cadena, "%s %d %c", "abc", 5, 'd');
```


----
  sscanf   Ejecuta entrada formateada de string.
========

Sintaxis:
   int sscanf (const char *buffer, const char *formato [, direccion, ...]);

Esta función es igual que la función scanf() con la diferencia de que la entrada de la función scanf() se coge del flujo stdin y la entrada de la función sscanf() se coge del string buffer.

Devuelve el número de bytes escritos. En caso de error, sprintf() devuelve EOF.
Devuelve el número de campos leídos, explorados, convertidos y almacenados con éxito. Si sscanf intenta leer más allá del final de buffer, entonces el valor devuelto es EOF.

Ejemplo:

```
  char cadena[80];
  int i;
  sscanf ("abc 6", "%s%d", cadena, &i);
```


----
  tmpfile   Abre un fichero temporal en modo binario.
=========

Sintaxis:
   FILE *tmpfile (void);

La función tmpfile() abre un fichero temporal para actualizarlo y devuelve un puntero a un flujo. La función utiliza automáticamente un único nombre de fichero para evitar conflictos con los ficheros existentes. La función devuelve un puntero nulo en caso de fallo; en cualquier otro caso devuelve un puntero a un flujo.

El fichero temporal creado por tmpfile() se elimina automáticamente cuando el fichero es cerrado o cuando el programa termina.

Ejemplo:

```
  FILE *pftemp;
```

```
  if ((pftemp = tmpfile ()) == NULL)
    {
      printf ("No se puede abrir el fichero de trabajo temporal.\n");
      exit (1);
    }
```

---
  tmpnam   Crea un nombre de fichero único.
========

Sintaxis:
```
   char *tmpnam (char *nombre_fichero);
```

La función tmpnam() genera un único nombre de fichero y lo guarda en el array apuntado por nombre. El objetivo de tmpnam() es generar el nombre de un fichero temporal que sea diferente de cualquier otro que exista en el directorio.

La función puede ser llamada hasta un número de veces igual a TMP_MAX, que está definido en stdio.h. Cada vez se genera un nuevo nombre de fichero temporal.

En caso de éxito se devuelve un puntero a una cadena de caracteres; en cualquier otro caso se devuelve un puntero nulo.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    char nombre[13];

    tmpnam (nombre);
    printf ("Nombre temporal: %s\n", nombre);
    return 0;
  }
```

---
  ungetc   Devuelve un carácter al flujo de entrada.
========

Sintaxis:
```
   int ungetc (int c, FILE *flujo);
```

Prototype in:
stdio.h

La próxima llamada a getc (u otras funciones de entrada de flujos) para flujo devolverá c.

La función ungetc() devuelve el carácter c si tiene éxito. Devuelve EOF si la operación falla.

Ejemplo:

```
  ungetc ('a', stdin);
  ch = getc (stdin); /* a ch se le asigna el carácter 'a' */
```

```
  vfprintf    Envía salida formateada a un flujo usando una lista de
==========   argumentos.

Sintaxis:
   int vfprintf (FILE *fp, const char *formato, va_list lista_de_arg);

Devuelve el número de bytes escritos. En caso de error, devuelve EOF.


---------
  vfscanf    Ejecuta entrada formateada de un flujo usando una lista de
=========   argumentos.

Sintaxis:
   int vfscanf (FILE *flujo, const char *formato, va_list lista_de_arg);

Devuelve el número de campos leídos, explorados, convertidos y almacenados
con éxito.


---------
  vprintf    Envía salida formateada a stdout usando una lista de
=========   argumentos.

Sintaxis:
   int vprintf (const char *formato, va_list lista_de_arg);

Devuelve el número de bytes escritos. En caso de error, devuelve EOF.


--------
  vscanf    Ejecuta entrada formateada de stdin usando una lista de
========   argumentos.

Sintaxis:
   int vscanf (const char *formato, va_list lista_de_arg);

Devuelve el número de campos leídos, explorados, convertidos y almacenados
con éxito. Devuelve EOF en caso de que se detecte el fin de fichero.


----------
  vsprintf    Envía salida formateada a un string usando una lista de
==========   argumentos.

Sintaxis:
   int vsprintf (char *buffer, const char *formato, va_list lista_de_arg);

Devuelve el número de bytes escritos. En caso de error, devuelve EOF.


---------
  vsscanf    Ejecuta entrada formateada de un string usando una lista de
=========   argumentos.

Sintaxis:
   int vsscanf (const char *buffer, const char *formato, va_list lista_arg);

Devuelve el número de campos leídos, explorados, convertidos y almacenados
con éxito. Devuelve EOF en caso de que se detecte el fin de fichero.

Las funciones vprintf(), vfprintf() y vsprintf() son funciones equivalentes
a printf(), fprintf() y sprintf() respectivamente.
Las funciones vscanf(), vfscanf() y vsscanf() son funciones equivalentes
```

a scanf(), fscanf() y sscanf() respectivamente.
La diferencia se encuentra en que la lista de argumentos se sustituye por un puntero a una lista de argumentos. Este puntero está definido en stdarg.h. Consulta la discusión del fichero stdarg.h en la lección 5 para tener más detalles y ver un ejemplo.

FUNCIONES A¥ADIDAS POR TURBO C AL ANSI EN FICHERO DE CABECERA STDIO.H

GLOSARIO:

```
----------
  clearerr    Borra indicación de error.
==========
-----------
  fcloseall    Cierra todos los flujos abiertos.
===========
--------
  fdopen    Asocia un flujo con un descriptor de fichero.
========
---------
  fgetpos    Obtiene la posición actual del puntero de fichero.
=========
--------
  fileno   Macro que devuelve el descriptor de fichero asociado con un flujo.
========
----------
  flushall    Vuelca todos los flujos abiertos.
==========
---------
  fsetpos    Posiciona el puntero de fichero de un flujo.
=========
------
  getw    Obtiene un entero de un flujo.
======
------
  putw    Escribe un entero en un flujo.
======
```

ESTUDIO DE LAS FUNCIONES EXPUESTAS EN EL GLOSARIO:

```
----------
  clearerr    Borra indicación de error.
==========
```

Sintaxis:

```
   void clearerr (FILE *flujo);
```

La función clearerr() borra los indicadores de error y fin de fichero para el flujo especificado.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    FILE *fp;
    char ch;
```

```
    /* abre un fichero para escritura */
    fp = fopen ("PRUEBA.TXT", "w");

    /* fuerza una condición de error al intentar leer */
    ch = fgetc (fp);
    printf ("%c\n",ch);

    if (ferror( fp))
      {
        /* visualiza un mensaje de error */
        printf ("Error leyendo de PRUEBA.TXT\n");

        /* borra los indicadores de error y EOF */
        clearerr (fp);
      }

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

---
fcloseall Cierra todos los flujos abiertos.
==========

Sintaxis:

```
   int fcloseall (void);
```

Devuelve el número total de flujos cerrados, o EOF si fue detectado algún error.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    int flujos_cerrados;

    /* abre dos flujos */
    fopen ("FICHERO1", "w");
    fopen ("FICHERO2", "w");

    /* cierra los flujos abiertos */
    flujos_cerrados = fcloseall ();

    if (flujos_cerrados == EOF)
      /* imprime un mensaje de error */
      printf ("Ha ocurrido un error al intentar cerrar los ficheros.\n");
    else
       /* imprime resultado de la función fcloseall(): */
       printf("%d flujos fueron cerrados.\n", flujos_cerrados);

    return 0;
  }
```

---
fdopen Asocia un flujo con un descriptor de fichero.
========

Sintaxis:

```
   FILE *fdopen (int descriptor, char *tipo);
```

Devuelve un puntero al nuevo flujo abierto o NULL en el caso de un error.

Los valores posibles del argumento tipo son los mismos que para los de la función fopen(). Consiste en un string con una combinación de los siguientes caracteres:

Tipo	Significado
r	Abre para lectura solamente
w	Crea para escritura; sobreescribe fichero existente
a	Añade, abre para escritura al final del fichero, o crea fichero para escritura
+	Símbolo de suma para permitir operaciones de lectura/escritura
b	Abre en modo binario
t	Abre en modo texto

Las funciones de stdio.h: fileno() y fdopen(), utilizan el concepto de descriptor de fichero; este concepto se explica en las ventanas inmediatamente siguientes.

---

fgetpos — Obtiene la posición actual del puntero de fichero.

Sintaxis:

```
int fgetpos (FILE *flujo, fpos_t *pos);
```

La posición almacenada en *pos puede ser pasada a la función fsetpos() para poner la posición del puntero de fichero.

Devuelve 0 en caso de éxito y un valor distinto de cero en otro caso.

fpos_t es un tipo declarado con typdef en el fichero stdio.h que indica posición de fichero.

Ejemplo:

```
#include <string.h> /* strlen() devuelve la longitud de un string */
#include <stdio.h> /* para utilizar: FILE, fpos_t, fopen (), fwrite (),
                      fgetpos (), printf () y fclose () */

int main (void)
{
  FILE *flujo;
  char string[] = "Esto es un test";
  fpos_t posfich;

  /* abre un fichero para actualizarlo */
  flujo = fopen ("FICHERO", "w+");

  /* escribe un string en el fichero */
  fwrite (string, strlen (string), 1, flujo);

  /* informa de la posición del puntero de fichero */
  fgetpos (flujo, &posfich);
  printf ("El puntero de fichero está en el byte %ld\n", posfich);

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

----
  fileno   Macro que devuelve el descriptor de fichero asociado con un flujo.
========

Sintaxis:
  int fileno (FILE *flujo);

Las funciones de stdio.h: fileno() y fdopen(), utilizan el concepto de descriptor de fichero; este concepto se explica en las ventanas inmediatamente siguientes.

----
  flushall   Vuelca todos los flujos abiertos.
==========

Sintaxis:
   int flushall (void);

Vacía los buffers para los flujos de entradas y escribe los buffers en los ficheros para los flujos de salida.

Devuelve un entero que es el número de flujos abiertos de entrada y salida.

Ejemplo:

```
  #include <stdio.h>

  int main (void)
  {
    FILE *flujo;

    /* crea un fichero */
    flujo = fopen ("FICHERO", "w");

    /* vuelca todos los flujos abiertos */
    printf ("%d flujos fueron volcados.\n", flushall());

    /* cierra el fichero */
    fclose (flujo);
    return 0;
  }
```

----
  fsetpos   Posiciona el puntero de fichero de un flujo.
=========

Sintaxis:
   int fsetpos (FILE *flujo, const fpos_t *pos);

La nueva posición apuntada por pos es el valor obtenido por una llamada previa a la función fgetpos().

En caso de éxito, devuelve 0. En caso de fallo, devuelve un valor distinto de 0.

fpos_t es un tipo declarado con typdef en el fichero stdio.h que indica posición de fichero.

Ejemplo:

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

```
  void mostrar_posicion (FILE *flujo);

  int main (void)
  {
    FILE *flujo;
    fpos_t posicion_fichero;

    /* abre un fichero para actualización */
    flujo = fopen ("FICHERO", "w+");

    /* salva la posición del puntero de fichero */
    fgetpos (flujo, &posicion_fichero);

    /* escribe algunos datos en el fichero */
    fprintf (flujo, "Esto es una prueba");

    /* muestra la posición corriente en fichero */
    mostrar_posicion (flujo);

    /* pone una nueva posición de fichero, la visualiza */
    if (fsetpos (flujo, &posicion_fichero) == 0)
      mostrar_posicion (flujo);
    else
      {
        fprintf (stderr, "Error poniendo puntero de fichero.\n");
        exit (1);
      }

    /* cierra el fichero */
    fclose (flujo);
    return 0;
  }

  void mostrar_posicion (FILE *flujo)
  {
    fpos_t pos;

    /* visualiza la posición actual del puntero de fichero de un flujo */
    fgetpos (flujo, &pos);
    printf ("Posición de fichero: %ld\n", pos);
  }


------
  getw    Obtiene un entero de un flujo.
======

Sintaxis:
   int getw (FILE *flujo);

Devuelve el próximo entero en el flujo de entrada, o EOF si ocurre un error
o se detecta el fin de fichero. Usa las funciones feof() o ferror() para
verificar eof o error.

Ejemplo:

  int entero = getw (pf);


------
  putw    Escribe un entero en un flujo.
======
```

Sintaxis:

```
int putw (int d, FILE *flujo);
```

Devuelve el entero d. En caso de error, devuelve EOF.

Ejemplo:

```
putw (10, pf);
```

SISTEMA DE FICHEROS TIPO UNIX

Debido a que el lenguaje C se desarrolló inicialmente bajo el sistema operativo UNIX, se creó un segundo sistema de ficheros. A este sistema se le llama E/S sin buffer, E/S de bajo nivel o E/S tipo UNIX. Hoy en día, este sistema de ficheros está totalmente en desuso y se considera obsoleto, además el nuevo estándar ANSI ha decidido no estandarizar el sistema de E/S sin buffer tipo UNIX. Por todo lo dicho no se puede recomendar este sistema a los nuevos programadores C. Sin embargo, todavía existen programas que lo usan y es soportado por la mayoría de los compiladores de C. Así que incluimos una breve explicación al respecto.

DESCRIPTORES DE FICHEROS

A diferencia del sistema de E/S de alto nivel, el sistema de bajo nivel no utiliza punteros a ficheros de tipo FILE; el sistema de bajo nivel utiliza descriptores de fichero de tipo int. A cada fichero se le asocia un número (su descriptor de fichero).

Hay tres descriptores de fichero predefinidos:

0	entrada estándar
1	salida estándar
2	salida de errores estándar

Los cuatro pasos para la manipulación de ficheros con E/S de alto nivel que vimos antes se transforman en la E/S de bajo nivel en los siguientes pasos:

1) Declarar un descriptor de fichero para el fichero a abrir.

```
int fd;
```

2) Abrir el fichero.

```
if ((fd = open (nombre_fichero, modo_apertura)) = -1)
  {
    printf ("Error al intentar abrir el fichero.\n");
    exit (1);
  }
```

3) Manipular el fichero con las funciones que tenemos disponible para ello.

4) Cerrar el fichero.

```
close (fd);
```

Tanto las funciones open() como close() como todas las que tenemos disponi-

bles con relación al sistema de ficheros de bajo nivel se explican en la siguiente ventana, pero sólo para aquellos usuarios que tengan la opción de turbo puesta a on, debido a que estas funciones no pertenecen ya al estándar ANSI y por lo tanto no son portables entre distintas implementaciones del C.

FUNCIONES DE TURBO C EN FICHERO DE CABECERA IO.H

GLOSARIO:

----
  access   Determina la accesibilidad de un fichero.
========
----
  chmod   Cambia el modo de acceso.
=======

  _chmod   Cambia el modo de acceso.
========
----
  chsize   Cambia tamaño de fichero.
========

  _close/close   Cierra un descriptor de fichero.
==============

  _creat/creat   Crea un nuevo fichero o sobreescribe uno existente.
==============
----
  creatnew   Crea un nuevo fichero.
==========
----
  creattemp   Crea un fichero único en el directorio dado por el nombre de
===========   fichero.
-----
  dup   Duplica un descriptor de fichero.
=====
------
  dup2   Duplica el descriptor de fichero viejo_descriptor sobre el
======   descriptor de fichero existente nuevo_descriptor.
-----
  eof   Chequea fin de fichero.
=====
----
  filelength   Obtiene el tamaño de un fichero en bytes.
============
----
  getftime   Obtiene fecha y hora de un fichero.
==========
----
  ioctl   Controla dispositivos de I/O.
=======
----
  isatty   Chequea tipo de dispositivo.
========
------
  lock   Pone los bloqueos de compartición de ficheros para controlar el
======   acceso concurrente de ficheros.
----
  lseek   Mueve el puntero de fichero de lectura/escritura.
=======

```
  open    Abre un fichero para lectura o escritura.
======
-------
  _open    Abre un fichero para lectura o escritura.
=======
------
  read    Lee de un fichero.
======
-------
  _read    Lee de fichero.
=======
----------
  setftime    Pone fecha y hora de un fichero.
==========
---------
  setmode    Pone modo de un fichero de apertura.
=========
-------
  sopen    Macro que abre un fichero en el modo compartido.
=======
------
  tell    Obtiene la posición corriente de un puntero de fichero.
======
--------
  unlink    Borra un fichero.
========
--------
  unlock    Libera bloqueos de compartición de ficheros para controlar el
========   acceso concurrente.
-------
  write    Escribe en un fichero.
=======
--------
  _write    Escribe en un fichero.
========
----------------------
  HANDLE_MAX (#define)    Número máximo de descriptores.
----------------------
```

ESTUDIO DE LAS FUNCIONES EXPUESTAS EN EL GLOSARIO:

```
--------
  access    Determina la accesibilidad de un fichero.
========
```

Sintaxis:

```
   int access (const char *nombre_fichero, int modo_acceso);
```

þ modo_acceso = 0 chequea para existencia de fichero
þ modo_acceso = 2 chequea para permiso de escritura
þ modo_acceso = 4 chequea para permiso de lectura
þ modo_acceso = 6 chequea para permiso de lectura y escritura

Si el acceso requerido es permitido, devuelve 0; en otro caso, devuelve -1
y se asigna un valor a la variable errno.

Ejemplo:

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

int existe_fichero (char *nombre_fichero);
```

```
int main (void)
{
   printf ("%d existe NOEXISTE.FIC\n",
           existe_fichero ("NOEXISTE.FIC") ? "Sí" : "No");
   return 0;
}

int existe_fichero (char *nombre_fichero)
{
   return (access (nombre_fichero, 0) == 0);
}
```

---
chmod Cambia el modo de acceso.

Sintaxis:

```
   int chmod (const char *path, int modo_acceso);
```

Cuando tiene éxito chmod() cambia el modo de acceso al fichero y devuelve 0.
En otro caso chmod() devuelve -1.

El parámetro modo_acceso puede tomar alguno de los siguientes valores definidos en <sys\stat.h>:

S_IFMT	Máscara de tipo de fichero
S_IFDIR	Directorio
S_IFIFO	FIFO especial
S_IFCHR	Carácter especial
S_IFBLK	Bloque especial
S_IFREG	Fichero regular
S_IREAD	Poseedor puede leer
S_IWRITE	Poseedor puede escribir
S_IEXEC	Poseedor puede ejecutar

Ejemplo:

```
#include <sys\stat.h>
#include <stdio.h>
#include <io.h>

void hacer_solo_lectura (char *nombre_fichero);

int main (void)
{
   hacer_solo_lectura ("NOEXISTE.FIC");
   hacer_solo_lectura ("MIFICH.FIC");
   return 0;
}

void hacer_solo_lectura (char *nombre_fichero)
{
    int estado;

    estado = chmod (nombre_fichero, S_IREAD);
    if (estado)
      printf ("No se pudo hacer %s sólo lectura\n", nombre_fichero);
    else
      printf ("Hecho %s sólo lectura\n", nombre_fichero);
}
```

```
--------
  _chmod    Cambia el modo de acceso.
========
```

Sintaxis:

```
   int _chmod (const char *path, int func [ , int atrib ] );
```

Si func es 0, _chmod() devuelve los atributos del fichero. Si func es 1, los atributos son puestos. Si la operación tiene éxito, _chmod() devuelve la palabra de atributo del fichero; en otro caso, devuelve -1. En el caso de un error se asigna valor a errno.

El parámetro atrib representa los atributos de fichero de MS-DOS y puede tomar los siguientes valores definidos en dos.h:

```
  FA_RDONLY   Atributo de sólo lectura
  FA_HIDDEN   Fichero oculto
  FA_SYSTEM   Fichero de sistema
  FA_LABEL    Etiqueta de la unidad
  FA_DIREC    Directorio
  FA_ARCH     Archivo
```

```
--------
  chsize    Cambia tamaño de fichero.
========
```

Sintaxis:

```
   int chsize (int descriptor, long tamanio);
```

Si tiene éxito, chsize() devuelve 0. Si falla, devuelve -1 y se le da valor a errno.

Ejemplo:

```
#include <string.h> /*para utilizar strlen(): devuelve longitud de string*/
#include <fcntl.h> /* para utilizar la constante O_CREAT, ver open() */
#include <io.h> /* para utilizar write(), chsize() y close() */

int main (void)
{
   int descriptor;
   char buf[11] = "0123456789";

   /* crea fichero de texto conteniendo 10 bytes */
   descritor = open ("FICH.FIC", O_CREAT);
   write (descriptor, buf, strlen (buf));

   /* truncar el fichero a 5 bytes de tamaño */
   chsize (descriptor, 5);

   /* cierra el fichero */
   close(handle);
   return 0;
}
```

```
--------------
  _close/close    Cierra un descriptor de fichero.
==============
```

Sintaxis:

```
   int _close (int descriptor);
```

```
   int close (int descriptor);
```

Si tiene éxito, close() y _close() devuelven 0; en caso contrario, estas funciones devuelven -1 y se le da valor a errno.

Ejemplo:

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

void main (void)
{
    int descriptor;
    char buf[11] = "0123456789";

    /* crea un fichero conteniendo 10 bytes */
    descriptor = open ("NUEVO.FIC", O_CREAT);
    if (descriptor > -1)
      {
        write (descriptor, buf, strlen (buf));

        /* cierra el fichero */
        close (descriptor);
      }
    else
      {
        printf ("Error abriendo fichero\n");
      }
    return 0;
}
```

_creat/creat Crea un nuevo fichero o sobreescribe uno existente.

Sintaxis:

```
   int _creat (const char *path, int atrib);
   int creat (const char *path, int modo_acceso);
```

La función creat() abre el fichero en el modo dado por la variable global _fmode. La función _creat() siempre abre en modo binario. Si la operación tiene éxito, el descriptor del nuevo fichero es devuelto; en caso contario, un -1 es devuelto y asignado valor a errno.

Los valores posibles para modo_acceso se describieron en la función chmod().

La variable global _fmode está definida en los ficheros fcntl.h y stdlib.h de esta forma: int _fmode; Por defecto, se inicializa con el valor O_TEXT.

Ejemplo:

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char buf[11] = "0123456789";
```

```
   /* cambia el modo de fichero por defecto de texto a binario */
   _fmode = O_BINARY;

   /* crea un fichero binario para lectura y escritura */
   descriptor = creat ("FICHERO.FIC", S_IREAD | S_IWRITE);

   /* escribe 10 bytes al fichero */
   write (descriptor, buf, strlen (buf));

   /* cierra el fichero */
   close (descriptor);
   return 0;
}
```

---

creatnew — Crea un nuevo fichero.

Sintaxis:

```
   int creatnew (const char *path, int modo);
```

La función creatnew es identica a la función _creat() con la excepción de que es devuelto un error si el fichero ya existe. (Versiones del DOS 3.0 o superiores)

Los valores posibles para el parámetro modo se explicaron en la función creat().

Ejemplo:

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <dos.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char buf[11] = "0123456789";

   /* intenta crear un fichero que no existe */
   descriptor = creatnew ("FICHERO.FIC", 0);

   if (descriptor == -1)
     printf ("FICHERO.FIC ya existe.\n");
   else
     {
       printf ("FICHERO.FIC creado con éxito.\n");
       write (descriptor, buf, strlen (buf));
       close (descriptor);
     }
   return 0;
}
```

---

creattemp — Crea un fichero único en el directorio dado por el nombre de fichero.

Sintaxis:

```
   int creattemp (char *path, int atrib);
```

Esta función es similar a la función _creat(), excepto que el nombre de
fichero es el nombre de path que debe terminar con un \. El nombre de
fichero debería ser bastante grande para alojar el nombre de fichero.
(Versiones de MS-DOS 3.0 o superiores)

Los valores posibles para atrib son los mismos que para la función creat().

Ejemplo:

```
#include <string.h>
#include <stdio.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char nombre_de_path[128];

   strcpy (nombre_de_path, "\\"); /*copia el string "\\" en nombre_de_path*/

   /* crea un fichero único en el directorio raíz */
   descriptor = creattemp (nombre_de_path, 0);

   printf ("%s fue el fichero único creado.\n", nombre_de_path);
   close (descriptor);
   return 0;
}
```

---
  dup   Duplica un descriptor de fichero.
=====

Sintaxis:
```
   int dup (int descriptor);
```

Si la operación tiene éxito, dup() devuelve el descriptor del nuevo fichero;
en otro caso, dup() devuelve -1 y se asigna valor a errno.

---
  dup2   Duplica el descriptor de fichero viejo_descriptor sobre el
======   descriptor de fichero existente nuevo_descriptor.

Sintaxis:
```
   int dup2 (int viejo_descriptor, int nuevo_descriptor);
```

Devuelve 0 si tiene éxito; -1 si ocurre un error.

---
  eof   Chequea fin de fichero.
=====

Sintaxis:
```
   int eof (int descriptor);
```

Devuelve uno de los siguientes valores:

Valor	Significado
1	Fin de fichero
0	No fin de fichero
-1	Error; errno es asignado

```
------------
  filelength    Obtiene el tamaño de un fichero en bytes.
============

Sintaxis:
   long filelength (int descriptor);

Si ocurre un error, devuelve -1 y se asigna valor a errno.

Ejemplo:

#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char buf[11] = "0123456789";

   /* crea un fichero conteniendo 10 bytes */
   descriptor = open ("FICHERO.FIC", O_CREAT);
   write (descriptor, buf, strlen (buf));

   /* visualiza el tamaño del fichero */
   printf ("Longitud del fichero en bytes: %ld\n", filelength (descriptor));

   /* cierra el fichero */
   close (descriptor);
   return 0;
}


----------
  getftime    Obtiene fecha y hora de un fichero.
==========

Sintaxis:
   int getftime (int descriptor, struct ftime *pftime);

Devuelve 0 si tiene éxito, -1 si ocurre un error y además pone errno.

La descripción de struct ftime se ha descrito en la función setftime().


-------
  ioctl    Controla dispositivos de I/O.
=======

Sintaxis:
   int ioctl (int descriptor, int func [ , void *argdx, int argcx ] );

Para func 0 o 1, el valor devuelto es la información del dispositivo (DX de
la llamada de IOCTL).

Para valores de func de 2 hasta 5, el valor devuelto es el número de bytes
transferidos realmente.

Para los valores de func 6 ó 7, el valor devuelto es el estado del
dispositivo.
```

En cualquier caso, si es detectado un error, un valor de -1 es devuelto, y se da valor a errno.

IOCTL (Control de entrada/salida para los dispositivos) es la función 68 (44 hexadecimal) de la interrupción 21 hexadecimal del DOS. Esta función del DOS tiene varias subfunciones que realizan distintas tareas. En las versiones DOS-3 había 11 subfunciones. El argumento func de ioctl() del C se corresponde con dichas subfunciones. Para saber lo que hace cada sub-función debe consultar tu manual del DOS.

Los argumentos argdx y argcx se refieren a los registros DX y CX de la CPU.

---

isatty — Chequea tipo de dispositivo.

Sintaxis:

```
int isatty (int descriptor);
```

Si el dispositivo es un dispositivo de carácter, isatty() devuelve un valor distinto de cero.

Ejemplo:

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

int main (void)
{
   int descriptor;

   descriptor = fileno (stdprn);
   if (isatty (descriptor))
     printf("Descriptor %d es un tipo de dispositivo\n", descriptor);
   else
     printf("Descriptor %d no es un tipo de dispositivo\n", descriptor);
   return 0;
}
```

---

lock — Pone los bloqueos de compartición de ficheros para controlar el acceso concurrente de ficheros.

Sintaxis:

```
int lock (int descriptor, long desplazamiento, long longitud);
```

Previene el acceso de lectura o escritura por otro programa para la región que empieza en la dirección desplazamiento y abarca longitud bytes.

Devuelve 0 en caso de éxito, -1 en caso de error.

---

lseek — Mueve el puntero de fichero de lectura/escritura.

Sintaxis:

```
long lseek (int descriptor, long desplazamiento, int desde_donde);
```

Devuelve la nueva posición del fichero, medida en bytes desde el comienzo del fichero. Devuelve -1L en caso de error, y da valor a errno.

```
------
  open    Abre un fichero para lectura o escritura.
======

Sintaxis:
   int open (const char *path, int acceso [ , unsigned modo ] );

Si la operación se hace correctamente, open() devuelve un descriptor de
fichero; en cualquier otro caso devuelve -1 y le da valor a errno.

Las definiciones de bits para el argumento acceso están en el fichero
fcntl.h y son las siguientes:

  O_APPEND   Añade a final de fichero
  O_BINARY   No traslación
  O_CREAT    Crea y abre fichero
  O_EXCL     Apertura exclusiva
  O_RDONLY   Sólo lectura
  O_RDWR     Lectura/escritura
  O_TEXT     Traslación CR-LF
  O_TRUNC    Apertura con truncación
  O_WRONLY   Sólo escritura

Para _open(), el valor de acceso en MS-DOS 2.x está limitado a O_RDONLY,
O_WRONLY y O_RDWR.

Para MS-DOS 3.x, los siguientes valores adicionales pueden ser usados
también:

  O_NOINHERIT   Proceso hijo hereda fichero
  O_DENYALL     Error si abierto para lectura/escritura
  O_DENYWRITE   Error si abierto para escritura
  O_DENYREAD    Error si abierto para lectura
  O_DENYNONE    Permite acceso concurrente

Sólo una de las opciones O_DENYxxx pueden ser incluidas en una simple
apertura.

Los valores posibles que puede tomar el argumento modo se describieron
en la función chmod().

Ejemplo:

#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char mensaje[] = "Hola mundo";

   if ((descriptor = open ("PRUEBA.$$$", O_CREAT | O_TEXT)) == -1)
     {
       perror ("Error");
       return 1;
     }
   write (descriptor, mensaje, strlen (mensaje));
   close (descriptor);
   return 0;
}
```

```
-------
  _open    Abre un fichero para lectura o escritura.
=======

Sintaxis:
   int _open (const char *nombre_de_fichero, int flags);

Si la operación tiene éxito, _open() devuelve un descriptor de fichero; en
cualquier otro caso devuelve -1.

Las definiciones de bits para el argumento flags son las mismas que para
el argumento acceso en la descripción de la función open().

Ejemplo:

#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char mensaje[] = "Hola mundo";

   if ((descriptor = _open ("PRUEBA.$$$", O_RDWR)) == -1)
     {
       perror ("Error");
       return 1;
     }
   _write (descriptor, mensaje, strlen (mensaje));
   _close (descriptor);
   return 0;
}


------
  read    Lee de un fichero.
======

Sintaxis:
   int read (int descriptor, void *buffer, unsigned longitud);

Si la operación tiene éxito, devuelve un entero indicando el número de
bytes colocados en el buffer; si el fichero fue abierto en modo texto,
read() no cuenta retornos de carros o caracteres Ctrl-Z en el número de
bytes leídos.

En caso de error, devuelve -1 y le da valor a errno.


-------
  _read    Lee de fichero.
=======

Sintaxis:
   int _read (int descriptor, void *buffer, unsigned longitud);

Devuelve el número de bytes leídos; si se detecta el fin de fichero,
devuelve 0; si ocurre un error devuelve -1 y da valor a errno.
```

---
  setftime   Pone fecha y hora de un fichero.
==========

Sintaxis:
   int setftime (int descriptor, struct ftime *ptime);

Devuelve 0 si tiene éxito; en otro caso devuelve -1.

struct ftime está declarada en el fichero io.h del siguiente modo:

```
  struct ftime {
    unsigned  ft_tsec  : 5; /* intervalo de dos segundos */
    unsigned  ft_min   : 6; /* minutos */
    unsigned  ft_hour  : 5; /* horas */
    unsigned  ft_day   : 5; /* días */
    unsigned  ft_month : 4; /* meses */
    unsigned  ft_year  : 7; /* año */
  };
```

---
  setmode   Pone modo de un fichero de apertura.
=========

Sintaxis:
   int setmode (int descriptor, int modo);

Devuelve 0 si tiene éxito; en otro caso -1.

Los valores posibles de modo se describieron en la función open().

Ejemplo:

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int resultado;

   resultado = setmode (fileno (stdprn), O_TEXT);
   if (resultado == -1)
     perror ("Modo no disponible");
   else
     printf ("Modo cambiado con éxito");
   return 0;
}
```

---
  sopen   Macro que abre un fichero en el modo compartido.
=======

Sintaxis:
   sopen (path, acceso, shflag, modo)

Está incluido para compatibilidad con las distintas versiones de Turbo C
y otros compiladores.

El significado de los parámetros acceso y modo se han explicado en las
funciones open() y chmod() respectivamente. El parámetro shflag contiene
el modo de compartición de ficheros y las constantes definidas para ello

en el fichero share.h son las siguientes:

```
  SH_COMPAT     Modo de compatibilidad
  SH_DENYRW     Denegado acceso de lectura y escritura
  SH_DENYWR     Denegado acceso de escritura
  SH_DENYRD     Denegado acceso de lectura
  SH_DENYNONE   Permite acceso de lectura y escritura
  SH_DENYNO     Igual que SH_DENYNONE (compatibilidad)
```

---
  tell   Obtiene la posición corriente de un puntero de fichero.
======

Sintaxis:
```
   long tell (int descriptor);
```

Devuelve la posición actual del puntero de fichero o -1 en caso de error.

Ejemplo:

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main (void)
{
   int descriptor;
   char mensaje[] = "Hola mundo";

   if ((descriptor = open ("PRUEBA.$$$", O_CREAT | O_TEXT | O_APPEND)) == -1)
     {
       perror ("Error");
       return 1;
     }
   write (descriptor, mensaje, strlen (mensaje));
   printf ("El puntero de fichero está en el byte %ld\n", tell (descriptor));
   close (descriptor);
   return 0;
}
```

---
  unlink   Borra un fichero.
========

Sintaxis:
```
   int unlink (const char *nombre_de_fichero);
```

Si el nombre de fichero tiene atributo de sólo lectura, unlink fallará.
Llama a chmod() primero para cambiar el atributo de fichero.

Devuelve 0 en caso de éxito; -1 en caso de error.

El prototipo de esta función está en los ficheros dos.h, io.h y stdio.h.
La función remove() cuyo prototipo se encuentra en stdio.h es en realidad
una macro que se expande a una llamada a unlink().

---
  unlock   Libera bloqueos de compartición de ficheros para controlar el
========   acceso concurrente.

Sintaxis:
   int unlock (int descriptor, long desplazamiento, long longitud);

Devuelve 0 si tiene éxito; -1 si ocurre un error.

---
  write   Escribe en un fichero.
=======

Sintaxis:
   int write (int descriptor, void *buffer, unsigned longitud);

Devuelve el número de bytes escritos, o -1 si ocurre un error.

  _write   Escribe en un fichero.
========

Sintaxis:
   int _write (int descriptor, void *buffer, unsigned longitud);

Devuelve el número de bytes escritos, o -1 si ocurre un error.

  HANDLE_MAX (#define)

Número máximo de descriptores.

LECCIÓN

INDICE DE LA LECCION 10

- Primera parte:

  * Introducción a la biblioteca del C.
  * Valores límites (**limits.h**).
  * Tipos y macros estándares (**stddef.h**).
  * Funciones de caracteres y de cadenas (**ctype.h, string.h**).
    (también **mem.h** para los usuarios de Turbo C).

  * Funciones matemáticas (**math.h**).

- Segunda parte:

  * Funciones de pantalla y de gráficos (**conio.h** y **graphics.h**).

INTRODUCCION A LA BIBLIOTECA DEL C

La biblioteca de C contiene el código objeto de las funciones proporcionadas con el compilador.

Aunque las bibliotecas (ficheros con extensión .LIB) son parecidas a los ficheros objetos (fichero con extensión .OBJ), existe una diferencia crucial: No todo el código de una biblioteca se añade al programa. Cuando se enlaza un programa que consta de varios ficheros objetos, todo el código de cada fichero objeto se convierte en parte del programa ejecutable final. Esto ocurre se esté o no utilizando el código. En otras palabras, todos los ficheros objetos especificados en tiempo de enlace se unen para formar el programa. Sin embargo, este no es el caso de los ficheros de biblioteca.

Una biblioteca es una colección de funciones. A diferencia de un fichero objeto, un fichero de biblioteca guarda una serie de información para cada función de tal forma que cuando un programa hace referencia a una función contenida en una biblioteca, el enlazador toma esta función y añade el código objeto al programa. De esta forma sólo se añadirán al fichero ejecutable aquellas funciones que realmente se utilicen en el programa.

Para utilizar una función de biblioteca debemos incluir su correspondiente fichero de cabecera para que nuestro programa conozca el prototipo de la función a utilizar. En los ficheros de cabecera (suelen tener extensión .H) además de los prototipos de las funciones puede haber más información como macros, declaración de tipos, declaración de variables globales, etc.

Los ficheros de cabecera definidos por el estándar ANSI se presentan en la siguiente tabla.

Fichero de cabecera	Propósito
assert.h	Define la macro assert().
ctype.h	Uno de caracteres.
float.h	Define valores en coma flotante dependiente de la implementación.
limits.h	Define los límites dependientes de la implementación.
locale.h	Soporta la función setlocale().
math.h	Definiciones utilizadas por la biblioteca matemática.
setjmp.h	Soporta saltos no locales.
signal.h	Define los valores de señal.
stdarg.h	Soporta listas de argumentos de longitud variable.
stddef.h	Define algunas constantes de uso común.
stdio.h	Soporta la E/S de fichero.
stdlib.h	Otras declaraciones.
string.h	Soporta funciones de cadena.
time.h	Soporta las funciones de tiempo del sistema.

Turbo C añade los siguientes ficheros de cabecera:

Fichero de cabecera	Propósito
alloc.h	Asignación dinámica.
bios.h	Funciones relacionadas con la ROM BIOS.

conio.h	E/S por consola.
dir.h	Directorio.
dos.h	Sistema operativo DOS.
errno.h	Errores del sistema.
fcntl.h	Constantes simbólicas utilizadas por open().
graphics.h	Gráficos.
io.h	E/S estándar a bajo nivel.
mem.h	Funciones de memoria.
process.h	Funciones de proceso.
share.h	Constantes simbólicas utilizadas por sopen().
sys\stat	Información de ficheros.
sys\timeb	Hora actual.
sys\types	Definición de tipos.
values.h	Constantes simbólicas para compatibilidad con UNIX.

El resto de esta lección y las dos lecciones siguientes estarán dedicadas a la descripción de cada fichero de cabecera.h.

Nota: las características ya estudiadas en lecciones anteriores no se volverán a explicar (por ejemplo, ficheros stdarg.h y assert.h).

VALORES LIMITES

En el fichero de cabecera **<limits.h>** se encuentra una serie de macros que definen valores límites para algunos tipos de datos.

FICHERO DE CABECERA LIMITS.H

CONSTANTES SIMBOLICAS:
****

CHAR_xxx (#defines)

CHAR_BIT	Tipo char, número de bits en un dato tipo cahr
CHAR_MAX	Tipo char, mínimo valor
CHAR_MIN	Tipo char, máximo valor

Estos valores son independientes de si tipo char está definido como signed o unsigned por defecto.

INT_xxx (#defines)

INT_MAX	Tipo int, máximo valor
INT_MIN	Tipo int, mínimo valor

LONG_xxx #defines

LONG_MAX	Tipo long, máximo valor

```
  LONG_MIN   Tipo long, mínimo valor
```

Máximo and mínimo valor para tipo long.

```
----------------------
SCHAR_xxx (#defines)
----------------------

SCHAR_MAX   Tipo char, máximo valor
SCHAR_MIN   Tipo char, mínimo valor

----------------------
SHRT_xxx (#defines)
----------------------

SHRT_MAX   Tipo short, máximo valor
SHRT_MIN   Tipo short, mínimo valor

---------------------
  UCHAR_MAX (#define)
---------------------
```

Máximo valor para tipo unsigned char.

```
--------------------
  UINT_MAX (#define)
--------------------
```

Máximo valor para tipo unsigned int.

```
---------------------
  ULONG_MAX (#define)
---------------------
```

Máximo valor para tipo unsigned int.

```
---------------------
  USHRT_MAX (#define)
---------------------
```

Máximo valor para tipo unsigned short.

En el ejemplo 1 de esta lección se utilizan todas estas macros.

TIPOS Y MACROS ESTANDARES

El estándar ANSI define unos cuantos tipos estándares y macros en el fichero **<stddef.h>**.

FICHERO DE CABECERA STDDEF.H

El estándar ANSI define unos cuantos tipos estándares y macros en el fichero stddef.h. Uno de los tipos es ptrdiff_t que es un tipo entero con signo y el resultado de restar dos punteros. La macro size_t es el

tipo entero sin signo del resultado del operador de tiempo de compilación sizeof. Las dos macros son NULL (que se corresponde con los punteros nulos) y ERRNO (que se corresponde con el valor modificable utilizado para guardar los diferentes códigos de error generados por las funciones de biblioteca).

En Turbo C, la variable global errno sustituye a la macro ERRNO. Además el fichero stddef.h incluye el fichero errno.h. En el fichero errno.h se encuentran declaradas dos variables globales (errno y _doserrno) y las definiciones de los números de error.

FUNCIONES DE CARACTERES Y DE CADENAS

La biblioteca estándar de C tiene un rico y variado conjunto de funciones de manejo de **caracteres** y de **cadenas**.

La utilización de las funciones de caracteres requieren la inclusión del fichero **<ctype.h>**; y la utilización de las funciones de caracteres requieren la inclusión del fichero **<string.h>**.

En C una cadena es un array de caracteres que finaliza con un carácter nulo. Puesto que el C no tiene asociadas operaciones de comprobación de cadenas, es responsabilidad del programador proteger los arrays del desbordamiento. Si un array se desborda, el comportamiento queda indefinido.

Las funciones de caracteres toman un argumento entero, pero sólo se utiliza el byte menos significativo. En general, nosotros somos libres de utilizar un argumento de tipo carácter ya que automáticamente se transforma en entero en el momento de la llamada.

FICHERO DE CABECERA CTYPE.H

---
is* Macros definidas en ctype.h
=====

Macro	Verdad (true) si c es...
isalnum(c)	Una letra o dígito
isalpha(c)	Una letra
isdigit(c)	Un dígito
iscntrl(c)	Un carácter de borrado o un carácter de control ordinario
isascii(c)	Un carácter ASCII válido
isprint(c)	Un carácter imprimible
isgraph(c)	Un carácter imprimible, excepto el carácter espacio
islower(c)	Una letra minúscula
isupper(c)	Una letra mayúscula
ispunct(c)	Un carácter de puntuación
isspace(c)	Un espacio, tabulador, retorno de carro, nueva línea, tabulación vertical, o alimentación de línea
isxdigit(c)	Verdad si c es un dígito hexadecimal
Macros de TC	**Convierte c ...**
_toupper(c)	En el rango [a-z] a caracteres [A-Z]

```
_tolower(c) | En el rango [A-Z] a caracteres [a-z]
toascii(c)  | Mayor de 127 al rango 0-127 poniendo todos los bits a cero
            | excepto los 7 bits más significativos


-----
to*    Funciones declaradas en ctype.h
=====

----------------------+---------------------------
Prototipo             | Qué hace
----------------------+---------------------------
int toupper (int ch); | Devuelve ch en mayúscula
int tolower (int ch); | Devuelve ch en minúscula
```

OJO!!! Con las macros anteriores se debe evitar hacer operaciones como las siguientes:

```
  x = isdigit (getch ());
  y = isdigit (*p++);
```

Supongamos que la macro isdigit() está definida así:

```
#define isdigit(c) ((c) >= '0' && (c) <= '9')
```

Entonces las dos asignaciones anteriores se expanderían a:

```
  x = ((getch ()) >= '0' && (getch ()) <= '9')
  y = ((*p++) >= '0' && (*p++) <= '9')
```

El error cometido se ve claramente: en el primer caso nuestra intención era leer un sólo carácter y en realidad leemos dos, y en el segundo caso nuestra intención era incrementar en uno el puntero p y en realidad lo incrementamos en dos. Si isdigit() fuera una función en vez de una macro no habría ningún problema.

En el ejemplo 1 de esta lección se muestra cómo usar estas macros y estas funciones.

FICHERO DE CABECERA STRING.H

NOTA: Ver observaciones al final de esta ventana

GLOSARIO:

```
--------
  stpcpy    (TC) Copia un string en otro.
========
------------------
  strcat, _fstrcat    Añade fuente a destino.
==================
------------------
  strchr, _fstrchr    Encuentra c en s.
==================
------------------
  strcmp, _fstrcmp    Compara un string con otro.
==================
---------
  strcmpi    Macro que compara strings.
=========
```

----
  strcpy   Copia el string fuente al string destino.
========
----
  strcspn, _fstrcspn   Explora un string.
====================
----
  strdup, _fstrdup   (TC) Obtiene una copia duplicada de s, o copia s a
==================   una nueva localización.

  _strerror   (TC) Construye un mensaje de error hecho a medida.
===========
----
  strerror   Devuelve un puntero al string que contiene el mensaje de error.
==========
----
  stricmp, _fstricmp   (TC) Compara un string con otro ignorando el caso.
====================
----
  strlen, _fstrlen   Calcula la longitud de un string.
==================
----
  strlwr, _fstrlwr   (TC) Convierte s a caracteres en minúsculas.
==================
----
  strncat, _fstrncat   Añade como máximo longmax caracteres de fuente a
====================   destino.
----
  strncmp, _fstrncmp   Compara como mucho longmax caracteres de un string
====================   con otro.
----
  strncmpi   (TC) Compara un trozo de un string con un trozo de otro,
==========   sin sensibilidad al caso.
----
  strcoll   Compara dos strings.
=========
----
  strncpy, _fstrncpy   Copia como máximo longmax caracteres de fuente a
====================   destino.
----
  strnicmp, _fstrnicmp   (TC) Compara como máximo n caracteres de un string
======================   con otro, ignorando el caso.
----
  strnset, _fstrnset   (TC) Copia el carácter ch en las primeras n
====================   posiciones de s.
----
  strpbrk, _fstrpbrk   Explora un string.
====================
----
  strrchr, _fstrrchr   Encuentra la última ocurrencia de c en s.
====================
----
  strrev, _fstrrev   (TC) Invierte todos los caracteres de s (excepto el
==================   carácter terminador nulo).
----
  strset, _fstrset   (TC) Copia el carácter ch a todas las posiciones de s.
==================
----
  strspn, _fstrspn   Explora un string para encontrar un segmento que es un
==================   subconjunto de un conjunto de caracteres.
----
  strstr, _fstrstr   Encuentra la primera ocurrencia de un substring en
==================   un string.

```
  strtok, _fstrtok    Explora s1 para encontrar el primer token no contenido
==================   en s2.
------------------
  strupr, _fstrupr    (TC) Convierte todos los caracteres de s a mayúsculas.
==================
---------
  strxfrm    (TC) Transforma un trozo de un string.
=========


ESTUDIO DE LAS FUNCIONES DEL GLOSARIO:

--------
  stpcpy    (TC) Copia un string en otro.
========

Sintaxis:
   char *stpcpy (char *destino, const char *fuente);

Es igual que strcpy(), excepto que stpcpy devuelve

  destino + strlen (fuente).

Ejemplo:

#include <string.h>
#include <stdio.h>

void main (void)
{
   char s1[10];
   char *s2 = "abc";

   printf ("%s", stpcpy (s1, s2) - 2); /* s1 contiene  abc  e imprime  bc  */
}


------------------
  strcat, _fstrcat    Añade fuente a destino.
==================

Sintaxis:
  Near: char *strcat (char *destino, const char *fuente);
  Far:  char far * far _fstrcat (char far *destino, const char far *fuente);

Devuelve destino.

Ejemplo:

#include <stdio.h>
#include <string.h>

void main (void)
{
   char s1[10] = "a";
   char *s2 = "bc";

   printf ("%s", strcat (s1, s2)); /* s1 contiene  abc  e imprime  abc  */
}


------------------
  strchr, _fstrchr    Encuentra c en s.
==================
```

Sintaxis:
  Near: char *strchr (const char *s, int c);
  Far: char far * far _fstrchr (const char far *s, int c);

Devuelve un puntero a la primera ocurrencia del carácter c en s; si c no aparece en s, strchr() devuelve NULL.

Ejemplo:

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

void main (void)
{
   char *s = "abc";
   char c1 = 'b', c2 = 'd';

   printf ("%s", strchr (s, c1)); /* imprime  bc  */
   printf ("%s", strchr (s, c2)); /* imprime  (null)  */
}
```

---
  strcmp, _fstrcmp   Compara un string con otro.
==========

Sintaxis:
   int strcmp (const char *s1, const char *s2);
   int far _fstrcmp (const char far *s1, const char far *s2);

Devuelve uno de los valores siguientes:

  < 0 si s1 es menor que s2
== 0 si s1 es igual que s2
  > 0 si s1 es mayor que s2

Ejecuta una comparación con signo.

Ejemplo:

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

void main (void)
{
   char *s1 = "a";
   char *s2 = "bc";

   printf ("%d", strcmp (s1, s2)); /* imprime  -1  */
   printf ("%d", strcmp (s2, s1)); /* imprime  1  */
}
```

---
  strcmpi   Macro que compara strings.
=========

Sintaxis:
   int strcmpi (const char *s1, const char *s2);

Esta rutina está implementada como una macro para compatibilidad con otros compiladores. Es igual que stricmp().

Ejemplo:

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

void main (void)
{
   char *s1 = "aaa";
   char *s2 = "AAA";

   printf ("%d", strcmpi (s1, s2)); /* imprime  0  */
}
```

---
  strcoll   Compara dos strings.
=========

Sintaxis:

```
   int strcoll (char *s1, char *s2);
```

La función strcoll() compara el string apuntado por s1 con el string apuntado por s2, según los valores puestos por setlocale().

Ejemplo:

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

void main (void)
{
   char *s1 = "a";
   char *s2 = "bc";

   printf ("%d", strcoll (s1, s2)); /* imprime  -1  */
   printf ("%d", strcoll (s2, s1)); /* imprime  1  */
}
```

---
  strcpy   Copia el string fuente al string destino.
========

Sintaxis:

```
   char *strcpy (char *destino, const char *fuente);
```

Devuelve destino.

Ejemplo:

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

void main (void)
{
   char s1[10] = "def";
   char *s2 = "abc";

   printf ("%s", strcpy (s1, s2)); /* s1 contiene  abc  e imprime  abc  */
}
```

---
  strcspn, _fstrcspn   Explora un string.

```
====================

Sintaxis:
  Near: size_t strcspn (const char *s1, const char *s2);
  Far:  size_t far far _fstrcspn (const char far *s1, const char far *s2);

Devuelve la longitud del substring inicial apuntado por s1 que está
constituido sólo por aquellos caracteres que no están contenidos en el
string 2. Dicho de otra forma, strcspn() devuelve el índice el primer
carácter en el string apuntado por s1 que está como carácter del string
apuntado por s2.

Ejemplo:

#include <stdio.h>
#include <string.h>

void main (void)
{
   char *s1 = "defdb";
   char *s2 = "abc";

   printf ("%d", strcspn (s1, s2)); /* imprime  4  */
}


------------------
  strdup, _fstrdup    (TC) Obtiene una copia duplicada de s, o copia s a
==================   una nueva localización.

Sintaxis:
  Near: char *strdup (const char *s);
  Far:  char far * far _fstrdup (const char far *s);

Devuelve un puntero a la copia duplicada de s, o devuelve NULL si no hubo
espacio suficiente para la asignación de la copia. El programador es res-
ponsable de liberar el espacio asignado por strdup() cuando ya no sea ne-
cesario.

Ejemplo:

#include <stdio.h>
#include <string.h>
#include <alloc.h>

void main (void)
{
   char *s1;
   char *s2 = "abc";

   if ((s1 = strdup (s2)) != NULL)
     {
       printf ("%s", s1); /* imprime  abc  */
       free (s1);
     }
}


-----------
  _strerror    (TC) Construye un mensaje de error hecho a medida.
===========

Sintaxis:
   char *_strerror (const char *s);
```

El mensaje de error está constituido por s, dos puntos, un espacio, el mensaje de error más reciente generado por el sistema, y un carácter de nueva línea.

El string s debe tener 94 caracteres o menos.

Devuelve un puntero al string que contiene el mensaje de error.

Ejemplo:

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

void main (void)
{
   printf ("%s", _strerror ("Prueba")); /* imprime  Prueba: Error 0  */
} /* la impresión descrita puede variar entre distintas implementaciones */
```

---
  strerror   Devuelve un puntero al string que contiene el mensaje de error.
==========

Sintaxis:
```
   char *strerror (int numerr);
```

Devuelve un puntero al mensaje de error asociado con numerr.

Ejemplo:

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

void main (void)
{
   printf ("%s", strerror (3)); /* imprime  Path not found  */
} /* la impresión descrita puede variar entre distintas implementaciones */
```

---
  stricmp, _fstricmp   (TC) Compara un string con otro ignorando el caso.
==========

Sintaxis:
```
  Near: int stricmp (const char *s1, const char *s2);
  Far:  int far _fstrcmp (const char far *s1, const char far *s2);
```

Devuelve uno de los siguientes valores:

```
   < 0 si s1 es menor que s2
  == 0 si s1 es igual que s2
   > 0 si s1 es mayor que s2
```

Ejecuta una comparación con signo.

La macro strcmpi(), definida en string.h, es equivalente a stricmp().

Ejemplo:

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

void main (void)
```

```
{
   char *s1 = "aaa";
   char *s2 = "AAA";

   printf ("%d", strcmpi (s1, s2)); /* imprime  0  */
}
```

---

strlen, _fstrlen — Calcula la longitud de un string.

Sintaxis:

```
  Near: size_t strlen (const char *s);
  Far: size_t _fstrlen (const char far *s);
```

Devuelve el número de caracteres que hay en s, no contando el carácter terminador nulo.

Ejemplo:

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

void main (void)
{
   char s[10] = "abc";

   printf ("%d", strlen (s)); /* imprime  3  */
   printf ("%d", sizeof (s)); /* imprime  10  */
}
```

---

strlwr, _fstrlwr — (TC) Convierte s a caracteres en minúsculas.

Sintaxis:

```
  Near: char *strlwr (char *s);
  Far:  char far * far _fstrlwr (char char far *s);
```

Devuelve un puntero a s.

Ejemplo:

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

void main (void)
{
   char *s = "AbC";

   printf ("%s", strlwr (s)); /* imprime  abc  */
}
```

---

strncat, _fstrncat — Añade como máximo longmax caracteres de fuente a destino.

Sintaxis:

```
  Near: char *strncat (char *destino, const char *fuente, size_t longmax);
  Far:  char far * far _fstrncat (char far *destino, const char far *fuente,
                                  size_t longmax);
```

Devuelve destino.

Ejemplo:

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

void main (void)
{
   char s1[10];
   char *s2 = "bcdef";
   int n1 = 2;
   int n2 = 10;

   strcpy (s1, "a");
   printf ("%s", strncat (s1, s2, n1)); /* imprime  abc  */
   strcpy (s1, "a");
   printf ("%s", strncat (s1, s2, n2)); /* imprime  abcdef  */
}
```

---

strncmp, _fstrncmp Compara como mucho longmax caracteres de un string con otro.

Sintaxis:

```
  Near: int strncmp (const char *s1, const char *s2, size_t longmax);
  Far:  int far _fstrncmp (const char far *s1, const char far *s2,
                           size_t longmax);
```

Devuelve uno de los siguientes valores:

```
  < 0 si s1 es menor que s2
== 0 si s1 es igual que s2
  > 0 si s1 es mayor que s2
```

Ejecuta una comparación con signo.

Ejemplo:

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

void main (void)
{
   char *s1 = "aaa";
   char *s2 = "aab";
   int n1 = 2;
   int n2 = 5;

   printf ("%d", strncmp (s1, s2, n1)); /* imprime  0  */
   printf ("%d", strncmp (s1, s2, n2)); /* imprime  -1  */
}
```

---

strncmpi (TC) Compara un trozo de un string con un trozo de otro, sin sensibilidad al caso.

Sintaxis:

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

La función strncmpi() ejecuta una comparación con signo entre s1 y s2, para

una longitud máxima de n bytes, empezando con el primer carácter de cada string y continuando con los caracteres siguientes hasta encontrar caracteres correspondientes diferentes o hasta que se han examinado n caracteres.

Devuelve un valor (<0, 0, o >0) basado en el resultado de la comparación de s1 (o parte de él) con s2 (o parte de él).

Ejemplo:

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

void main (void)
{
   char *s1 = "aaa";
   char *s2 = "AAb";
   int n1 = 2;
   int n2 = 5;

   printf ("%d", strncmpi (s1, s2, n1)); /* imprime  0  */
   printf ("%d", strncmpi (s1, s2, n2)); /* imprime  -1  */
}
```

---

strncpy, _fstrncpy

Copia como máximo longmax caracteres de fuente a destino.

Sintaxis:

```
Near: char *strncpy (char *destino, const char *fuente, size_t longmax);
Far:  char far * far _fstrncpy (char far *destino, const char far *fuente,
                                size_t longmax)
```

Si son copiados longmax caracteres, no es añadido el carácter nulo; por lo tanto, el contenido de destino no es un string terminado en nulo.

Devuelve destino.

Ejemplo:

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

void main (void)
{
   char s1[10];
   char *s2 = "abcd";
   int n1 = 2;
   int n2 = 10;

   printf ("%s", (strncpy (s1, s2, n1), *(s1+n1) = 0, s1)); /*imprime  ab */
   printf ("%s", strncpy (s1, s2, n2)); /* imprime  abcd  */
}
```

---

strnicmp, _fstrnicmp

(TC) Compara como máximo n caracteres de un string con otro, ignorando el caso.

Sintaxis:

```
Near: int strnicmp (const char *s1, const char *s2, size_t longmax);
Far:  int far _fstrnicmp (const char far *s1, const char far *s2,
                          size_t longmax);
```

Devuelve uno de los siguientes valores:

```
   < 0 si s1 es menor que s2
  == 0 si s1 es igual que s2
   > 0 si s1 es mayor que s2
```

Ejecuta comparación con signo.

La macro strncmpi(), definida en string.h, es equivalente a strnicmp().

Ejemplo:

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

void main (void)
{
   char *s1 = "aaa";
   char *s2 = "AAb";
   int n1 = 2;
   int n2 = 5;

   printf ("%d", strnicmp (s1, s2, n1)); /* imprime  0  */
   printf ("%d", strnicmp (s1, s2, n2)); /* imprime  -1  */
}
```

---

strnset, _fstrnset

(TC) Copia el carácter ch en las primeras n posiciones de s.

Sintaxis:

```
  Near: char *strnset (int *s, int ch, size_t n);
  Far:  char far * far _fstrnset (char far *s, int ch, size_t n);
```

Se para cuando los n caracteres son copiados o se encuentra NULL. Devuelve un puntero a s.

Ejemplo:

```
"
#include <stdio.h>
#include <string.h>

void main (void)
{
   char s[10] = "abc";
   char c1 = 'x';
   char c2 = 'y';
   int n1 = 10;
   int n2 = 2;

   printf ("%s", strnset (s, c1, n1)); /* s contiene  xxx  e imprime  xxx  */
   printf ("%s", strnset (s, c2, n2)); /* s contiene  yyx  e imprime  yyx  */
}
```

---

strpbrk, _fstrpbrk

Explora un string.

Sintaxis:

```
  Near: char *strpbrk (const char *s1, const char *s2);
  Far:  char far * far _fstrpbrk (const char far *s1, const char far *s2);
```

Devuelve un puntero al primer carácter del string apuntado por s1 que se corresponde con algún carácter en el string apuntado por s2. El carácter nulo de terminación no se incluye. Si no hay correspondencia, se devuelve un puntero nulo.

Ejemplo:

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

void main (void)
{
   char *s1 = "abc";
   char *s2 = "de";
   char *s3 = "db";

   printf ("%s", strpbrk (s1, s2)); /* imprime  (null)  */
   printf ("%s", strpbrk (s1, s3)); /* imprime  bc  */
}
```

---

strrchr, _fstrrchr — Encuentra la última ocurrencia de c en s.

Sintaxis:

```
Near: char *strrchr (const char *s, int c);
Far:  char far * far _fstrrchr (const char far *s, int c);
```

Devuelve un puntero a la última ocurrencia del carácter c, o NULL si c no aparece en s.

Ejemplo:

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

void main (void)
{
   char *s = "abcabc";
   char c1 = 'b', c2 = 'd';

   printf ("%s", strrchr (s, c1)); /* imprime  bc  */
   printf ("%s", strrchr (s, c2)); /* imprime  (null)  */
}
```

---

strrev, _fstrrev — (TC) Invierte todos los caracteres de s (excepto el carácter terminador nulo).

Sintaxis:

```
Near: char *strrev (char *s);
Far:  char far * far _fstrrev (char far *s);
```

Devuelve un puntero al string invertido.

Ejemplo:

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

void main (void)
{
```

```
   char *s = "abc";

   printf ("%s", strrev (s)); /* imprime  cba  */
}
```

---

strset, _fstrset — (TC) Copia el carácter ch a todas las posiciones de s.

Sintaxis:

```
  Near: char *strset (char *s, int ch);
  Far:  char far * far _fstrset (char far *s, int ch);
```

Termina cuando se encuentra el primer carácter nulo.

Devuelve un puntero a s.

Ejemplo:

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

void main (void)
{
   char s[10] = "abcd";
   char c = 'z';

   printf ("%s", strset (s, c)); /* s contiene  zzzz  e imprime  zzzz  */
}
```

---

strspn, _fstrspn — Explora un string para encontrar un segmento que es un subconjunto de un conjunto de caracteres.

Sintaxis:

```
  Near: size_t strspn (const char *s1, const char *s2);
  Far:  size_t far _fstrspn (const char far *s1, const char far *s2);
```

Devuelve la longitud del substring inicial del string apuntado por s1 que está constituido sólo por aquellos caracteres contenidos en el string apuntado por s2. Dicho de otra forma, strspn() devuelve el índice del primer carácter en el string apuntado por s1 que no se corresponde con ningún carácter del string apuntado por s2.

Ejemplo:

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

void main (void)
{
   char *s1 = "acbbad";
   char *s2 = "abc";

   printf ("%d", strspn (s1, s2)); /* imprime  5  */
}
```

---

strstr, _fstrstr — Encuentra la primera ocurrencia de un substring en un string.

Sintaxis:
  Near: char *strstr (const char *s1, const char *s2);
  Far: char far * far _fstrstr (const char far *s1, const char far *s2);

Devuelve un puntero a la primera ocurrencia en la cadena apuntada por s1
de la cadena apuntada por s2 (excepto el carácter nulo de terminación de
s2). Devuelve un puntero nulo si no se encuentra.

Ejemplo:

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

void main (void)
{
   char *s1 = "abcdef";
   char *s2 = "cd";
   char *s3 = "cdg";

   printf ("%s", strstr (s1, s2)); /* imprime  cdef  */
   printf ("%s", strstr (s1, s3)); /* imprime  (null)  */
}
```

---

strtok, _fstrtok

Explora s1 para encontrar el primer token no contenido en s2.

Sintaxis:
  Near: char *strtok (char *s1, const char *s2);
  Far: char far * far _fstrtok (char far *s1, const char far *s2);

s2 define caracteres separadores; strtok() interpreta el string s1 como
una serie de tokens separados por los caracteres separadores que hay en s2.

Si no se encuentra ningún token en s1, strtok() devuelve NULL.

Si se encuentra un token, se pone un carácter nulo en s1 siguiendo al
token, y strtok() devuelve un puntero al token.

En las llamadas siguientes a strtok() con NULL como primer argumento, usa
el string previo s1, empezando después del último token encontrado.

Nótese que la cadena inicial es, por tanto, destruida.

Ejemplo:

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

void main (void)
{
   char *s1 = "ab cd, e; fg,h:ijk.";
   char *s2 = " ,;:.";
   char *p;

   p = strtok (s1, s2);
   while (p)
     {
       printf (p);
       p = strtok (NULL, s2);
     }
   /* el bucle while imprime:  abcdefghijk  */
}
```

```
------------------
  strupr, _fstrupr    (TC) Convierte todos los caracteres de s a mayúsculas.
==================
```

Sintaxis:

```
  Near: char *strupr (char *s);
  Far:  char far * far _fstrupr (char far *s);
```

Devuelve un puntero a s.

Ejemplo:

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

void main (void)
{
   char *s = "aBc";

   printf ("%s", strupr (s)); /* imprime  ABC  */
}
```

```
---------
  strxfrm    (TC) Transforma un trozo de un string.
=========
```

Sintaxis:

```
   size_t strxfrm (char *s1, char *s2, size_t n);
```

Es equivalente a strncpy().

Ejemplo:

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

void main (void)
{
   char s1[10];
   char *s2 = "abcdefg";
   printf ("\n%d ", strxfrm (s1, s2, 10)); /* s1 contiene  abcdefg
                                              e imprime  7  */
}
```

OBSERVACIONES:

1) Aquellas funciones que tienen al principio de la descripción: (TC) no están en el ANSI C y sí en TURBO C. Asimismo, la distinción entre funciones near y far no está en el ANSI, sino que también es propio de TURBO C: el prototipo de near en aquellas funciones que no tienen (TC) es el prototipo de la función en el ANSI. No obstante, aquellos usuarios que tengan la opción turbo a off, verán todas las funciones ya que muchas de estas funciones que no pertenecen al ANSI C están en muchos compiladores y además así tienen una visión más amplia de la variedad de operaciones que se pueden hacer con los strings (llamados en castellano cadenas).

2) Las funciones memccpy(), memchr(), memcmp(), memcpy(), memicmp(), memmove(), memset(), movedata(), movmem() y setmem() están declaradas en Turbo C en los ficheros string.h y mem.h. De entre las funciones

anteriores, pertenecen al ANSI las funciones memchr(), memcmp(), memcpy() y memset(), entre todas éstas, la función memchr() está declarada en el ANSI en el fichero ctype.h y el resto en string.h. Estas funciones las estudiarán en la ventana siguiente los usuarios que tengan activada la opción de turbo puesta a on; los que tengan la opción de turbo a off no la estudiarán porque en realidad son superfluas: todo lo que se puede hacer con ellas, se puede hacer con las funciones de cadenas.

FICHERO DE CABECERA MEM.H

GLOSARIO:

---
memccpy, _fmemccpy Copia un bloque de n bytes de fuente a destino.
==========
---
memchr, _fmemchr Busca el carácter c en los primeros n bytes del array s.
==========
---
memcmp, _fmemcmp Compara dos strings, s1 y s2, para una longitud de
========== n bytes exactamente.
---
memcpy, _fmemcpy Copia un bloque de n bytes de fuente a destino.
==========
---
memicmp, _fmemicmp Compara los primeros n bytes de s1 y s2, ignorando
========== el caso.
---
memmove Copia un bloque de n bytes de fuente a destino.
==========
---
memset, _fmemset Copia n veces el byte c en s.
==========
---
movedata Copia n bytes.
==========
---
movmem Mueve un bloque de longitud bytes de fuente a destino.
==========
---
setmem Asigna un valor a un rango de memoria.
==========

DESCRIPCION DE LAS FUNCIONES QUE APARECEN EN EL GLOSARIO:

---
memccpy, _fmemccpy Copia un bloque de n bytes de fuente a destino.
==========

Sintaxis:

```
Near: void *memccpy (void *destino, const void *fuente, int c, size_t n);
Far:  void far * far _fmemccpy (void far *destino, const void far *fuente,
                                int c, size_t n);
```

Esta función se para después de copiar un byte que coincida con c y devuelve un puntero al byte en destino inmediatamente siguiente a c; en otro caso devuelve NULL.

Ejemplo:

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

void main (void)
{
   char s1[10];
   const char *s2 = "abc";
   const char c1 = 'd';
   const char c2 = 'b';

   printf ("%s", memccpy (s1, s2, c1, strlen (s2))); /* imprime  (null)  */
   printf ("%s", memccpy (s1, s2, c2, strlen (s2))); /* imprime  c  */
}
```

---

memchr, _fmemchr — Busca el carácter c en los primeros n bytes del array s.

Sintaxis:

```
Near: void *memchr (const void *s, int c, size_t n);
Far:  void far * far _fmemchr (const void far *s, int c, size_t n);
```

Devuelve un puntero a la primera ocurrencia de c en s; devuelve NULL si c no aparece en el array s.

Ejemplo:

```
#include <stdio.h>
#include <mem.h>
#include <string.h>

void main (void)
{
   const char *s = "abc";
   const char c1 = 'd';
   const char c2 = 'b';

   printf ("%s", memchr (s, c1, strlen (s))); /* imprime  (null)  */
   printf ("%s", memchr (s, c2, strlen (s))); /* imprime  (bc)  */
}
```

---

memcmp, _fmemcmp — Compara dos strings, s1 y s2, para una longitud de n bytes exactamente.

Sintaxis:

```
Near: int memcmp (const void *s1, const void *s2, size_t n);
Far:  int far _fmemcmp (const void far *s1, const void far *s2, size_t n);
```

Devuelve uno de los valores siguientes:

```
 < 0 si s1 is menor que s2
== 0 si s1 is igual que s2
 > 0 si s1 is mayor que s2
```

Ejemplo:

```
#include <stdio.h>
#include <mem.h>
#include <string.h>
```

```
void main (void)
{
   const char *s1 = "abc";
   const char *s2 = "aBc";

   printf ("%d", memcmp (s1, s2, strlen (s1))); /* imprime  32  */
}
```

---

memcpy, _fmemcpy — Copia un bloque de n bytes de fuente a destino.

Sintaxis:

```
  Near: void *memcpy (void *destino, const void *fuente, size_t n);
  Far:  void far *far _fmemcpy (void far *destino, const void far *fuente,
                                size_t n);
```

Devuelve destino.

Ejemplo:

```
#include <stdio.h>
#include <mem.h>
#include <string.h>

void main (void)
{
   char s1[10];
   const char *s2 = "abc";

   printf ("%s", memcpy (s1, s2, strlen (s2))); /* imprime  abc  */
}
```

---

memicmp, _fmemicmp — Compara los primeros n bytes de s1 y s2, ignorando el caso.

Sintaxis:

```
  Near: int memicmp (const void *s1, const void *s2, size_t n);
  Far:  int far _fmemicmp (const void far *s1, const void far *s2, size_t n);
```

Devuelve uno de los valores siguientes:

```
   < 0 si s1 is menor que s2
  == 0 si s1 is igual que s2
   > 0 si s1 is mayor que s2
```

Ejemplo:

```
#include <stdio.h>
#include <mem.h>
#include <string.h>

void main (void)
{
   const char *s1 = "abc";
   const char *s2 = "aBc";

   printf ("%d", memicmp (s1, s2, strlen (s1))); /* imprime  0  */
}
```

```
---------
  memmove    Copia un bloque de n bytes de fuente a destino.
=========

Sintaxis:
   void *memmove (void *destino, const void *fuente, size_t n);

Devuelve destino.

Ejemplo:

#include <stdio.h>
#include <mem.h>

void main (void)
{
   char s1[10];
   const char *s2 = "abc";

   printf ("%s", memmove (s1, s2, strlen (s2))); /* imprime  abc  */
}


------------------
  memset, _fmemset    Copia n veces el byte c en s.
==================

Sintaxis:
  Near: void *memset (void *s, int c, size_t n);
  Far:  void far * far _fmemset (void far *s, int c, size_t n);

Devuelve s.

Ejemplo:

#include <stdio.h>
#include <mem.h>

void main (void)
{
   char s[5];
   char c = 'a';

   printf ("%s", (memset (s, c, 4), s[4] = 0, s)); /* imprime  aaaa  */
}


----------
  movedata    Copia n bytes.
==========

Sintaxis:
   void movedata(unsigned segmento_fuente, unsigned desplazamiento_fuente,
                 unsigned segmento_destino, unsigned desplazamiento_destino,
                 size_t n);

Copia n bytes de segmento_fuente:desplazamiento_fuente a
segmento_destino:desplazamiento_destino.

Ejemplo:

#include <stdio.h> /* printf () */
#include <mem.h>   /* movedata () */
#include <dos.h>   /* FP_SEG(): devuelve segmento de una dirección
```

```
                    FP_OFF(): devuelve desplazamiento de una dirección */

void main (void)
{
  char far *s1[10];
  char far *s2 = "abc";

  movedata (FP_SEG (s2), FP_OFF (s2), FP_SEG (s1), FP_OFF (s1), strlen (s2));
  s1[strlen(s1)] = 0;
  printf ("%s", s1); /* imprime  abc  */
}
```

---

movmem Mueve un bloque de longitud bytes de fuente a destino.

Sintaxis:

```
   void movmem (void *fuente, void *destino, unsigned longitud);
```

Ejemplo:

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

void main (void)
{
   char s1[10];
   char *s2 = "abc";

   movmem (s2, s1, strlen (s2));
   printf ("%s", s1); /* imprime  abc  */
}
```

---

setmem Asigna un valor a un rango de memoria.

Sintaxis:

```
   void setmem (void *destino, int longitud, char valor);
```

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

void main (void)
{
   char s[10];
   char c = 'a';

   setmem (s, sizeof(s)-1, c);
   s[9] = 0;
   printf ("%s", s); /* imprime  aaaaaaaaa  */
}
```

FUNCIONES MATEMATICAS

Las funciones matemáticas toman argumentos de tipo **double** y devuelven valores de tipo **double**. Estas funciones se dividen en las siguientes

categorías:
- Funciones trigonométricas. - Funciones hiperbólicas.
- Funciones logarítmicas y exponenciales. - Otras.
Todas las funciones matemáticas necesitan la cabecera **<math.h>** en cualquier programa que las utilice. Además, al declarar las funciones matemáticas, esta cabecera define tres macros: **EDOM**, **ERANGE**, y **HUGE_VAL**. Si uno de los argumentos de las funciones matemáticas no se encuentra en uno de los dominios para el cual está definido, se devuelve un valor definido por la implementación y la variable global **errno** se activa a **EDOM** (error de dominio). Si el resultado de una rutina es demasiado grande como para ser representado por un tipo **double**, se produce desbordamiento. Esto da lugar a que la rutina devuelva **HUGE_VAL**, y **errno** se active a **ERANGE** (que indica error de rango). Si se produce un desbordamiento por abjao, la rutina devuelve 0 y activa **errno** a **ERANGE**.

FICHERO DE CABECERA MATH.H

GLOSARIO:
****

---
  abs Macro que devuelve el valor absoluto de de un entero.
=====
------
  acos Arcocoseno.
======
------
  asin Arcoseno.
======
------
  atan Arcotangente.
======
----
  atan2 Arcotangente de y/x.
=======
------
  atof Convierte una cadena punto flotante.
======
------
  cabs (TC) Valor absoluto de un número complejo.
======

------
  ceil Redondea por arriba.
======
-----
  cos Coseno.
=====
------
  cosh Coseno hiperbólico.
======
-----
  exp Calcula e elevando a la x-éxima potencia.
=====
------
  fabs Valor absoluto de valor en punto flotante.
======
----
  floor Redondea por abajo.
=======

```
------
  fmod    Calcula x módulo y, el resto de x/y.
======
-------
  frexp    Descompone un double en mantisa y exponente.
=======
-------
  hypot    (TC) Calcula hipotenusa de un triángulo rectángulo.
=======

------
  labs    Calcula el valor absoluto de un long.
======
-------
  ldexp    Calcula el producto entre x y 2 elevado a exp.
=======
-----
  log    Función logaritmo neperiano.
=====
-------
  log10    Función logaritmo en base 10.
=======
---------
  matherr    (TC) Define un manejador de errores matemáticos.
=========

------
  modf    Descompone en parte entera y parte fraccionaria.
======
------
  poly    (TC) Genera un polinomio de los argumentos de esta función.
======

-----
  pow    Función potencia, x elevado a y.
=====
-------
  pow10    (TC) Función potencia, 10 elevado a p.
=======

-----
  sin    Función seno.
=====
------
  sinh    Función seno hiperbólico.
======
------
  sqrt    Calcula raíz cuadrada.
======
-----
  tan    Tangente.
=====
------
  tanh    Tangente hiperbólica.
======
------------------
  COMPLEX (struct)    (TC) Representación de número complejo.
==================

----------------
  EDOM (#define)     Código de error para error de dominio.
================
------------------
  ERANGE (#define)     Código de error para error de rango.
```

==========

  EXCEPTION (struct)   (TC) Formato de información de error.
==========

  HUGE_VAL (#define)   Valor de overflow para las funciones matemáticas.

  M_xxxx (#defines)   (TC) Valores constantes para funciones logarítmicas.

  PI (#defines)   (TC) Constantes comunes de ã.
==========

  M_SQRTxx (#defines)   (TC) Valores constantes para raíces cuadradas de 2.

  _mexcep (enum)   (TC) Errores matemáticos.

FUNCIONES:
****

-----
  abs   Macro que devuelve el valor absoluto de de un entero.
=====

Sintaxis:
   int abs (int x);

En Turbo C, el prototipo de abs() está en math.h y en stdlib.h. En el
ANSI C sólo se encuentra en el fichero stdlib.h

------
  acos   Arcocoseno.
======

Sintaxis:
   double acos (double x);

Devuelve el arcocoseno de x (valores de 0 a ã). El argumento x debe estar
en el rango -1 a 1; en cualquier otro caso se produce un error de dominio.

------
  asin   Arcoseno.
======

Sintaxis:
   double asin (double x);

Devuelve el arcoseno de x (valores en el rango -ã/2 a ã/2). El argumento x
debe estar en el rango de -1 a 1; en cualquier otro caso se produce un error
de dominio.

------
  atan   Arcotangente.
======

Sintaxis:
```
   double atan (double x);
```

Devuelve el arcotangente de x (un valor en el rango -ã/2 a ã/2).

---
atan2 Arcotangente de y/x.

Sintaxis:
```
   double atan2 (double y, double x);
```

Devuelve el arcotangente de y/x (un valor en el rango -ã a ã). Utiliza el signo de sus argumentos para obtener el cuadrante del valor devuelto.

---
atof Convierte una cadena punto flotante.

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

Devuelve el valor contenido en s convertido a tipo double, o 0 si s no puede ser convertido.

En Turbo C, el prototipo de atof() está en math.h y en stdlib.h. En el ANSI C sólo se encuentra en el fichero stdlib.h

---
cabs (TC) Valor absoluto de un número complejo.

Sintaxis:
```
   double cabs (struct complex z);
```

Devuelve el valor absoluto de z como un double.

Ver cómo es struct complex al final de esta ventana.

Ejemplo:

```
  /* Este programa imprime:  El valor absoluto de 2.00i 1.00j es 2.24  */

  #include <stdio.h>
  #include <math.h>

  int main (void)
  {
    struct complex z;
    double val;

    z.x = 2.0;
    z.y = 1.0;
    val = cabs (z);

    printf ("El valor absoluto de %.2lfi %.2lfj es %.2lf", z.x, z.y, val);
    return 0;
  }
```

```
------
  ceil    Redondea por arriba.
======

Sintaxis:
   double ceil (double x);

Devuelve el menor entero mayor o igual que x y lo representa como double.

Por ejemplo, dado 1.02, ceil() devuelve 2.0. Dado -1.02, ceil() devuelve -1.


-----
  cos    Coseno.
=====

Sintaxis:
   double cos (double x);

Devuelve el coseno de x. El valor de x debe darse en radianes.


------
  cosh    Coseno hiperbólico.
======

Sintaxis:
   double cosh (double x);

Devuelve el coseno hiperbólico de x. El valor de x debe darse en radianes.


-----
  exp    Calcula e elevando a la x-éxima potencia.
=====

Sintaxis:
   double exp (double x);

Devuelve el número e elevado a la potencia de x.


------
  fabs    Valor absoluto de valor en punto flotante.
======

Sintaxis:
   double fabs (double x);

Devuelve el valor absoluto de x.


-------
  floor    Redondea por abajo.
=======

Sintaxis:
   double floor (double x);

Devuelve el mayor entero (representando en double) que no es mayor que num.

Por ejemplo, dado 1.02, floor() devuelve 1.0. Dado -1.02, floor()
devuelve -2.0.
```

```
------
  fmod    Calcula x módulo y, el resto de x/y.
======

Sintaxis:
   double fmod (double x, double y);

Devuelve el resto de la división entera x/y.


-------
  frexp    Descompone un double en mantisa y exponente.
=======

Sintaxis:
   double frexp (double x, int *exp);

La función frexp() descompone el número x en una mantisa de rango entre
0.5 y 1 y en un exponente entero tal que x = mantisa * (2 elevado a exp).
Se devuelve la mantisa, y el exponente se guarda en la variable apuntada
por exp.


-------
  hypot    (TC) Calcula hipotenusa de un triángulo rectángulo.
=======

Sintaxis:
   double hypot (double x, double y);

Devuelve hipotenusa de un triángulo rectángulo en el que los catetos son
x e y.



------
  labs    Calcula el valor absoluto de un long.
======

Sintaxis:
   long int labs (long int x);

Devuelve el valor absoluto de x, un long int.

En Turbo C, el prototipo de labs() está en math.h y en stdlib.h. En el
ANSI C sólo se encuentra en el fichero stdlib.h


-------
  ldexp    Calcula el producto entre x y 2 elevado a exp.
=======

Sintaxis:
   double ldexp (double x, int exp);

Devuelve x * pow (2, exp). Si se produce desbordamiento, se devuelve
HUGH_VAL.


-----
  log    Función logaritmo neperiano.
=====
```

Sintaxis:
```
   double log (double x);
```

Devuelve el logaritmo neperiano de x. Se produce error de dominio si x es negativo, y un error de rango si el argumento es 0.

---
log10 Función logaritmo en base 10.

Sintaxis:
```
   double log10 (double x);
```

Devuelve el logaritmo en base 10 de x. Se produce un error de dominio si x es negativo, y un error de rango si el argumento es 0.

---
matherr (TC) Define un manejador de errores matemáticos.

Sintaxis:
```
   int matherr (struct exception *e);
```

Esta función no puede ser llamada directamente por el usuario. Cuando ocurre un error matemático, el sistema llama automáticamente a esta función. El usuario puede suministrar su propia función matherr para controlar los errores detectados por la librería matemática.

La función matherr() debe devolver un valor distinto de cero para indicar que se resuelve problema; en otro caso devueve 0.

Ejemplo 1:

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

  void main (void)
  {
    double x;

    log (-1);
    pow (0, -2);
    exp (-1000);
    sin (10e70);
    x = exp (1000);
    printf ("x = %lg", x);
  }
```

Ejemplo 2:

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

  int matherr (struct exception *e)
  {
    printf ("%s (%lg [, %lg]): %s.\n", e->name, e->arg1, e->arg2,
            e->type == DOMAIN ? "Error de dominio en argumento" :
            e->type == SING ? "Singularidad en argumento." :
            e->type == OVERFLOW ? "Error de rango de overflow" :
            e->type == UNDERFLOW ? "Error de rango de underflow" :
            e->type == TLOSS ? "Pérdida total de significancia" :
            e->type == PLOSS ? "Pérdida parcial de significancia" :
```

```
            "Error matemático");
    e->retval = e->type == OVERFLOW ? HUGE_VAL : 1;
    return 1;
  }

  void main (void)
  {
    double x;

    log (-1);
    pow (0, -2);
    exp (-1000);
    sin (10e70);
    x = exp (1000);
    printf ("x = %lg", x);
  }
```

La ejecución del programa del ejemplo 1 imprime lo siguiente (en la versión 2.0 de Borland C++):

```
  log: DOMAIN error
  pow: DOMAIN error
  exp: OVERFLOW error
  x = 1.79769e+308
```

La ejecución del programa del ejemplo 2 imprime lo siguiente (en cualquier versión de Turbo C):

```
  log (-1 [, 0]): Error de dominio en argumento.
  pow (0 [, -2]): Error de dominio en argumento.
  exp (-1000, [, 0]): Error de rango de underflow.
  sin (1e+71 [, 0]): Pérdida total de significancia.
  exp (1000 [, 0]): Error de rango de overflow.
  x = 1.79769e+308
```

Ver al final de esta ventana cómo es struct exception y el sinificado de los tipos enumerados y constantes simbólicas que aparecen en el segundo ejemplo.

---

modf — Descompone en parte entera y parte fraccionaria.

Sintaxis:

```
   double modf (double x, double *parte_entera);
```

La función modf() descompone x en sus partes entera y fraccionaria. Devuelve la parte fraccionaria y sitúa la parte entera en la variable apuntada por parte_entera.

---

poly — (TC) Genera un polinomio de los argumentos de esta función.

Sintaxis:

```
   double poly (double x, int grado, double coefs[]);
```

Devuelve el valor de un polinomio en x, de grado n, con coeficientes

```
  coefs[0], ..., coefs[n].
```

Por ejemplo: Si n = 4, el polinomio generado es

```
  (coefs[4] * x^4) + (coefs[3] * x^3) +
  (coefs[2] * x^2) + (coefs[1] * x^1) +
  (coefs[0])
```

Ejemplo:

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

  /* polinomio:  x**3 - 2x**2 + 5x - 1 */

  int main (void)
  {
    double array[] = { -1.0, 5.0, -2.0, 1.0 };
    double resultado;

    resultado = poly (2.0, 3, array);
    printf ("El polinomio: x**3 - 2.0x**2 + 5x - 1 en 2.0 es %lg.\n",
            resultado);
    return 0;
  }

  /* imprime:  El polinomio: x**3 - 2.0x**2 + 5x - 1 en 2.0 es 9.  */
```

---
pow Función potencia, x elevado a y.
=====

Sintaxis:
```
   double pow (double base, double exponente);
```

Devuelve base elevado a exponente. Se produce un error de dominio si base es 0 y exponente es menor o igual que 0. También puede ocurrir si base es negativo y exponente no es entero. Un desbordamiento produce un error de rango.

---
pow10 (TC) Función potencia, 10 elevado a p.
=======

Sintaxis:
```
   double pow10 (int p);
```

Devuelve 10 elevado a p.

---
sin Función seno.
=====

Sintaxis:
```
   double sin (double x);
```

Devuelve el seno de x. El valor de x debe darse en radianes.

---
sinh Función seno hiperbólico.

======

Sintaxis:
   double sinh (double x);

Devuelve el seno hiperbólico de x. El valor de x debe darse en radianes.

------
  sqrt   Calcula raíz cuadrada.
======

Sintaxis:
   double sqrt (double x);

Devuelve la raíz cuadrada de x. Si se llama con un número negativo se
produce un error de dominio.

-----
  tan   Tangente.
=====

Sintaxis:
   double tan (double x);

Devuelve la tangente de x. El valor de x debe darse en radianes.

------
  tanh   Tangente hiperbólica.
======

Sintaxis:
   double tanh (double x);

Devuelve la tangente hiperbólica de x.

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
****

  COMPLEX (struct)
==================

(TC) Representación de número complejo.

  struct complex {
    double x, y;
  };

Esta estructura es usada solamente por la función cabs().

Ver ejemplo de cómo usar esta estructura en la descripción de la función
cabs().

  EDOM (#define)
================

Código de error para errores de dominios matemáticos, es decir, cuando

el argumento de la función matemática está fuera del dominio.

Este valor es asignado a errno cuando se produce un error de rango.

  ERANGE (#define)
==========

Código de error para resultados fuera de rango.

Este valor es asignado a errno cuando se produce un error de rango.

  EXCEPTION (struct)
==========

(TC) El formato de información de error para las rutinas matemáticas.

```
  struct exception {
    int     type;
    char   *name;
    double  arg1, arg2, retval;
  };
```

Ver ejemplo en la descripción de la función matherr() para saber cómo usar esta estructura.

  HUGE_VAL (#define)

Valor de overflow para las funciones matemáticas.

Ver ejemplo en descripción de la función matherr() para ver cómo se puede usar.

  M_xxxx (#defines)

(TC) Los valores constantes para funciones logarítmicas.

```
  M_E        El valor de e.
  M_LOG2E    El valor de log(e).
  M_LOG10E   El valore log10(e).
  M_LN2      El valore log(2).
  M_LN10     El vvalo log(10).
```

  PI (#defines)
==========

(TC) Constantes comunes de ã.

```
  M_PI         ã
  M_PI_2       Un medio de ã                    (ã/2)
  M_PI_4       Un cuarto de ã                   (ã/4)
  M_1_PI       Uno divido por ã                 (1/ã)
```

```
  M_2_PI       Dos divido por ã                (2/ã)
  M_1_SQRTPI   Raíz cuadrada de ã              (ûã)
  M_2_SQRTPI   Mitad de la raíz cuadrada de ã  ((ûã)/2)
```

M_SQRTxx (#defines)

(TC) Valores constantes para raíces cuadradas de 2.

```
  M_SQRT2    Raíz cuadrada de 2               (û2)
  M_SQRT_2   Mitad de la raíz cuadrada de 2   (û2)/2
```

_mexcep (enum)

(TC) Estas constantes representan posibles errores matemáticos.

```
  DOMAIN      Error de dominio en argumento.
  SING        Singularidad en argumento.
  OVERFLOW    Error de rango de overflow.
  UNDERFLOW   Error de rango de underflow.
  TLOSS       Pérdida total de significancia.
  PLOSS       Pérdida parcial de significancia.
```

Ver ejemplo en descripción de la función matherr para saber cómo usar estas constantes.

partir aquí interesa que esté dentro del begint

FUNCIONES DE PANTALLA Y DE GRAFICOS

Las funciones de pantalla y de gráficos no están definidas por el estándar ANSI. Esto ocurre por una simple razón: son, por naturaleza, dependientes del entorno fijado y en gran parte no portables. Actualmente no hay una interfaz gráfica que sea ampliamente aceptada, ni hay un conjunto universal de órdenes de control de pantalla. Sin embargo, estos tipos de funciones son de importancia primordial cuando se crea software que requiere control de pantalla de alta resolución y de calidad de gráficos para el mercado de software. Las funciones descritas en las dos siguientes pantallas pertenecen al **Turbo C**.
Estas funciones se descomponen en dos grupos: aquellas que desempeñan funciones relacionadas con las pantallas de texto y aquellas que se refieren a los gráficos. Las funciones gráficos requieren la cabecera **<graphics.h>** y las funciones de pantalla requieren la cabecera **<conio.h>**.
Tanto en modo texto como en modo gráfico, la mayoría de las funciones trabajan con una ventana. Una ventana es un trozo rectangular de la pantalla que hace la función de pantalla completa. Por defecto, la ventana de trabajo es la pantalla completa; pero se puede cambiar con la función **window()** en modo texto y con la función **setviewport()** en modo gráfico. Por ejemplo, la sentencia gotoxy(2,2) hace que el cursor se posicione en la posición (2,2) relativa a la ventana actual, si la ventana actual es la pantalla entera, entonces se posicionará en la posición (2,2) de la pantalla.

FICHERO DE CABECERA CONIO.H (TC)

GLOSARIO:
****

cgets Lee una cadena de consola.

clreol Borra hasta final de línea en ventana de texto.

clrscr Borra ventana de texto.

cprintf Escribe salida formateada en la ventana de texto en la pantalla.

cputs Escribe una cadena en la ventana de texto en la pantalla.

cscanf Lee entrada formateada de consola.

delline Borra línea en ventana de texto.

getch and getche Lee carácter de consola, con eco a pantalla (getche), o sin eco (getch).

getpass Lee un password (palabra de paso).

gettext Copia texto de pantalla en modo texto a memoria.

gettextinfo Obtiene información de vídeo en modo texto.

gotoxy Posiciona cursor en ventana de texto.

highvideo Selecciona caracteres de texto en alta intensidad.

insline Inserta línea en blanco en ventana de texto en la posición actual del cursor.

kbhit Chequea para ver si se ha pulsado alguna tecla, es decir, para ver si hay alguna tecla disponible en el buffer de teclas.

lowvideo Selecciona salida de caracteres en ventana de texto en baja intensidad.

movetext Copia texto en pantalla de un rectángulo a otro (en modo texto).

normvideo Selecciona caracteres en intensidad normal.

putch Escribe un carácter en la ventana de texto sobre en la pantalla.

---
  puttext   Copia texto de memoria a la pantalla.
=========

  _setcursortype   Selecciona la apariencia del cursor.
==========
---
  textattr   Pone los atributos de texto para las funciones de ventana de
==========   texto.
---
  textbackground   Selecciona nuevo color de fondo de los caracteres en
==========   modo texto.
---
  textcolor   Selecciona nuevo color de texto de los caracteres en modo
==========   texto.
---
  textmode   Cambia modo de pantalla (en modo texto).
==========
---
  ungetch   Devuelve un carácter al teclado.
=========
---
  wherex   Devuelve posición horizontal del cursor dentro de la ventana
========   de texto corriente.
---
  wherey   Devuelve posición vertical del cursor dentro de la ventana
========   de texto corriente.
---
  window   Define ventana activa en modo texto.
========

  COLORS (enum)   Colores/atributos del vídeo CGA estándar.
==========

  BLINK   Sirve para sumar a color de fondo cuando queremos escribir
=======   caracteres parpadeantes.
---
  directvideo (variable gloval)   Controla salida de vídeo.
==========

  TEXT_INFO (struct)   Información de ventana de texto actual.
==========
---
  text_modes (enum)   Modos de vídeo estándar.
==========

  _wscroll (variable global)   Controla el scroll en las ventanas de texto.
==========

FUNCIONES:
****

---
  cgets   Lee una cadena de consola.
=======

Sintaxis:
   char *cgets (char *cad);

cad[0] debe contener la longitud máxima de la cadena a ser leída. A la
vuelta, cad[1] contiene el número de caracteres leídos realmente. La cadena
empieza en cad[2]. La función devuelve &cad[2].

----
  clreol   Borra hasta final de línea en ventana de texto.
========

Sintaxis:
   void clreol (void);

----
  clrscr   Borra ventana de texto.
========

Sintaxis:
   void clrscr(void);

----
  cprintf   Escribe salida formateada en la ventana de texto en la pantalla.
=========

Sintaxis:
   int cprintf (const char *formato [, argumento,...]);

Devuelve el número de bytes escritos.

----
  cputs   Escribe una cadena en la ventana de texto en la pantalla.
=======

Sintaxis:
   int cputs (const char *cad);

Devuelve el último carácter escrito.

----
  cscanf   Lee entrada formateada de consola.
========

Sintaxis:
   int cscanf (char *formato [, direccion, ...]);

Devuelve el número de campos procesados con éxito. Si una función intenta
leer en final de fichero, el valor devuelto es EOF.

----
  delline   Borra línea en ventana de texto.
=========

Sintaxis:
   void delline (void);

----
  getch and getche   Lee carácter de consola, con eco a pantalla (getche),
==================   o sin eco (getch).

Sintaxis:
   int getch (void);
   int getche (void);

Ambas funciones devuelven el carácter leído. Los caracteres están disponibles inmediatamente (no hay buffer de líneas completas).

Las teclas especiales tales como las teclas de función y las teclas de los cursores están representadas por una secuencia de dos caracteres: un carácter cero seguido por el código de exploración para la tecla presionada.

---
getpass Lee un password (palabra de paso).

Sintaxis:
```
char *getpass (const char *prompt);
```

El valor devuelto es un puntero a una cadena estática que es sobreescrita en cada llamada.

---
gettext Copia texto de pantalla en modo texto a memoria.

Sintaxis:
```
int gettext (int izq, int ar, int der, int ab, void *destino);
```

Las coordenadas son absolutas, no son relativas a la ventana actual. La esquina superior izquierda es (1,1). Devuelve un valor distinto de cero si tiene éxito.

---
gettextinfo Obtiene información de vídeo en modo texto.

Sintaxis:
```
void gettextinfo (struct text_info *r);
```

El resultado es devuelto en r.

---
gotoxy Posiciona cursor en ventana de texto.

Sintaxis:
```
void gotoxy (int x, int y);
```

---
highvideo Selecciona caracteres de texto en alta intensidad.

Sintaxis:
```
void highvideo (void);
```

Afecta a subsecuentes llamadas a funciones de ventana de texto tales como putch() y cprintf().

---
insline Inserta línea en blanco en ventana de texto en la posición actual del cursor.

Sintaxis:
   void insline (void);

Las líneas por debajo de la posición del cursor son subidas una línea hacia arria y la última línea se pierde.

---
kbhit Chequea para ver si se ha pulsado alguna tecla, es decir, para ver si hay alguna tecla disponible en el buffer de teclas.
=======

Sintaxis:
   int kbhit (void);

Si una tecla está disponible, kbhi() devuelve un entero distino de cero; si no es así, devuelve 0.

---
lowvideo Selecciona salida de caracteres en ventana de texto en baja intensidad.
==========

Sintaxis:
   void lowvideo (void);

Afecta a la salida subsiguiente escritas con funcines de ventana de texto tales como putch() y cprintf().

---
movetext Copia texto en pantalla de un rectángulo a otro (en modo texto).
==========

Sintaxis:
   int movetext (int izq, int ar, int der, int ab, int izqdest, int ardest);

Las coordenadas son relativas a la esquina superior izquierda de la pantalla (1,1).

Devuelve un valor distinto de cero si la operación tuvo éxito.

---
normvideo Selecciona caracteres en intensidad normal.
==========

Sintaxis:
   void normvideo (void);

Afecta a la salida subsecuente de funciones de ventana de texto tales como putch() y cprintf().

---
putch Escribe un carácter en la ventana de texto sobre en la pantalla.
=======

Sintaxis:
   int putch (int ch);

Usa el color y atributo de visualización actuales.

Devuelve ch, el carácter visualizado. En caso de error, devuelve EOF.

---
  puttext   Copia texto de memoria a la pantalla.
=========

Sintaxis:
   int puttext (int izq, int ar, int der, int ab, void *fuente);

Las coordenadas son coordenadas de pantalla absoluta, no relativas a la
ventana actual. La esquina superior izquierda es (1,1).

Devuelve un valor distinto de cero si tiene éxito.

  _setcursortype   Selecciona la apariencia del cursor.
===============

Sintaxis:
   void _setcursortype (int t_cur);

El tipo de cursor t_cur puede ser

  _NOCURSOR   el cursor no se ve
  _SOLIDCURSOR   bloque sólido
  _NORMALCURSOR   cursor compuesto de varias líneas inferiores

---
  textattr   Pone los atributos de texto para las funciones de ventana de
==========   texto.

Sintaxis:
   void textattr(int newattr);

---
  textbackground   Selecciona nuevo color de fondo de los caracteres en
===============   modo texto.

Sintaxis:
   void textbackground (int nuevocolor);

---
  textcolor   Selecciona nuevo color de texto de los caracteres en modo
==========   texto.

Sintaxis:
   void textcolor (int nuevocolor);

---
  textmode   Cambia modo de pantalla (en modo texto).
==========

Sintaxis:
   void textmode (int nuevomodo);

No sirve para cambiar de modo gráfico a modo texto.

---
  ungetch   Devuelve un carácter al teclado.

=========

Sintaxis:
   int ungetch (int ch);

La próxima llamada a getch() o cualquier otra función de entrada de consola
devolverá ch.

Devuelve el carácter ch si tiene éxito. Si devuelve EOF, indica que hubo
un error.

----
  wherex   Devuelve posición horizontal del cursor dentro de la ventana
========   de texto corriente.

Sintaxis:
   int wherex (void);

Devuelve un entero en el rango de 1 a 80.

----
  wherey   Devuelve posición vertical del cursor dentro de la ventana
========   de texto corriente.

Sintaxis:
   int wherey (void);

Devuelve un entero en el rango de 1 a 25.

----
  window   Define ventana activa en modo texto.
========

Sintaxis:
   void window (int izq, int ar, int der, int ab);

La esquina superior izquierda de la pantalla es (1,1).

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
****

  COLORS (enum)
===============

Colores/atributos del vídeo CGA estándar.

BLACK	DARKGRAY
BLUE	LIGHTBLUE
GREEN	LIGHTGREEN
CYAN	LIGHTCYAN
RED	LIGHTRED
MAGENTA	LIGHTMAGENTA
BROWN	YELLOW
LIGHTGRAY	WHITE

  BLINK
=======

Esta constante se le suma a color de fondo para visualizar caracteres parpadeantes en modo texto.

---

directvideo (variable gloval)

Controla salida de vídeo.

```
int directvideo;
```

La variable global directvideo controla si la salida en consola de nuestro programa va directamente a la RAM de vídeo (directvideo = 1) o va vía llamadas a la ROM BIOS (directvideo = 0).

El valor por defecto es directvideo = 1 (salida a consola va directamente a RAM de vídeo).

Para usar directvideo = 1, el hardware de vídeo de nuestro sistema debe ser idéntico a los adaptadores de visualización de IBM.

Poner directvideo = 0 permite que la salida a consola trabaje con cualquier sistema que sea compatible a nivel BIOS con IBM.

TEXT_INFO (struct)

Información de ventana de texto corriente.

```
struct text_info
  {
    unsigned char  winleft,   wintop;
    unsigned char  winright,  winbottom;
    unsigned char  attribute, normattr;
    unsigned char  currmode;
    unsigned char  screenheight;
    unsigned char  screenwidth;
    unsigned char  curx, cury;
  };
```

---

text_modes (enum)

Modos de vídeo estándar.

```
LASTMODE   BW80
BW40       C80
C40        C4350
MONO
```

_wscroll (variable global)

Habilita o desahibilita el scroll en funciones de E/S de consola.

```
extern int_wscroll;
```

La variable global _wscroll es un flag (bandera) de E/S de consola. Podemos usarla para dibujar entre los ojos de una ventana sin provocar un scroll de la pantalla.

El valor por defecto es _wscroll = 1 (scroll permitido).

NOTA: En el ejemplo 3 de esta lección se muestra cómo se pueden utilizar estas funciones en un programa

FICHERO DE CABECERA GRAPHICS.H (TC)

GLOSARIO:
****

-----
  arc   Dibuja un arco.
=====
-----
  bar   Dibuja una barra.
=====
----
  bar3d   Dibuja una barra en 3-D.
=======
----
  circle   Dibuja un círculo en (x,y) con el radio dado.
========
----
  cleardevice   Borra la pantalla gráfica.
=============
----
  clearviewport   Borra el viewport corriente.
===============
----
  closegraph   Cierra el sistema gráfico.
============
----
  detectgraph   Determina el controlador y el modo gráfico a usar
=============   chequenado el hardware
----
  drawpoly   Dibuja un polígono.
==========
----
  ellipse   Dibuja un arco elíptico.
=========
----
  fillellipse   Dibuja y rellena una elipse.
=============
----
  fillpoly   Dibuja y rellena un polígono.
==========
----
  floodfill   Rellena una región definida.
===========
----
  getarccoords   Obtiene las coordenadas de la última llamada a arc.
==============
----
  getaspectratio   Obtiene la cuadratura para el modo gráfico corriente.
================

```
------------
  getbkcolor    Devuelve el color de fondo actual.
============
----------
  getcolor    Devuelve el color de dibujar actual.
==========
-------------------
  getdefaultpalette    Devuelve la estructura de definición de paleta.
===================
---------------
  getdrivername    Devuelve un puntero al nombre del controlador gráfico
===============   actual.
----------------
  getfillpattern    Copia un patrón de relleno definido por el usuario en
================   memoria.
-----------------
  getfillsettings    Obtiene información acerca del patrón y color de
=================   relleno actual.
--------------
  getgraphmode    Devuelve el modo gráfico actual.
==============
----------
  getimage    Salva una imagen de la región especificada en memoria.
==========
-----------------
  getlinesettings    Obtiene el estilo, patrón y grosor actual de línea.
=================
-------------
  getmaxcolor    Devuelve el valor del color máximo.
=============
------------
  getmaxmode    Devuelve el número de modo gráfico máximo para el
============   controlador corriente.
---------------------
  getmaxx and getmaxy    Devuelve la coordenada x o y máxima de pantalla.
=====================
-------------
  getmodename    Devuelve un puntero a una cadena que contiene el nombre
=============   del modo gráfico especificado.
--------------
  getmoderange    Obtiene el rango de los modos para un controlador
==============   gráfico dado.
------------
  getpalette    Obtiene información acerca de la paleta actual.
============
----------------
  getpalettesize    Devuelve el número de entradas de la paleta.
================
----------
  getpixel    Obtiene el color de un pixel especificado.
==========
-----------------
  gettextsettings    Obtiene información acerca de las características del
=================   texto gráfico actual.
-----------------
  getviewsettings    Obtiene información acerca del viewport actual.
=================
------
  getx    Devuelve la coordenada x de la posición actual.
======
------
  gety    Devuelve la coordenada y de la posición actual.
======
--------------
```

graphdefaults Pone todos los valores gráficos a sus valores por defecto.
---
grapherrormsg Devuelve un puntero a una cadena con el mensaje de error.

_graphfreemem Manejador de usuario para desasignar memoria gráfica.

_graphgetmem Manejador de usuario para asignar memoria gráfica.
---
graphresult Devuelve un código de error para la última operación gráfica que no tuvo éxito.
---
imagesize Devuelve el número de bytes requeridos para almacenar una imagen.
---
initgraph Inicializa el sistema gráfico.
---
installuserdriver Instala un nuevo controlador de dispositivo a la tabla de controladores de dispotivo BGI.
---
installuserfont Carga un fichero de estilo de caracteres (.CHR) que no está dentro del sistema BGI.
---
line Dibuja una línea entre dos puntos especificados.
---
linerel Dibuja una línea a una distancia relativa a la posición actual.
---
lineto Dibuja una línea desde la posición actual hasta (x,y).
---
moverel Cambia la posición actual a una distancia relativa.
---
moveto Cambia la posición actual a (x,y).
---
outtext Visualiza una cadena en el viewport (modo gráfico).
---
outtextxy Visualiza una cadena en el lugar especificado (modo gráfico).
---
pieslice Dibuja y rellena un sector de círculo.
---
putimage Escribe una imagen en la pantalla.
---
putpixel Escribe un pixel en el punto especificado.
---
rectangle Dibuja un rectángulo (modo gráfico).
---
registerbgidriver Registra controlador gráfico en el enlazado.
---
registerbgifont Registra estilo de texto en el enlazado.

```
=================
----------------
  restorecrtmode    Restaura modo de pantalla previa a entrar al modo
===============   gráfico.
--------
  sector    Dibuja y rellena un sector elíptico.
========
---------------
  setactivepage    Pone la página activa para la salida gráfica.
===============
---------------
  setallpalette    Cambia los colores de todas las paletas.
===============
----------------
  setaspectratio    Pone la cuadratura gráfica
================
------------
  setbkcolor    Pone el color de fondo actual usando la paleta.
============
----------
  setcolor    Pone el color actual para dibujar.
==========
----------------
  setfillpattern    Selecciona un patrón de relleno definido por el usuario.
================
--------------
  setfillstyle    Pone el patrón y color de relleno.
==============
-----------------
  setgraphbufsize    Cambia el tamaño del buffer gráfico interno.
=================
--------------
  setgraphmode    Pone el sistema en modo gráfico y borra la pantalla.
==============
--------------
  setlinestyle    Pone el estilo de línea, anchura y patrón actual.
==============
------------
  setpalette    Cambia un color de la paleta.
============
---------------
  setrgbpalette    Define colores para la tarjeta gráfica IBM-8514.
===============
----------------
  settextjustify    Pone justificación de texto para modo gráfico.
================
--------------
  settextstyle    Pone las características actuales del texto.
==============
-----------------
  setusercharsize    Factor de amplificacion de los caracteres definidos
=================   por el usuario para los estilos de caracteres.
-------------
  setviewport    Pone el viewport actual para salida gráfica.
=============
---------------
  setvisualpage    Pone el número de página gráfica visual.
===============
--------------
  setwritemode    Pone el modo de escritura para el dibujo de líneas en modo
==============   gráfico.
------------
  textheight    Devuelve la altura de una cadena en pixels.
============
```

textwidth	Devuelve la anchura de una cadena en pixels.
ARCCOORDSTYPE (struct)	Coordenadas de la última llamada a arc().
EGA_xxxx (#defines)	Colores para las funciones setpalette() y setallpalette().
fill_patterns (enum)	Patrones de relleno para las funciones getfillsettings() y setfillstyle().
FILLSETTINGSTYPE (struct)	Usado para obtener los valores de relleno actuales por la función getfillsettings().
font_names (enum)	Nombres de estilos de texto.
graphics_drivers (enum)	Controladores gráficos BGI.
graphics_errors (enum)	Códigos de error devueltos por graphresult().
graphics_modes (enum)	Modos gráficos para cada controlador BGI.
line_styles (enum)	Estilos de línea para las funciones getlinesettings() y setlinestyle().
line_widths (enum)	Anchuras de las líneas para las funciones getlinesettings() y setlinestyle().
MAXCOLORS (#define)	Define el número máximo de colores para la estructura palettetype.
PALETTETYPE (struct)	Información de la paleta para el controlador gráfico actual utilizado por las funciones getpalette(), setpalette() y setallpalette().
POINTTYPE (struct)	Coordenadas de un punto.
putimage_ops (enum)	Operadores para putimage().
text_just (enum)	Justificación horizontal y vertical para la función settextjustify().
*_DIR (Dirección) (#defines)	Dirección para la salida gráfica.
TEXTSETTINGSTYPE (struct)	Usado para obtener los valores de texto actual por la función gettextsettings().
USER_CHAR_SIZE (#define)	Tamaño de los caracteres definidos por el usuario.
VIEWPORTTYPE (struct)	Estructura usada para obtener información sobre el viewport actual por la función getviewsettings().

FUNCIONES:
****

```
-----
  arc    Dibuja un arco.
=====

Sintaxis:
   void far arc (int x, int y, int ang_comienzo, int ang_final, int radio);

(x,y) es el punto central; ang_comienzo y ang_final son los ángulos de
comienzo y final en grados; radio es el radio.


-----
  bar    Dibuja una barra.
=====

Sintaxix:
   void far bar (int izq, int ar, int der, int ab);


-------
  bar3d    Dibuja una barra en 3-D.
=======

Sintaxis:
   void far bar3d (int izq, int ar, int der, int ab, int profundidad,
                   int flag_de_encima);

Si flag_de_encima es 0 no se dibuja la cara superior de la barra.


--------
  circle    Dibuja un círculo en (x,y) con el radio dado.
========

Sintaxis:
   void far circle (int x, int y, int radio);


-------------
  cleardevice    Borra la pantalla gráfica.
=============

Sintaxis:
   void far cleardevice (void);


---------------
  clearviewport    Borra el viewport corriente.
===============

Sintaxis:
   void far clearviewport (void);


------------
  closegraph    Cierra el sistema gráfico.
============

Sintaxis:
   void far closegraph (void);


-------------
```

```
  detectgraph    Determina el controlador y el modo gráfico a usar
=============   chequenado el hardware

Sintaxis:
   void far detectgraph (int far *graphdriver, int far *graphmode);


----------
  drawpoly    Dibuja un polígono.
==========

Sintaxis:
   void far drawpoly (int numero_de_puntos, int far *puntos_de_poligono);

*puntos_de_poligono apunta a numero_de_puntos pares de valores. Cada par
da los valores de x e y para un punto del polígono.


---------
  ellipse    Dibuja un arco elíptico.
=========

Sintaxis:
   void far ellipse (int x, int y, int ang_comienzo, int ang_final,
                     int radiox, int radioy);

(x,y) es el punto central; ang_comienzo y ang_final son los ángulos de
comienzo y final en grados; radiox y radioy son los radios horizontal y
vertical.


-------------
  fillellipse    Dibuja y rellena una elipse.
=============

Sintaxis:
   void far fillellipse (int x, int y, int radiox, int radioy);

Usa (x,y) como el punto central y rellena el arco usando el patrón de
relleno actual; radiox y radioy son los radios horizontal y vertical.


----------
  fillpoly    Dibuja y rellena un polígono.
==========

Sintaxis:
   void far fillpoly(int numpoints, int far *polypoints[]);

*puntos_de_poligono apunta a numero_de_puntos pares de valores. Cada par
da los valores de x e y para un punto del polígono.


-----------
  floodfill    Rellena una región definida.
===========

Sintaxis:
   void far floodfill (int x, int y, int color_borde);

(x,y) es un punto que reside dentro de la región a rellenar.


--------------
```

```
  getarccoords    Obtiene las coordenadas de la última llamada a arc.
==============

Sintaxis:
   void far getarccoords (struct arccoordstype far *coords_arc);


----------------
  getaspectratio    Obtiene la cuadratura para el modo gráfico corriente.
================

Sintaxis:
   void far getaspectratio (int far *cuadx, int far *cuady);

cuadx debería ser 10000. Usa cuadx=10000 cuando los pixels son cuadrados
(VGA); <10000 para pixels altos.


------------
  getbkcolor    Devuelve el color de fondo actual.
============

Sintaxis:
   int far getbkcolor (void);


----------
  getcolor    Devuelve el color de dibujar actual.
==========

Sintaxis:
   int far getcolor (void);


-------------------
  getdefaultpalette    Devuelve la estructura de definición de paleta.
===================

Sintaxis:
   struct palettetype *far getdefaultpalette (void);

Devuelve un puntero a la estructura de paleta por defecto para el
controlador actual inicializado mediante una llamada a initgraph().


---------------
  getdrivername    Devuelve un puntero al nombre del controlador gráfico
===============   actual.

Sintaxis:
   char *far getdrivername(void);

El puntero far devuelto apunta a una cadena que identifica el controlador
gráfico actual.


----------------
  getfillpattern    Copia un patrón de relleno definido por el usuario en
================   memoria.

Sintaxis:
   void far getfillpattern (char far *patron);
```

```
-----------------
  getfillsettings    Obtiene información acerca del patrón y color de
=================   relleno actual.

Sintaxis:
   void far getfillsettings (struct fillsettingstype far *fillinfo);


--------------
  getgraphmode    Devuelve el modo gráfico actual.
==============

Sintaxis:
   int far getgraphmode (void);

Antes de llamar a getgraphmode() se debe llamar a initgraph() o
setgraphmode().


----------
  getimage    Salva una imagen de la región especificada en memoria.
==========

Sintaxis:
   void far getimage(int izq, int ar, int der, int ab, void far *bitmap);


-----------------
  getlinesettings    Obtiene el estilo, patrón y grosor actual de línea.
=================

Sintaxis:
   void far getlinesettings (struct linesettingstype far *infolinea);


-------------
  getmaxcolor    Devuelve el valor del color máximo.
=============

Sintaxis:
   int far getmaxcolor(void);


------------
  getmaxmode    Devuelve el número de modo gráfico máximo para el
============   controlador corriente.

Sintaxis:
   int far getmaxmode (void);

El modo mínimo es 0.


---------------------
  getmaxx and getmaxy    Devuelve la coordenada x o y máxima de pantalla.
=====================

Sintaxis:
   int far getmaxx(void);
   int far getmaxy(void);


-------------
  getmodename    Devuelve un puntero a una cadena que contiene el nombre
```

=============   del modo gráfico especificado.

Sintaxis:
   char * far getmodename (int numero_de_modo);

El puntero devuelto apunta al nombre (cadena) del modo especificado por
numero_de_modo.

----
  getmoderange   Obtiene el rango de los modos para un controlador
==============   gráfico dado.

Sintaxis:
   void far getmoderange (int controlador_grafico, int far *mode_bajo,
                          int far *mode_alto);

----
  getpalette   Obtiene información acerca de la paleta actual.
============

Sintaxis:
   void far getpalette (struct palettetype far *paleta);

----
  getpalettesize   Devuelve el número de entradas de la paleta.
================

Sintaxis:
   int far getpalettesize (void);

Devuelve el número de entradas de la paleta permitidas para el modo de
controlador gráfico actual.

----
  getpixel   Obtiene el color de un pixel especificado.
==========

Sintaxis:
   unsigned far getpixel (int x, int y);

----
  gettextsettings   Obtiene información acerca de las características del
=================   "texto gráfico actual.

Sintaxis:
   void far gettextsettings (struct textsettingstype far *texttypeinfo);

----
  getviewsettings   Obtiene información acerca del viewport actual.
=================

Sintaxis:
   void far getviewsettings (struct viewporttype far *viewport);

------
  getx   Devuelve la coordenada x de la posición actual.
======

Sintaxis:
   int far getx (void);

El valor es relativo al viewport.

---
  gety   Devuelve la coordenada y de la posición actual.
======

Sintaxis:
   int far gety (void);

El valor es relativo al viewport.

---
  graphdefaults   Pone todos los valores gráficos a sus valores por defecto.
===============

Sintaxis:
   void far graphdefaults (void);

---
  grapherrormsg   Devuelve un puntero a una cadena con el mensaje de error.
===============

Sintaxis:
   char *far grapherrormsg (int codigo_de_error);

Devuelve un puntero a una cadena asociada con el valor devuelvto por
graphresult().

  _graphfreemem   Manejador de usuario para desasignar memoria gráfica.
===============

   void far _graphfreemem (void far *ptr, unsigned tamanio);

Esta función es llamada por las rutinas que hay en la biblioteca gráfica
para liberar memoria. Podemos controlar esta asignación de memoria sumi-
nistrando nuestras propias funciones _graphgetmem() y _graphfreemen().

  _graphgetmem   Manejador de usuario para asignar memoria gráfica.
==============

Sintaxis:
   void far * far _graphgetmem (unsigned tamanio);

Esta función es llamada por las rutinas que hay en la biblioteca gráfica
para asignar memoria. Podemos controlar esta asignación de memoria sumi-
nistrando nuestras propias funciones _graphgetmem() y _graphfreemen().

---
  graphresult   Devuelve un código de error para la última operación
=============   gráfica que no tuvo éxito.

Sintaxis:
   int far graphresult (void);

Devuelve el código de error para la última operación gráfica que informó de un error y pone el nivel de error a grOK.

---

imagesize — Devuelve el número de bytes requeridos para almacenar una imagen.

Sintaxis:

```
unsigned far imagesize (int izq, int ar, int der, int ab);
```

Si el tamaño requerido para la imagen seleccionada es mayor o igual que 64K - 1 bytes, imagesize() devuelve 0xFFFF.

---

initgraph — Inicializa el sistema gráfico.

Sintaxis:

```
void far initgraph (int far *controlador_grafico, int far *modo_grafico,
                    char far *path_para_controlador);
```

NOTA: El parámetro paht_para_controlador usa la sintaxis:

```
"..\\bgi\\drivers"
```

donde
- bgi\drivers el el nombre de directorio donde buscar los controladores
- el parámetro está encerrado entre comillas
- el path para los ficheros de controladores gráficos incluyen dos barras invertidas

---

installuserdriver — Instala un nuevo controlador de dispositivo a la tabla de controladores de dispotivo BGI.

Sintaxis:

```
int far installuserdriver (char far *nombre, int huge (*detect) (void));
```

El parámetro nombre es el nombre del nuevo fichero de controlador de dispositivo (.BGI) y detect es un puntero a una función de autodetección opcional que puede acompañar al nuevo controlador. Esta función de autodetección no tiene ningún parámetro y devuelve un valor entero.

---

installuserfont — Carga un fichero de estilo de caracteres (.CHR) que no está dentro del sistema BGI.

Sintaxis:

```
int far installuserfont (char far *nombre);
```

El parámetro nombre es el nombre del fichero que contiene las características del nuevo tipo de carácter en modo gráfico.

Al mismo tiempo pueden ser instalados hasta 20 estilos de caracteres.

---

line — Dibuja una línea entre dos puntos especificados.

```
Sintaxis:
   void far line (int x1, int y1, int x2, int y2);

Dibuja una línea desde (x1,y1) hasta (x2,y2) usando el color, estilo de
línea y grosor actuales.


---------
  linerel    Dibuja una línea a una distancia relativa a la posición actual.
=========

Sintaxis:
   void far linerel (int dx, int dy);

Usa el color, estilo de línea y grosor actual.


--------
  lineto    Dibuja una línea desde la posición actual hasta (x,y).
========

Sintaxis:
   void far lineto (int x, int y);


---------
  moverel    Cambia la posición actual a una distancia relativa.
=========

Sintaxis:
   void far moverel (int dx, int dy);


--------
  moveto    Cambia la posición actual a (x,y).
========

Sintaxis:
   void far moveto (int x, int y);


---------
  outtext    Visualiza una cadena en el viewport (modo gráfico).
=========

Sintaxis:
   void far outtext (char far *cadena_de_texto);


-----------
  outtextxy    Visualiza una cadena en el lugar especificado (modo gráfico).
===========

Sintaxis:
   void far outtextxy (int x, int y, char far *cadena_de_texto);


----------
  pieslice    Dibuja y rellena un sector de círculo.
==========

Sintaxis:
   void far pieslice (int x, int y, int ang_comienzo, int ang_final,
```

```
                      int radio);


----------
  putimage    Escribe una imagen en la pantalla.
==========

Sintaxis:
   void far putimage (int izq, int ar, void far *bitmap, int op);

bitmap apunta a un mapa de bits, normalmente creado por la función
getimage(). El valor op especifica cómo se combina la imagen con el
contenido actual del área en (izq,ar).


----------
  putpixel    Escribe un pixel en el punto especificado.
==========

Sintaxis:
   void far putpixel (int x, int y, int color);


-----------
  rectangle    Dibuja un rectángulo (modo gráfico).
===========

Sintaxis:
   void far rectangle (int izq, int ar, int der, int ab);

Una el estilo, grosor y color de línea actual.


-------------------
  registerbgidriver    Registra controlador gráfico en el enlazado.
===================

Sintaxis:
   int registerbgidriver (void (*driver) (void));

Informa al sistema gráfico que el controlador dispositivo apuntador por
driver fue incluido en tiempo de enlazado.


-----------------
  registerbgifont    Registra estilo de texto en el enlazado.
=================

Sintaxis:
   int registerbgifont (void (*font) (void));

Informa al sistema gráfico que el estilo de texto apuntado por font fue
incluido en tiemo de enlazado.


----------------
  restorecrtmode    Restaura modo de pantalla previa a entrar al modo
================   gráfico.

Sintaxis:
   void far restorecrtmode (void);


--------
```

sector Dibuja y rellena un sector elíptico.

Sintaxis:
```
   void far sector (int x, int y, int ang_comienzo, int ang_final,
                    int radiox, int radioy);
```

x e y definen el punto central; ang_comienzo y ang_final definen los ángulos de comienzo y final; radiox y radioy son los radios horizontal y vertical.

El sector es dibujado con el color activo y es rellenado con el color y patrón de relleno actual.

---

setactivepage Pone la página activa para la salida gráfica.

Sintaxis:
```
   void far setactivepage (int pagina);
```

Las salidas gráficas siguientes a la llamada a esta función irán a la página de visualización especificada. Esta página puede no ser la página visual que es la que actualmente está visualizada.

---

setallpalette Cambia los colores de todas las paletas.

Sintaxis:
```
   void far setallpalette (struct palettetype far *paleta);
```

---

setaspectratio Pone la cuadratura gráfica

Sintaxis:
```
   void far setaspectratio (int cuadx, int cuady);
```

cuadx debería ser 10000. Usa cuadx=10000 cuando los pixels son cuadrados (VGA); <10000 para pixels altos.

---

setbkcolor Pone el color de fondo actual usando la paleta.

Sintaxis:
```
   void far setbkcolor(int color);
```

---

setcolor Pone el color actual para dibujar.

Sintaxis:
```
   void far setcolor (int color);
```

---

setfillpattern Selecciona un patrón de relleno definido por el usuario.

```
================

Sintaxis:
   void far setfillpattern (char far *patron_usuario, int color);

El parámetro patron_usuario apunta a un área de 8 bytes donde se encuentra
el patrón de bits 8 por 8.


--------------
  setfillstyle    Pone el patrón y color de relleno.
==============

Sintaxis:
   void far setfillstyle (int patron, int color);

El parámetro patron identifica un patrón predefinido.

Para poner un patrón de relleno definido por el usuario, llamar a la
función setfillpattern().


-----------------
  setgraphbufsize    Cambia el tamaño del buffer gráfico interno.
=================

Sintaxis:
   unsigned far setgraphbufsize (unsigned tambuf);

Esta función debe ser llamada antes de llamar a la función initgraph().
Devuelve el tamaño previo del buffer interno.


--------------
  setgraphmode    Pone el sistema en modo gráfico y borra la pantalla.
==============

Sintaxis:
   void far setgraphmode (int modo);


--------------
  setlinestyle    Pone el estilo de línea, anchura y patrón actual.
==============

Sintaxis:
   void far setlinestyle (int estilo_de_linea, unsigned patron_usuario,
                          int grosor);

Pone el estilo y grosor para el dibujo de líneas en funciones gráficas.


------------
  setpalette    Cambia un color de la paleta.
============

Sintaxis:
   void far setpalette (int num_de_color, int color);


---------------
  setrgbpalette    Define colores para la tarjeta gráfica IBM-8514.
===============
```

Sintaxis:
   void far setrgbpalette (int numcolor, int rojo, int verde, int azul);

El parámetro numcolor es la entrada de la paleta a ser cargada (número entre 0 y 255). Los parámetros rojo, verde y azul definen los colores componentes.

Sólo el byte menos significativo de estos valores es usado, y sólo sus 6 bits más significativos son cargados en la paleta.

---
settextjustify Pone justificación de texto para modo gráfico.

Sintaxis:
   void far settextjustify (int horiz, int vert);

Afecta a la salida de texto con outtext(), etc. El texto es justificado horizontalmente y verticalmente.

Los valores para horiz y vert son los siguientes:

Param	Nombre	Val	Cómo justificar
horiz	LEFT_TEXT	(0)	izquierda <
	CENTER_TEXT	(1)	> centrar texto <
	RIGHT_TEXT	(2)	> derecha
vert	BOTTOM_TEXT	(0)	de abajo a arriba
	CENTER_TEXT	(1)	centrar texto
	TOP_TEXT	(2)	de arriba a abajo

---
settextstyle Pone las características actuales del texto.

Sintaxis:
  void far settextstyle (int estilo, int direccion, int tamanio_de_caracter);

---
setusercharsize Factor de amplificacion de los caracteres definidos por el usuario para los estilos de caracteres.

Sintaxis:
   void far setusercharsize (int multx, int divx, int multy, int divy);

---
setviewport Pone el viewport actual para salida gráfica.

Sintaxis:
   void far setviewport(int izq, int ar, int der, int ab, int clip);

---
setvisualpage Pone el número de página gráfica visual.

Sintaxis:
   void far setvisualpage (int pagina);

Algunos adaptadores gráficos tienen más de una página de memoria. La página visual es la página que está actualmente visualizada en la pantalla. Las funciones gráficas escriben en la página activa, definida por setactivepage().

---
setwritemode Pone el modo de escritura para el dibujo de líneas en modo
========== gráfico.

Sintaxis:
```
   void far setwritemode (int modo);
```

Si el modo es 0, las líneas sobreescriben el contenido actual de la pantalla.

Si el modo es 1, una operación exclusive OR (XOR) es ejecutada entre los pixels de la línea y los puntos correspondientes sobre la pantalla.

---
textheight Devuelve la altura de una cadena en pixels.
==========

Sintaxis:
```
   int far textheight (char far *cadena_con_texto);
```

La función textwidth() es útil para ser usadas con funciones gráficas tales como outtext().

---
textwidth Devuelve la anchura de una cadena en pixels.
==========

Sintaxis:
```
   int far textwidth (char far *cadena_con_texto);
```

La función textwidth() es útil para ser usadas con funciones gráficas tales como outtext().

CONTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
****

ARCCOORDSTYPE (struct)
==========

Usado por la función getarccords() para obtener las coordenadas de la última llamada a arc().

```
  struct arccoordstype
    {
      int  x, y;               /* punto central */
      int  xstart, ystart;  /* posición inicial */
      int  xend, yend;        /* posición final */
    };
```

EGA_xxxx (#defines)
==========

Colores para las funciones setpalette() y setallpalette().

```
  EGA_BLACK         EGA_DARKGRAY
  EGA_BLUE          EGA_LIGHTBLUE
  EGA_GREEN         EGA_LIGHTGREEN
  EGA_CYAN          EGA_LIGHTCYAN
  EGA_RED           EGA_LIGHTRED
  EGA_MAGENTA       EGA_LIGHTMAGENTA
  EGA_BROWN         EGA_YELLOW
  EGA_LIGHTGRAY     EGA_WHITE
```

---

fill_patterns (enum)

Patrones de relleno para las funciones getfillsettings() y setfillstyle().

EMPTY_FILL	Usa color de fondo
SOLID_FILL	Usa color de relleno sólido
LINE_FILL	Relleno con ---
LTSLASH_FILL	Relleno con ///
SLASH_FILL	Relleno con líneas gruesas ///
BKSLASH_FILL	Relleno con líneas gruesas \\\
LTBKSLASH_FILL	Relleno con \\\
HATCH_FILL	Sombreado claro
XHATCH_FILL	Sombreado espeso
INTERLEAVE_FILL	Lineas entrelazadas
WIDE_DOT_FILL	Puntos bastante espaciados
CLOSE_DOT_FILL	Puntos poco espaciados
USER_FILL	Relleno definido por el usuario

FILLSETTINGSTYPE (struct)

Usado para obtener los valores de relleno actuales por la función getfillsettings().

```
  struct fillsettingstype
    {
      int  pattern;
      int  color;
    };
```

---

font_names (enum)

Nombres de tipos de caracteres gráficos

```
  DEFAULT_FONT
  TRIPLEX_FONT
  SMALL_FONT
  SANS_SERIF_FONT
  GOTHIC_FONT
```

---

graphics_drivers (enum)

Controladores gráficos BGI.

```
  CGA           MCGA
  EGA           EGA64
  EGAMONO       IBM8514
  HERCMONO      ATT400
  VGA           PC3270
  DETECT (Requiere autodetección)


------------------------
  graphics_errors (enum)
========================

Códigos de error devuelto por graphresult().

  grOk               grNoInitGraph
  grNotDetected      grFileNotFound
  grInvalidDriver    grNoLoadMem
  grNoScanMem        grNoFloodMem
  grFontNotFound     grNoFontMem
  grInvalidMode      grError
  grIOerror          grInvalidFont
  grInvalidFontNum   grInvalidDeviceNum
  grInvalidVersion


------------------------
  graphics_modes (enum)
========================

Modos gráficos para cada controlador BGI
========================================
  CGAC0      |  320 x 200  |  paleta 0
  CGAC1      |  320 x 200  |  paleta 1
  CGAC2      |  320 x 200  |  paleta 2
  CGAC3      |  320 x 200  |  paleta 3
  CGAHI      |  640 x 200  |  2 colores
             |             |
  MCGAC0     |  320 x 200  |  paleta 0
  MCGAC1     |  320 x 200  |  paleta 1
  MCGAC2     |  320 x 200  |  paleta 2
  MCGAC3     |  320 x 200  |  paleta 3
  MCGAMED    |  640 x 200  |  2 colores
  MCGAHI     |  640 x 480  |  2 colores
             |             |
  EGALO      |  640 x 200  | 16 colores
  EGAHI      |  640 x 350  | 16 colores
  EGA64LO    |  640 x 200  | 16 colores
  EGA64HI    |  640 x 350  |  4 colores
             |             |
  EGAMONOHI  |  640 x 350  |  2 colores
  HERCMONOHI |  720 x 348  |  2 colores
             |             |
  ATT400C0   |  320 x 200  |  paleta 0
  ATT400C1   |  320 x 200  |  paleta 1
  ATT400C2   |  320 x 200  |  paleta 2
  ATT400C3   |  320 x 200  |  paleta 3
  ATT400MED  |  640 x 200  |  2 colores
  ATT400HI   |  640 x 400  |  2 colores
             |             |
  VGALO      |  640 x 200  | 16 colores
  VGAMED     |  640 x 350  | 16 colores
  VGAHI      |  640 x 480  | 16 colores
             |             |
```

```
  PC3270HI   |  720 x 350  |  2 colores
  IBM8514LO  |  640 x 480  |256 colores
  IBM8514HI  | 1024 x 768  |256 colores
```

```
--------------------
  line_styles (enum)
====================
```

Estilos de línea para las funciones getlinesettings() y setlinestyle().

```
  SOLID_LINE
  DOTTED_LINE
  CENTER_LINE
  DASHED_LINE
  USERBIT_LINE  estilo de línea definido por el usuario
```

```
--------------------
  line_widths (enum)
====================
```

Anchuras de línea para las funciones getlinesettings() y setlinestyle().

```
  NORM_WIDTH
  THICK_WIDTH
```

```
---------------------
  MAXCOLORS (#define)
=====================
```

Define el número máximo de entradas de colores para el campo array de colores en la estructura palettetype.

```
----------------------
  PALETTETYPE (struct)
======================
```

Contiene información de la paleta para el controlador gráfico actual. Esta estructura es usada por las funciones getpalette(), setpalette() y setallpalette().

```
  struct palettetype
    {
      unsigned char  size;
      signed   char  colors[MAXCOLORS+1];
    };
```

```
--------------------
  POINTTYPE (struct)
====================
```

Coordenadas de un punto.

```
  struct pointtype
    {
      int  x ;
      int  y ;
    };
```

---
putimage_ops (enum)
===

Operadores para putimage().

COPY_PUT	Copia
XOR_PUT	Exclusive OR
OR_PUT	Inclusive OR
AND_PUT	AND
NOT_PUT	Copia inversa de fuente

---
text_just (enum)
===

Justificación horizontal y vertical para la función settextjustify().

```
LEFT_TEXT
CENTER_TEXT
RIGHT_TEXT
BOTTOM_TEXT
TOP_TEXT
```

---
*_DIR (Dirección) (#defines)
===

Dirección de salida gráfica.

HORIZ_DIR	De izquierda a derecha
VERT_DIR	De abajo hacia arriba

TEXTSETTINGSTYPE (struct)
===

Usado para obtener los valores de texto actual por la función gettextsettings().

```
struct textsettingstype
  {
    int  font;
    int  direction;
    int  charsize;
    int  horiz;
    int  vert;
  };
```

USER_CHAR_SIZE (#define)
===

Tamaño de los caracteres definidos por el usuario (tamaño de caracteres = amplificación de caracteres en salida gráfica).

Valores que puede tomar esta constante simbólica:

- 1 visualiza caracteres en caja de 8-por-8 en la pantalla
- 2 visualiza caracteres en caja de 16-por-16 en la pantalla

```
   ...
þ 10  visualiza caracteres en caja de 80-por-80 en la pantalla
```

```
----------------------
  VIEWPORTTYPE (struct)
======================
```

Estructura usada para obtener información sobre el viewport corriente por la función getviewsettings().

```
  struct viewporttype
    {
      int  left;
      int  top;
      int  right;
      int  bottom;
      int  clip;
    };
```

NOTA: En el ejemplo 4 de esta lección se muestra cómo se pueden utilizar la mayoría de estas funciones.

LECCIÓN

INDICE DE LA LECCION 11

- Primera parte:

  * Saltos no locales (**setjmp.h**).
  * Envío y recepción de señales (**signal.h**).
  * Asignación dinámica (**alloc.h** en Turbo C).
  * Funciones de proceso (**process.h**).

- Segunda parte:

  * Funciones de directorio (**dir.h**).
  * Funciones del DOS (**interrupt** y **dos.h**).
  * Funciones de la ROM BIOS (**bios.h**).

SALTOS NO LOCALES

En el fichero de cabecera **<setjmp.h>** tenemos la información necesaria para poder realizar saltos no locales, es decir, saltos de una función a otra.

Recordad que la instrucción goto del
C sólo puede realizar un salto dentro
de una función.

FICHERO DE CABECERA SETJMP.H

En esta biblioteca sólo hay dos funciones: longjmp() y setjmp().

---
longjmp Ejecuta goto (salto) no local.
=========

Sintaxis:
```
void longjmp (jmp_buf jmpb, int valdev);
```

La instrucción longjmp() da lugar a que la ejecución del programa se retome en el punto en que se llamó por última vez a setjmp(). Estas dos funciones son la forma en que C permite saltar entre funciones. Tenga en cuenta que se necesita la cabecera setjmp.h.

La función longjmp() opera inicializando la pila ya descrita por jmpb, que debe haber sido activada en una llamada anterior a setjmp(). Esto da lugar a que la ejecución del programa se retome en la sentencia siguiente al setjmp() que la llamó. En otras palabras, la computadora es engañada haciéndole pensar que nunca dejó la función que llamó a setjmp().

El buffer jmpb es de tipo jmp_buf, que está definido en la cabecera setjmp.h. El buffer debe haber sido activado a través de una llamada a setjmp(), anterior a la llamada a longjmp().

El valor de valdev se transforma en el valor de vuelta de setjmp() y puede preguntarse por él para determinar de dónde viene longjmp(). El único valor no permitido es 0, puesto que setjmp() devuelve este valor cuando se le llama por primera vez.

Es importante comprender que longjmp() se debe llamar antes de que la función que llamó a setjmp() vuelva. Si no, el resultado queda indefinido. (Realmente, la mayor parte de las veces el programa aborta.)

---
setjmp Almacena información de contexto para poder realizar saltos no
======== locales.

Sintaxis:
```
int setjmp (jmp_buf jmpb);
```

La función setjmp() guarda el contenido de la pila del sistema en el buffer jmpb para ser utilizado más tarde por longjmp().

La función setjmp() devuelve 0 después de la llamada. Sin embargo, longjmp() pasa un argumento a setjmp() cuando se ejecuta, y es este valor (que siempre es disntino de cero) el valor de setjmp() después de la llamada a longjmp().

A continuación se presenta la estructura del buffer para estas dos funciones en Turbo C, si utilizas otro compilador puedes ver cómo es en el tuyo visua-

lizando el fichero setjmp.h. De todas formas no es de mucha utilidad saber cuál es el contenido de esta estructura.

---
jmp_buf (tipo)
==========

Un buffer de tipo jmp_buf es usado para salvar y restaurar el estado de un programa en un determinado momento.

```
typedef struct
  {
    unsigned  j_sp,   j_ss,
    unsigned  j_flag, j_cs;
    unsigned  j_ip,   j_bp;
    unsigned  j_di,   j_es;
    unsigned  j_si,   j_ds;
  } jmp_buf[1];
```

Veamos un ejemplo bastante ilustrativo de cómo poder realizar un salto no local. Este ejemplo corresponde al primer ejemplo de esta lección, de este modo lo puedes ejecutar.

```
/*
  Este programa imprime  1 2 3
*/

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

void func (jmp_buf);

void main (void)
{
  int valor;
  jmp_buf jmpb;

  printf ("1 ");
  valor = setjmp (jmpb);
  if (valor != 0)
    {
      printf ("3 ");
      exit (valor);
    }
  printf ("2 ");
  func (jmpb);
  printf ("4 ");
}

void func (jmp_buf jmpb)
{
  longjmp (jmpb, 1);
}
```

ENVIO Y RECEPCION DE SE¥ALES

En el fichero de cabecera **<signal.h>**

hay declaradas dos funciones junto
con algunas macros para poder enviar
y recibir señales (interrupciones)
en un programa en ejecución.

FICHERO DE CABECERA SIGNAL.H

Este fichero sólo declara dos funciones: raise() y signal(). Al ser estas dos funciones dependientes del sistema, el ANSI C no da una sintaxis general para estas funciones, sino que nos dice que la función raise() envía una señal a un programa en ejecución y la función signal() define la función que se ejecutará en un programa cuando éste reciba una determinada señal; el ANSI C también define las macros: SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, SIG_DFL y SEIG_IGN, todas ellas declaradas, junto con las funciones raise() y signal(), en el fichero de cabecera signal.h. Una descripción completa de todas estas macros y funciones se realizará en las siguientes líneas para aquellos usuarios que tengan la opción de turbo a on.

---
raise Envía una señal software.
=======

Sintaxis:
```
int raise (int sig);
```

Un programa se puede enviar señales a sí mismo usando raise(). Esta función ejecuta el manejador instalado por signal() para ese tipo de señal (o el manejador por defecto).

Devuelve 0 si tiene éxito y un valor distinto de 0 en caso contrario.

---
signal Especifica las acciones a realizar asociadas a una señal.
========

Sintaxis:
```
void (* signal (int sig, void (*func) (int sig [,int subcode]))) (int);
```

La función a la cual apunta func será llamada cuando una señal de tipo sig sea enviada.

Las señales son enviadas cuando ocurre una condición de excepción o cuando es llamada la función raise().

El prototipo de esta función en el fichero signal.h es:

```
void (* signal (int sig, void (*func) (/* int */))) (int);
```

lo que quiere decir que los parámetros de func están indefinidos pero que normalmente suele ser un int.

SIG_xxx (#defines)

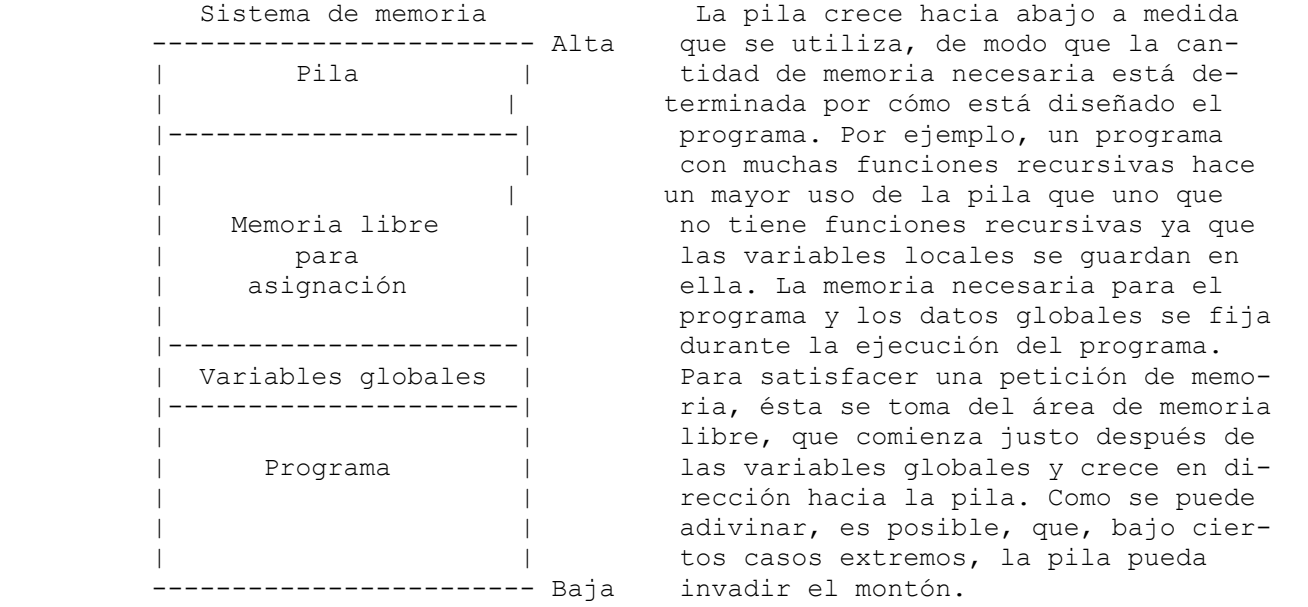
Las funciones predefinidas para manejar las señales generados por la función raise() o por sucesos externos son:

```
  SIG_DFL   Terminar el programa.
  SIG_IGN   Ninguna acción, ignorar señal.
  SIG_ERR   Devuelve código de error.


--------------------
  SIGxxxx (#defines)
====================

Los tipos de señales usados por las funciones raise() y signal() son:

  SIGABRT   Abortar
  SIGFPE    Trampa de punto flotante
  SIGILL    Instrucción ilegal
  SIGINT    Interrupción
  SIGSEGV   Violación de acceso a memoria
  SIGTERM   Terminar


Veamos a continuación tres ejemplos sobre las funciones y macros del
fichero de cabecera signal.h.

Ejemplo 1:

  /* Este programa imprime  Divide error  */

  #include <stdio.h>
  #include <signal.h>

  void main (void)
  {
    int x, y;

    x = 1; y = 0;
    printf ("x/y: %d", x / y);
  }

Ejemplo 2:

  /* Este programa imprime  Error de punto flotante.  */

  #include <stdio.h>
  #include <signal.h>

  void manejador (void)
  {
    printf ("Error de punto flotante.\n");
  }

  void main (void)
  {
    int x, y;

    signal (SIGFPE, manejador);

    x = 1; y = 0;
    printf ("x/y: %d", x / y);
  }

Ejemplo 3:

  /* Este programa imprime  Error de punto flotante.  */
```

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

void manejador (void)
{
  printf ("Error de punto flotante.\n");
}

void main (void)
{
  signal (SIGFPE, manejador);

  raise (SIGFPE);
}
```

ASIGNACION DINAMICA

Hay dos modos fundamentales por los que un programa en C puede guardar información en la memoria central de la computadora. El primero utilizando variables globales y locales (incluyendo arrays y estructuras). El segundo modo por el que se puede guardar información es mediante el sistema de asignación dinámica de C. Por este método se asigna de la memoria libre tanto espacio como se necesite. La zona de memoria libre se encuentra entre el área de almacenamiento permanente del programa y la pila. (Para la familia de procesadores 8088, el montón se supone que está en el segmento de datos por defecto.)



El núcleo del sistema de asignacion dinámica son las funciones malloc() y free() que se encuentran en la biblioteca estándar de C. Cada vez que se hace una petición de memoria mediante malloc(), se asigna una porción del resto de la memoria libre. Cada vez que se hace una llamada para liberar memoria mediante free(), la memoria se devuelve al sistema. La forma más común para implementar malloc() y free() es organizar la memoria libre en una lista enlazada. Sin embargo, el estándar ANSI propuesto afirma que el método de gestión de memoria depende de la implementación.

El estándar ANSI propuesto especifica que la información de cabecera necesaria para el sistema de asignación dinámica se encuentra en stdlib.h

y sólo define cuatro funciones para el sistema de asignación dinámica: calloc(), malloc(), free() y realloc(). Sin embargo, los compiladores suelen suministrar más funciones bien en el fichero de cabecera malloc.h o bien en el fichero alloc.h, dependiendo del compilador.

Las funciones añadidas al ANSI C son necesarias para soportar eficientemente la arquitectura segmentada de la familia de procesadores 8088. Estas funciones no son válidas para compiladores diseñados para otros procesadores como el 68000. Debido a la segmentación de memoria que presenta la familia de procesadores 8088, se incorporan dos nuevos modificadores de tipo no estándares que generalmente son soportados por compiladores construidos para estos procesadores. Son near y far, que se utilizan normalmente para crear punteros de otro tipo que el normalmente utilizado por el modelo de memoria del compilador. La arquitectura de la familia de procesadores 8088 permite el acceso directo a 1 megabyte de RAM (o más con el 80386 o 80486), pero necesita utilizar un segmento y una dirección de desplazamiento.

Esencialmente, es posible organizar la memoria de una computadora basada en el 8088 en uno de estos cuatro modelos (se presentan en orden creciente de tiempo de ejecución):

Pequeño: Todo el código debe ajustarse a un segmento de 64K y todos los datos deben ajustarse a un segundo segmento de 64K. Todos los punteros son de 16 bits. Este modelo da lugar a la ejecución más rápida del programa.

Medio: Todos los datos deben contenerse en un segmento de 64K, pero el código puede utilizar varios segmentos. Todos los punteros a datos son de 16 bits. Este modelo permite un rápido acceso a los datos, pero una ejecución del código lenta.

Compacto: Todo el código debe ajustarse a un segmento de 64K, pero los datos pueden utilizar varios segmentos. Todos los punteros a datos son de 32 bits. La organización compacta de la memoria supone un acceso lento a los datos, pero una ejecución de código rápida.

Grande: Código y datos utilizan varios segmentos. Todos los punteros son de 32 bits. Este modelo da lugar a la ejecución más lenta del programa.

> Como puedes imaginar, el acceso a memoria utilizando punteros de 16 bits es mucho más rápido que utilizando punteros de 32 bits. Sin embargo, los punteros de 32 bits son necesarios cuando el código, los datos, o ambos superan la barrera de los 64K. Una forma de evitar el uso de los punteros de 32 bits es permitir la creación de punteros de 16 ó 32 bits explícitamente por parte del programa, pasando por alto así el valor implícito. Típicamente, esto ocurre cuando un programa necesita gran cantidad de datos para una determinada operación. En este caso se crea un puntero far, y la memoria se asigna utilizando la versión no estándar de malloc() que asigna memoria fuera del segmento de datos por defecto. De esta forma todos los demás accesos a memoria son rápidos, y el tiempo de ejecución no se incrementa tanto como si se hubiera utilizado un modelo grande. A la inversa también puede ocurrir. Un programa que utilice un modelo grande puede fijar un puntero near a un fragmento de memoria frecuentemente accedido para incrementar el rendimiento.

En la siguiente ventana estudiaremos todas las funciones que están declaradas en el fichero de cabecera **<alloc.h>** que se encuentra en Turbo C.

Las funciones de asignación dinámica que define el ANSI C y que en el estándar están declaradas en el fichero de cabecera **<stdlib.h>**, en el caso de Turbo C, están declaradas también en el fichero alloc.h. Estas funciones son: **calloc()**, **malloc()**, **free()** y **realloc()**.

FICHERO DE CABECERA ALLOC.H (TC)

---
brk Cambia la asignación de espacio del segmento de datos.
=====

Sintaxis:
```
int brk (void *addr);
```

Pone el tope del segmento de datos del programa en la localización de memoria apuntada por addr. Si la operación tiene éxito, brk() devuelve el valor de 0. Si ocurre un fallo, devuelve el valor de -1 y da valor a errno.

Ejemplo:

```
/*
  Este programa imprime (en mi sistema):

  Cambiando asignación con brk().
  Antes de la llamada a brk(): 63488 bytes libres.
  Después de la llamada a brk(): 62496 bytes libres.

*/

#include <stdio.h>
#include <alloc.h>

void main (void)
{
  char *ptr;

  printf ("Cambiando asignación con brk().\n");
  ptr = malloc (1);
  printf ("Antes de la llamada a brk(): %lu bytes libres.\n",
          (long unsigned) coreleft());
  brk (ptr+1000);
  printf ("Después de la llamada a brk(): %lu bytes libres.\n",
          (long unsigned) coreleft());
}
```

---
calloc Asigna memoria principal.
========

Sintaxis:
```
void *calloc (size_t nelems, size_t tam);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h.

Asigna espacio para nelems elementos de tam bytes cada uno y almacena cero en el área.

Devuelve un puntero al nuevo bloque asignado o NULL si no existe bastante espacio.

---
coreleft Devuelve la cantidad de memoria no usada.

Sintaxis:

Modelos tiny, small, y medium:

```
unsigned coreleft (void);
```

Modelos compact, large, y huge:

```
unsigned long coreleft (void);
```

El lector de este texto quizás se pregunte cómo puede declararse la misma función coreleft() con dos tipos diferentes. Esto es muy fácil hacerlo utilizando las directivas de compilación condicional:

```
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
  unsigned long coreleft (void);
#else
  unsigned coreleft (void);
#endif
```

Ejemplo:

```
/*
  Este programa imprime (en mi sistema):

  La diferencia entre el bloque asignado más alto
  y la cima del montón es: 63552 bytes.

*/

#include <stdio.h>
#include <alloc.h>

void main (void)
{
   printf ("La diferencia entre el bloque asignado más alto\n"
           "y la cima del montón es: %lu bytes.\n",
           (unsigned long) coreleft());
}
```

---
farcalloc Asigna memoria del montón far.

Sintaxis:

```
void far *farcalloc (unsigned long nunids, unsigned long tamunid);
```

Asigna espacio para nunids elementos de tamunid cada uno. Devuelve un puntero al nuevo bloque asignado, o NULL si no hay suficiente memoria para el nuevo bloque.

---
farcoreleft Devuelve la cantidad de memoria no usada en el montón far.

Sintaxis:

```
   unsigned long farcoreleft (void);

Devuelve la cantidad total de espacio libre (en bytes) entre el bloque
asignado más alto y el final de la memoria.


---------
  farfree    Libera un bloque del montón far.
=========

Sintaxis:
   void farfree (void far *bloque);


--------------
  farheapcheck    Chequea y verifica el montón far.
==============

Sintaxis:
   int farheapcheck (void);

La función farheapcheck() camina a través del montón far y examina cada
bloque, chequeando sus punteros, tamaño, y otros atributos críticos.

El valor devuelto es menor de cero si ocurre un error y mayor de cero si
tiene éxito.


------------------
  farheapcheckfree    Chequea los bloques libres en el montón far para un
==================   valor constante.

Sintaxis:
   int farheapcheckfree (unsigned int valorrelleno);

El valor devuelto es menor de cero si ocurre un error y mayor de cero si
tiene éxito.


------------------
  farheapchecknode    Chequea y verifica un nodo simple en el montón far.
==================

Sintaxis:
   int farheapchecknode (void *nodo);

Si un nodo ha sido liberado y farheapchecknode() es llamado con un puntero
al bloque libre, farheapchecknode() puede devolver _BADNODE en vez del
esperado _FREEENTRY. Esto sucede porque los bloques libres adyacentes en
el montón son unidos, y el bloque en cuestión no existe.


-----------------
  farheapfillfree    Rellena el bloque libre en el montón far con un valor
=================   constante.

Sintaxis:
   int farheapfillfree (unsigned int valorrelleno);

El valor devuelto es menor de cero si ocurre un error y es mayor de cero
si tiene éxito.


-------------
```

```
  farheapwalk    Camina a través del montón far nodo a nodo.
=============

Sintaxis:
   int farheapwalk (struct farheapinfo *hi);

La función farheapwalk() asume que el montón es correcto. Usa farheapcheck()
para verificar el montón antes de usar farheapwalk(). _HEAPOK es devuelto
con el último bloque en el montón. _HEAPEND será devuelto en la próxima
llamada a farheapwalk().

La estructura farheapinfo está definida del siguiente modo:

struct heapinfo
   {
     void huge *ptr;
     unsigned long size;
     int in_use;
   };


-----------
  farmalloc    Asigma memoria del montón far.
===========

Sintaxis:
   void far *farmalloc (unsigned long nbytes);

Devuelve un puntero al nuevo bloque asignado, o NULL si no existe suficiente
espacio para el nuevo bloque.


------------
  farrealloc    Ajusta bloque asignado en montón far.
============

Sintaxis:
   void far *farrealloc (void far *viejobloque, unsigned long nbytes);

Devuelve la dirección del bloque reasignado, o NULL si falla. La nueva
dirección puede ser diferente a la dirección original.


------
  free    Libera bloques asignados con malloc() o calloc().
======

Sintaxis:
   void free (void *bloque);


-----------
  heapcheck    Chequea y verifica el montón.
===========

Sintaxis:
   int heapcheck (void);

La función farheapcheck() camina a través del montón y examina cada
bloque, chequeando sus punteros, tamaño, y otros atributos críticos.

El valor devuelto es menor de cero si ocurre un error y mayor de cero si
tiene éxito.
```

Ejemplo:

```
  /* Este programa imprime:  El montón está corrompido.  */

  #include <stdio.h>
  #include <alloc.h>

  void main (void)
  {
    char *p;

    p = malloc (100);
    free (p+1);

    if (heapcheck () < 0)
      printf ("El montón está corrompido.\n");
    else
      printf ("El montón es correcto.\n");
  }
```

---

heapcheckfree — Chequea los bloques libres en el montón para un valor constante.

Sintaxis:

```
   int heapcheckfree (unsigned int valorrelleno);
```

El valor devuelto es menor de cero si ocurre un error y mayor de cero si tiene éxito.

---

heapchecknode — Chequea y verifica un nodo simple en el montón.

Sintaxis:

```
   int heapchecknode (void *nodo);
```

Si un nodo ha sido liberado y heapchecknode() es llamado con un puntero al bloque libre, heapchecknode() puede devolver _BADNODE en vez del esperado _FREEENTRY. Esto sucede porque los bloques libres adyacentes en el montón son unidos, y el bloque en cuestión no existe.

El valor devuelto es menor de cero si ocurre un error y mayor de cero si tiene éxito.

Ejemplo:

```
  /*
    Este programa imprime (en mi sistema):

    Nodo  0: Entrada libre.
    Nodo  1: Entrada usada.
    Nodo  2: Entrada libre.
    Nodo  3: Entrada usada.
    Nodo  4: Entrada libre.
    Nodo  5: Entrada usada.
    Nodo  6: Entrada libre.
    Nodo  7: Entrada usada.
    Nodo  8: Entrada libre.
    Nodo  8: Entrada usada.

    Las macros _HEAPEMPTY, _HEAPCORRUPT, _BADNODE, _FREEENTRY y _USEDENTRY
```

```
    están declaradas en el fichero alloc.h
  */

  #include <stdio.h>
  #include <alloc.h>

  #define NUM_PTRS  10
  #define NUM_BYTES 16

  void main (void)
  {
    char * array [NUM_PTRS];
    int i;

    for (i = 0; i < NUM_PTRS; i++)
      array [i] = malloc (NUM_BYTES);

    for (i = 0; i < NUM_PTRS; i += 2)
      free (array [i]);

    for (i = 0; i < NUM_PTRS; i++)
      {
        printf ("Nodo %2d ", i);
        switch (heapchecknode (array [i]))
          {
            case _HEAPEMPTY:
              printf ("Ningún montón.\n");
              break;
            case _HEAPCORRUPT:
              printf ("Montón corrupto.\n");
              break;
            case _BADNODE:
              printf ("Nodo malo.\n");
              break;
            case _FREEENTRY:
              printf ("Entrada libre.\n");
              break;
            case _USEDENTRY:
              printf ("Entrada usada.\n");
              break;
            default:
              printf ("Código de vuelta desconocido.\n");
              break;
          }
      }
  }


--------------
  heapfillfree    Rellena el bloque libre en el montón con un valor
==============   constante.

Sintaxis:
   int heapfillfree (unsigned int valorrelleno);

El valor devuelto es menor de cero si ocurre un error y es mayor de cero
si tiene éxito.


----------
  heapwalk    Camina a través del montón nodo a nodo.
==========

Sintaxis:
```

```
   int heapwalk (struct heapinfo *hi);
```

La función heapwalk() asume que el montón es correcto. Usa heapcheck() para verificar el montón antes de usar heapwalk(). _HEAPOK es devuelto con el último bloque en el montón. _HEAPEND será devuelto en la próxima llamada a heapwalk().

La función heapwalk() recibe un puntero a una estructura de tipo heapinfo (declarada en alloc.h). Para la primera llamada a heapwalk(), pon el campo hi.ptr a null. La función heapwalk() devuelve en hi.ptr la dirección del primer bloque. El campo hi.size contiene el tamaño del bloque en bytes. El campo h.in_use es un flag que se pone a 1 si el bloque está actualmente en uso.

La estructura heapinfo está definida del siguiente modo:

```
struct heapinfo
   {
     void *ptr;
     unsigned int size;
     int in_use;
   };
```

---

malloc Asigna memoria principal.

Sintaxis:

```
   void *malloc (size_t tam);
```

El prototipo de esta función también se encuentra en el fichero de cabecera stdlib.h.

El parámetro tam está en bytes. Devuelve un puntero al nuevo bloque asignado, o NULL si no existe suficiente espacio para el nuevo bloque. Si tam == 0, devuelve NULL.

---

realloc Reasigna memoria principal.

Sintaxis:

```
   void *realloc (void *bloque, size_t tam);
```

El prototipo de esta función también se encuentra en el fichero de cabecera stdlib.h.

Intenta achicar o expandir el bloque asignado previamente a tam bytes. Devuelve la dirección del bloque reasignado, la cual puede ser diferente de la dirección original.

Si el bloque no puede ser reasignado o tam == 0, reallo() devuelve NULL.

Ejemplo:

```
  /*
    Este programa imprime (en mi sistema):

    El string es Hola
      Está en la dirección 05A0
    El string es Hola
      Está en la nueva dirección 05AE
```

```
  */

  #include <stdio.h>
  #include <alloc.h>
  #include <string.h>

  void main (void)
  {
    char *str;

     /* asigna memoria para string */
    str = malloc (10);

    /* copia "Hola" en string */
    strcpy (str, "Hola");

    printf ("El string es %s\n  Está en la dirección %p\n", str, str);
    str = realloc (str, 20);
    printf("El string is %s\n  Está en la nueva dirección %p\n", str, str);

    /* libera memoria */
    free (str);
  }


------
  sbrk    Cambia la asignación de espacio del segmento de datos.
======

Sintaxis:
   void *sbrk (int incr);

Suma incr bytes al valor umbral. Si la operación tiene éxito, sbrk()
devuelve el viejo valor umbral. Si falla, devuelve -1 y le da valor a errno.

Ejemplo:

  /*
    Este programa imprime (en mi sistema):

    Cambiando asignación con sbrk().
    Antes de la llamada a sbrk(): 63504 bytes libres.
    Después de la llamada a sbrk(): 62496 bytes libres.

  */

  #include <stdio.h>
  #include <alloc.h>

  void main (void)
  {
    printf ("Cambiando asignación con sbrk().\n");
    printf ("Antes de la llamada a sbrk(): %lu bytes libres.\n",
            (unsigned long) coreleft());
    sbrk (1000);
    printf ("Después de la llamada a sbrk(): %lu bytes libres.\n",
            (unsigned long) coreleft());
  }

partir
```

FUNCIONES DE PROCESO

En el fichero **<process.h>** de Turbo C nos encontramos la declaración de una serie de funciones que las podemos dividir en dos grupos: las que terminan el programa en ejecución para volver al sistema operativo y las que ejecutan otro programa.

FICHERO DE CABECERA PROCESS.H (TC)

GLOSARIO:
****

```
-------
  abort    Termina un proceso anormalmente.
=======
-------------------
  funciones exec...    Las funciones exec... permiten a nuestro programa
==================   ejecutar otros programs (procesos hijos).
------
  exit    Termina el programa.
======
-------
  _exit    Termina programa.
=======
--------------------   Las funciones spawn... permiten a nuestros programas
  spawn... functions    ejecutar procesos hijos (otros programas) y devolver
====================   el control a nuestro programa cuando el proceso hijo
                        finaliza.
--------
  system    Ejecuta un comando DOS.
========
```

FUNCIONES:
****

```
-------
  abort    Termina un proceso anormalmente.
=======
```

Sintaxis:

```
   void abort (void);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h

```
-------------------
  funciones exec...    Las funciones exec... permiten a nuestro programa
==================   ejecutar otros programs (procesos hijos).
```

Sintaxis:

```
   int execl (char *path, char *arg0, .., NULL);
   int execle (char *path, char *arg0, .., NULL, char **env);
   int execlp (char *path, char *arg0, ..);
   int execlpe (char *path, char *arg0, .., NULL, char **env);
```

```
   int execv (char *path, char *argv[]);
   int execve (char *path, char *argv[], char **env);
   int execvp (char *path, char *argv[]);
   int execvpe (char *path, char *argv[], char **env);
```

Cuando se hace una llamada exec..., el proceso hijo ocupa el lugar del proceso padre. Debe haber suficiente memoria disponible para cargar y ejecutar el proceso hijo.

Usa execl(), execle(), execlp() y execlpe() cuando conoces todos los argumentos que tendrá el proceso hijo a ejecutar.

Usa execv(), execve(), execvp() y execvpe() cuando no conoces a priori los argumentos que tendrá el proceso hijo a ejecutar.

Las letras al final de cada función exec... identifica qué variación se va a usar:

Ltr	Variación usada
p	Busca el path del DOS para el proceso hijo
l	exec pasó una lista fija de argumentos
v	exec pasó una lista variable de argumentos
e	exec pasó un puntero al entorno, permitiendo alterar el entorno que tendrá el proceso hijo

Si tiene éxito, las funciones exec no devuelven nada. En caso de error, las funciones exec devuelven -1 y asigna a errno el código de error.

exit — Termina el programa.

Sintaxis:

```
   void exit (int estado);
```

Antes de terminar, la salida buffereada es volcada, los ficheros son cerrados y las funciones exit() son llamadas.

El prototipo de esta función también se encuentra en el fichero stdlib.h.

_exit — Termina programa.

Sintaxis:

```
   void _exit (int estado);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h.

spawn... functions — Las funciones spawn... permiten a nuestros programas ejecutar procesos hijos (otros programas) y devolver el control a nuestro programa cuando el proceso hijo finaliza.

Sintaxis:

```
   int spawnl (int mode, char *path, char *arg0, ..., NULL);
   int spawnle (int mode, char *path, char *arg0, ..., NULL, char *envp[]);
   int spawnlp (int mode, char *path, char *arg0, ..., NULL);
   int spawnlpe (int mode, char *path, char *arg0, ..., NULL, char *envp[]);

   int spawnv (int mode, char *path, char *argv[]);
```

```
   int spawnve (int mode, char *path, char *argv[], char *envp[]);
   int spawnvp (int mode, char *path, char *argv[]);
   int spawnvpe (int mode, char *path, char *argv[], char *envp[]);
```

Usa spawnl(), spawnle(), spawnlp() y spawnlpe() cuando conoces todos los argumentos que tendrá el proceso hijo a ejecutar.

Usa spawnv(), spawnve(), spawnvp() y spawnvpe() cuando no conoces a priori los argumentos que tendrá el proceso hijo a ejecutar.

Las letras al final de cada función spawn... identifica qué variación se va a usar:

Ltr	Variación usada
p	Busca el path del DOS para el proceso hijo
l	spawn pasó una lista fija de argumentos
v	spawn pasó una lista variable de argumentos
e	spawn pasó un puntero al entorno, permitiendo alterar el entorno que tendrá el proceso hijo

Si la ejecución tiene éxito, el valor devuelto es el estado de salida del proceso hijo (0 para terminación normal).

Si el proceso no puede ser ejecutado, las funciones spawn... devolverán -1.

system — Ejecuta un comando DOS.

Sintaxis:

```
   int system (const char *comando);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h.

comando puede ejecutar un comando interno del DOS tales como DIR, un fichero de programa .COM o .EXE, o un fichero batch .BAT.

Devuelve 0 en caso de éxito, -1 en caso de error y se le asigna a errno uno de los siguientes valores: ENOENT, ENOMEM, E2BIG o ENOEXEC.

La función system() también se encuentra declarada en los ficheros stdlib.h y system.h. En el fichero system.h sólo se encuentra el prototipo de la función system().

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:

P_xxxx (#defines)

Modos usados por las funciones spawn.

P_WAIT	El proceso hijo se ejecuta separadamente, el proceso padre espera a la salida del hijo.
P_NOWAIT	Los procesos hijo y padre se ejecutan concurrentemente. (No implementado.)
P_OVERLAY	El proceso hijo reemplaza al proceso padre de tal forma que el padre ya no existe.

```
------------------------
  _psp (variable global)
========================
```

Dirección del segmento del PSP (Prefijo de Segmento de Programa) del programa.

```
  extern unsigned int _psp;
```

Esta variable también se encuentra declarada en los ficheros dos.h y stdlib.h.

FUNCIONES DE DIRECTORIO

En el fichero de cabecera **<dir.h>** de Turbo C tenemos declaradas una serie de funciones relacionadas con los directorios.

FICHERO DE CABECERA DIR.H (TC)

GLOSARIO:
****

```
-------
  chdir    Cambia directorio actual.
=======
------------------------
  findfirst and findnext    Busca directorio de disco.
========================   Continúa búsqueda de findfirst().
---------
  fnmerge    Construye un path de sus partes componentes.
=========
---------
  fnsplit    Descompone un nombre de path en sus partes componentes.
=========
-----------
  getcurdir    Obtiene directorio actual para la unidad especificada.
===========
--------
  getcwd    Obtiene directorio de trabajo actual.
========
---------
  getdisk    Obtiene unidad actual.
=========
-------
  mkdir    Crea un directorio.
=======
--------
  mktemp    Hace un nombre de fichero único.
========
-------
  rmdir    Quita un directorio.
=======
------------
  searchpath    Busca el path del DOS para un determinado fichero.
```

```
============
---------
  setdisk    Pone la unidad de disco actual.
=========

-------
  chdir    Cambia directorio actual.
=======

Sintaxis:
   int chdir (const char *path);

Si la operación tiene éxito, chdir() devuelve 0. En otro caso, devuelve -1
y asigna a errno código de error.


FUNCIONES:
**********

------------------------
  findfirst and findnext    Busca directorio de disco.
========================   Continúa búsqueda de findfirst().

Sintaxis:
  int findfirst (const char *nombrepath, struct ffblk *ffblk, int atributo);
  int findnext (struct ffblk *ffblk);

El path de especificación de fichero puede contener estos caracteres
comodín:

  ? (coincidencia de un carácter)
  * (coincidencia de una serie de caracteres)

Devuelve 0 si tiene éxito; devuelve -1 si ocurre un error, y se le asigna
a errno el código de error.

Ejemplo:

  #include <stdio.h>
  #include <dir.h>

  int main (void)
  {
    struct ffblk ffblk;
    int hecho;

    printf ("Listado de directorio *.*\n");
    hecho = findfirst ("*.*", &ffblk, 0);
    while (! hecho)
      {
        printf ("  %s\n", ffblk.ff_name);
        hecho = findnext (&ffblk);
      }

    return 0;
  }


---------
  fnmerge    Construye un path de sus partes componentes.
=========

Sintaxis:
   void fnmerge (char *path, const char *unidad, const char *dir,
```

```
                const char *nombre, const char *ext);
```

Ejemplo:

```
  #include <string.h>
  #include <stdio.h>
  #include <dir.h>

  int main (void)
  {
    char s[MAXPATH];
    char unidad[MAXDRIVE];
    char dir[MAXDIR];
    char fichero[MAXFILE];
    char ext[MAXEXT];

    getcwd (s, MAXPATH); /* obtiene el directorio de trabajo actual */
    if (s[strlen(s)-1] != '\\')
      strcat (s,"\\"); /* añade un carácter \ al final */
    fnsplit (s, unidad, dir, fichero, ext); /* descompone el string en
                                               elementos separados */
    strcpy (fichero, "DATOS");
    strcpy (ext," .TXT");
    fnmerge (s, unidad, dir, fichero, ext); /*fusiona todas las componentes*/
    puts (s); /* visualiza el string resultado */

    return 0;
  }
```

---

fnsplit Descompone un nombre de path en sus partes componentes.

Sintaxis:

```
   int fnsplit (const char *path, char *unidad, char *dir, char *nombre,
                char *ext);
```

Devuelve un entero compuesto de cinco flags.

Ejemplo:

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

  int main (void)
  {
    char *s;
    char unidad[MAXDRIVE];
    char dir[MAXDIR];
    char fichero[MAXFILE];
    char ext[MAXEXT];
    int flags;

    s = getenv ("COMSPEC"); /* obtiene la especificación de nombre completa
                               de command.com */
    flags = fnsplit (s, unidad, dir, fichero, ext);

    printf ("Información del procesador de comando:\n");
    if (flags & DRIVE)
      printf ("\tunidad: %s\n", unidad);
    if (flags & DIRECTORY)
      printf ("\tdirectorio: %s\n",dir);
```

```
    if (flags & FILENAME)
      printf ("\tfichero: %s\n", fichero);
    if (flags & EXTENSION)
      printf ("\textensión: %s\n", ext);

    return 0;
  }
```

---

getcurdir Obtiene directorio actual para la unidad especificada.

Sintaxis:

```
   int getcurdir (int unidad, char *directorio);
```

unidad es 0 para la unidad por defecto, 1 para A, 2 para B, etc.

Devuelve 0 si tiene éxito y -1 si hay algún error.

---

getcwd Obtiene directorio de trabajo actual.

Sintaxis:

```
   char *getcwd (char *buffer, int longitud_buffer);
```

Devuelve un puntero a buffer; en caso de error, devuelve NULL y a errno se le asigna el código de error.

Ejemplo:

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

  int main (void)
  {
     char buffer [MAXPATH];

     getcwd (buffer, MAXPATH);
     printf ("El directorio actual es: %s\n", buffer);
     return 0;
  }
```

---

getdisk Obtiene unidad actual.

Sintaxis:

```
   int getdisk (void);
```

Devuelve la unidad actual. La unidad A es la 0.

Ejemplo:

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

  int main (void)
  {
     int disco;
```

```
     disk = getdisk () + 'A';
     printf ("La unidad actual es: %c\n", disco);
     return 0;
  }
```

---
  mkdir   Crea un directorio.
=======

Sintaxis:
```
   int mkdir (const char *path);
```

Devuelve 0 si tiene éxito; -1 en caso de error y se asigna código de error a errno.

Ejemplo:

```
  #include <stdio.h>
  #include <conio.h>
  #include <process.h>
  #include <dir.h>

  int main (void)
  {
    int estado;

    clrscr ();
    estado = mkdir ("asdfjklm");
    (! estado) ? (printf ("Directorio creado\n")) :
                 (printf ("No es posible crear directorio\n"));

     getch ();
     system ("dir");
     getch ();

     estado = rmdir ("asdfjklm");
     (! estado) ? (printf ("Directorio borrado\n")) :
                  (perror ("No es posible borrar directorio\n"));

     return 0;
  }
```

---
  mktemp   Hace un nombre de fichero único.
========

Sintaxis:
```
   char *mktemp (char *nomfich);
```

Reemplaza nomfich por un nombre de fichero único y devuelve la dirección de nomfich; nomfich debería ser una cadena terminada en nulo con 6 X's restantes. Por ejemplo, "MIFICHXXXXXX".

Ejemplo:

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

  int main (void)
  {
     char *nombref = "TXXXXXX", *ptr;
```

```
     ptr = mktemp (nombref);
     printf ("%s\n", ptr);
     return 0;
  }


-------
  rmdir    Quita un directorio.
=======

Sintaxis:
   int rmdir (const char *path);

Devuelve 0 si tiene éxio; en caso de error, devuelve -1 y en errno se pone
el código de error.

Ejemplo:

  #include <stdio.h>
  #include <conio.h>
  #include <process.h>
  #include <dir.h>

  #define NOMBREDIR "testdir.$$$"

  int main (void)
  {
    int estado;

    estado = mkdir (NOMBREDIR);
    if (! estado)
      printf ("Directorio creado\n");
    else
      {
        printf ("No es posible crear directorio\n");
        exit (1);
      }

    getch ();
    system ("dir/p");
    getch ();

    estado = rmdir (NOMBREDIR);
    if (! estado)
      printf ("\nDirectorio borrado\n");
    else
      {
        perror ("\nNo es posible borrar directorio\n");
        exit (1);
      }

    return 0;
  }


------------
  searchpath    Busca el path del DOS para un determinado fichero.
============

Sintaxis:
   char *searchpath (const char *fichero);

Devuelve un puntero a una cadena con el nombre de path completo de fichero
si tiene éxito; en otro caso, devuelve NULL. Esta cadena está en un área
```

estática que es sobreescrita en cada nueva llamada.

Ejemplo:

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

  int main (void)
  {
    char *p;

    /* Busca TLINK.EXE y devuelve un puntero al path */
    p = searchpath ("TLINK.EXE");
    printf ("Búsqueda para TLINK.EXE: %s\n", p);

    /* Busca un fichero no existente */
    p = searchpath ("NOEXISTE.FIL");
    printf ("Búsqueda para NOEXISTE.FIL: %s\n", p);

    return 0;
  }
```

---

setdisk Pone la unidad de disco actual.

Sintaxis:

```
   int setdisk (int unidad);
```

Los valores pra unidad son

```
  0 = A
  1 = B
  2 = C
  etc.
```

Devuelve el número total de unidades disponibles.

Ejemplo:

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

  int main (void)
  {
    int salvar, disco, discos;

    /* salvar unidad original */
    salvar = getdisk ();

    /* imprimir número de unidades lógicas */
    discos = setdisk (salvar);
    printf ("%d unidades lógicas en el sistema\n\n", discos);

    /* imprime las letras de unidad disponible */
    printf ("Unidades disponibles:\n");
    for (disco = 0; disco < 26; ++disco)
      {
        setdisk (disco);
        if (disco == getdisk ())
          printf("%c: unidad disponible\n", disco + 'a');
      }
    setdisk (salvar);
```

```
    return 0;
  }


CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
************************************************

----------------
  FFBLK (struct)
================

Estructura de bloque de control de fichero del DOS.

  struct ffblk
    {
      char      ff_reserved[21];
      char      ff_attrib;
      unsigned  ff_ftime;
      unsigned  ff_fdate;
      long      ff_fsize;
      char      ff_name[13];
    };


--------------------
  fnsplit (#defines)
====================

Definiciones de bits devueltos por fnsplit para identificar qué partes de
un nombre de fichero fueron encontrados durante la descomposición.

  WILDCARDS   El Path contiene caracteres comodín
  EXTENSION   El path incluye extensión
  FILENAME    El path incluye un nombre de fichero
  DIRECTORY   El path incluye un subdirectorio
  DRIVE       El path incluye una unidad
```

FUNCIONES DEL DOS

En el fichero **<dos.h>** de Turbo C nos encontramos información (declaración de funciones, constantes, tipos, estructuras, ...) relacionadas con el sistema operativo DOS. Todas estas funciones interactúan directamente con el sistema operativo y por ello no están definidas en el estándar ANSI.

FUNCIONES DE INTERRUPCION

Turbo C posee la palabra clave **interrupt** que define una función como un manejador de interrupción. La sintaxis de esta palabra clave es:

```
  interrupt definicion_de_funcion;
```

Las funciones de interrupción son llamadas por el sistema cuando se genera alguna interrupción. En estas funciones se salvan todos los registros de la CPU y se terminan con la instrucción IRET.

Cuando en Turbo C usemos la palabra clave interrupt en un programa tenemos que desactivar el chequeo de pila y el uso de variables registros en el entorno.

FICHERO DE CABECERA DOS.H (TC)

GLOSARIO:
****

---
absread y abswrite — absread() lee sectores absolutos de disco.
abswrite() escribe sectores absolutos de disco.
---
allocmem — Asigna memoria.
---
bdos — Invoca una función del DOS, forma corta.
---
bdosptr — LLamada al sistema MS-DOS.
---
country — Devuelve información dependiente del país.
---
ctrlbrk — Pone manejador de control-break.
---
delay — Suspende la ejecución durante un intervalo (en milisegundos).
---
disable — Inhabilita las interrupciones.
---
dosexterr — Obtiene información de error extendido del DOS.
---
dostounix — Convierte fecha y hora a formato de hora UNIX.

__emit__ — Inserta valores literales directamente dentro del código.
---
enable — Habilita las interrupciones hardware.
---
macros FP_OFF, FP_SEG y MK_FP — Obtiene el offset de una direc. far (FP_OFF)
Obtiene el segmento de una direc. far(FP_SEG)
Hace un puntero far (MK_FP).
---
freemem — Libera un bloque de momoria del DOS asignado previamente con allocmen().
---
geninterrupt — Macro que genera una interrupción software.
---
getcbrk — Obtiene el estado de control-break.

```
---------
  getdate    Obtiene fecha del sistema (DOS).
=========
----------
  getdfree    Obtiene el espacio libre de disco.
==========
--------
  getdta    Obtiene la dirección de transferenciad de disco.
========
--------
  getfat    Obtiene información sobre la tabla de asignación de ficheros
========   para la unidad dada.
---------
  getfatd    Obtiene información sobre la tabla de asignación de fichero.
=========
----------
  getftime    Obtiene la fecha y hora de un fichero.
==========
--------
  getpsp    Obtiene el prefijo de segmento de programa.
========
---------
  gettime    Obtiene la hora del sistema.
=========
---------
  getvect    Obtiene un vector de interrupción.
=========
-----------
  getverify    Obtiene el estado de verificacion.
===========
---------
  harderr    Establece un manejador de error hardware.
=========
------------
  hardresume    Función de manejador de error hardware.
============
----------
  hardretn    Manejador de error hardware.
==========
-----
  inp    Macro que lee un byte de un puerto hardware.
=====
---------
  inportb    Lee un byte de un puerto hardware.
=========
-------
  int86    Interrupción de software 8086.
=======
--------
  int86x    Interfase de interrupción software 8086.
========
--------
  intdos    Interfase de interrupción DOS.
========
---------
  intdosx    Interfase de interrupción DOS.
=========
------
  intr    Interfase de interrupción software 8086.
======
------
  keep    Termina y queda residente.
======
---------
```

nosound Desactiva el altavoz del PC.

outp Macro que escribe un byte en un puerto hardware.

outport Escribe una palabra en un puerto hardware.

outportb Escribe un byte en un puerto hardware.

parsfnm Analiza un nombre de fichero y construye un bloque de control de fichero (FCB).

peek Devuelve la palabra que hay en la localización de memoria especificada por segmento:desplazamiento.

peekb Devuelve el byte que hay en la localización de memoria especificada por segmento:desplazamiento.

poke Almacena un valor entero en la posición de memoria especificada por segmento:desplazamiento.

pokeb Almacena un byte en la posición de memoria especificada por segmento:desplazamiento.

randbrd Lee bloque aleatorio.

randbwr Escribe bloque aleatorio usando el bloque de control de fichero (FCB).

segread Lee registros del segmento.

setblock Modifica el tamaño de un bloque asignado previamente.

setcbrk Pone el estado de control-break.

setdate Pone la fecha del DOS.

setdta Pone la dirección de transferencia de disco.

settime Pone la hora del sistema.

setvect Pone entrada de un vector de interrupción.

setverify Pone el estado de verificación.

sleep Suspende la ejecución durante un intervalo (en segundos).

sound Activa el altavoz del PC a una frecuencia especificada.

unixtodos Convierte fecha y hora de formato UNIX a formato DOS.

```
==========
--------
  unlink    Borra un fichero.
========

FUNCIONES:
**********

----------------------
  absread  y  abswrite    absread() lee sectores absolutos de disco.
======================   abswrite() escribe sectores absolutos de disco.

Sintaxis:
   int absread (int drive, int nsects, long lsect, void *buffer);
   int abswrite (int drive, int nsects, long lsect, void *buffer);


þ drive   es 0 = A, 1 = B, 2 = C, etc.
þ nsects  es el número de sectores a leer/escribir
þ lsect   es el sector lógico de comienzo (0 es el primero)
þ buffer  es la dirección del área de datos

64K es la cantidad de memoria más grande por llamada que puede ser leída o
escrita.

Devuelve 0 si tiene éxito; en caso de error, devuelve -1 y pone en errno
el número de error.


----------
  allocmem    Asigna memoria.
==========

Sintaxis:
   int allocmem (unsigned tam, unsigned *pseg);

tam es el número de párrafos a asignar (un párrafo son 16 bytes). La direc-
ción del segmento del área asignada es almacenada en *pseg; el offset = 0.

Devuelve -1 si tiene éxito. En otro caso devuelve el tamaño del bloque
disponible más grande, y pone en _doserrno y errno el código de error.

Ejemplo:

  #include <dos.h>
  #include <alloc.h>
  #include <stdio.h>

  int main (void)
  {
    unsigned int tam, pseg;
    int estado;

    tam = 64; /* (64 x 16) = 1024 bytes */
    estado = allocmem (tam, &pseg);
    if (estado == -1)
      printf ("Memoria asignada en segmeto: %X\n", pseg);
    else
      printf ("Asignación fallida: el número máximo de párrafos disponibles"
              " es %u\n", estado);

    return 0;
  }
```

```
------
  bdos    Invoca una función del DOS, forma corta.
======

Sintaxis:
   int bdos (int dosfun, unsigned dosdx, unsigned dosal);

El valor devuelto de bdos() es el valor que pone en AX la llamada del
sistema.

Ejemplo:

  #include <stdio.h>
  #include <dos.h>

  /* Obtiene unidad actual como 'A', 'B', ... */
  char unidad_actual (void)
  {
    char unidadact;

    /* Obtiene disco actual como 0, 1, ... */
    unidadact = bdos (0x19, 0, 0);
    return ('A' + unidadact);
  }

  int main (void)
  {
    printf ("La unidad actual es %c:\n", unidad_actual ());
    return 0;
  }


---------
  bdosptr    LLamada al sistema MS-DOS.
=========

Sintaxis:
   int bdosptr (int dosfun, void *argument, unsigned dosal);

El valor devuelto por bdosptr() es el valor de AX si tiene éxito, o -1 si
falla. En caso de fallo, se pone errno y _doserrno con el código de error.


---------
  country    Devuelve información dependiente del país.
=========

Sintaxis:
   struct COUNTRY *country (int xcode, struct COUNTRY *pc);

Devuelve el puntero pc.

Ejemplo:

  #include <stdio.h>
  #include <dos.h>

  #define USA 0

  void main (void)
  {
    struct COUNTRY pc;
```

```
    country (USA, &pc);

    printf ("\nco_date: %d", pc.co_date);
    printf ("\nco_curr: %s", pc.co_curr);
    printf ("\nco_thsep: %s", pc.co_thsep);
    printf ("\nco_desep: %s", pc.co_desep);
    printf ("\nco_dtsep: %s", pc.co_dtsep);
    printf ("\nco_tmsep: %s", pc.co_tmsep);
  }


---------
  ctrlbrk    Pone manejador de control-break.
=========

Sintaxis:
   void ctrlbrk (int (*manejador) (void));

La función manejador devuelve 0 para abortar el programa actual; en otro
caso el programa continuará la ejecución.

Ejemplo:

  #include <stdio.h>
  #include <dos.h>

  #define ABORT 0

  int c_break (void)
  {
    printf ("Control-Break presionado.  Abortando programa ...\n");
    return (ABORT);
  }

  void main (void)
  {
    ctrlbrk (c_break);
    for (;;)
      {
        printf ("Bucle... Presiona <Ctrl-Break> para salir:\n");
      }
  }


-------
  delay    Suspende la ejecución durante un intervalo (en milisegundos).
=======

Sintaxis:
   void delay (unsigned milisegundos);


---------
  disable    Inhabilita las interrupciones.
=========

Sintaxis:
   void disable (void);

Inhabilita todas las interrupciones hardware excepto NMI.


-----------
  dosexterr    Obtiene información de error extendido del DOS.
```

==========

Sintaxis:
   int dosexterr (struct DOSERROR *eblkp);

Rellena los campos de *eblkp basados en la última llamada al DOS. Devuelve
el valor del campo de_exterror de la estructura.

----
  dostounix   Convierte fecha y hora a formato de hora UNIX.
==========

Sintaxis:
   long dostounix (struct date *d, struct time *t);

Devuelve la versión de UNIX de la hora actual: número de segundos desde el
1 de Enero de 1970 (GMT).

  __emit__   Inserta valores literales directamente dentro del código.
==========

Sintaxis:
   void __emit__ (argument, ...);

Ejemplo:

```
  #include <dos.h>

  int main (void)
  {
    /*
      Emite código que generará una int 5 (imprimir pantalla)
    */
    __emit__ (0xcd,0x05);
    return 0;
  }
```

----
  enable   Habilita las interrupciones hardware.
========
Sintaxis:
   void enable (void);

Ejemplo:

```
  /* NOTA: Cuando se utilizan rutinas de servicio de interrupción,
     no se puede compilar el programa con la opción de testear el
     desbordamiento de pila puesta a on y obtener un programa eje-
     cutable que opere correctamente */

  #include <stdio.h>
  #include <dos.h>
  #include <conio.h>

  #define INTR 0X1C  /* interrupción de pulso de reloj */

  void interrupt (*viejo_manejador) (void);

  int cont = 0;
```

```
  void interrupt manejador (void)
  {
    /* inhabilita las interrupciones mientras se maneja la interrupción */
    disable ();
    /* incrementa el contador global */
    cont++;
    /* vuelve a habilitar las interrupciones al final del manejador */
    enable ();
    /* llama a la rutina original */
    viejo_manejador ();
  }

  int main (void)
  {
    /* salva el vector de interrupción original */
    viejo_manejador = getvect (INTR);

    /* instala el nuevo manejador de interrupción */
    setvect (INTR, manejador);

    /* bucle hasta mientras el contador no exceda de 20 */
    while (cont < 20)
      printf ("cont es %d\n", cont);

    /* vuelve a poner el manejador de interrupción original */
    setvect (INTR, viejo_manejador);

    return 0;
  }
```

---------------------------------þObtiene el offset de una direc. far (FP_OFF)
macros FP_OFF, FP_SEG y MK_FP þObtiene el segmento de una direc. far(FP_SEG)
==================================þHace un puntero far (MK_FP).

Sintaxis:

```
  unsigned FP_OFF (void far *p);
  unsigned FP_SEG (void far *p);
  void far *MK_FP (unsigned seg, unsigned ofs);
```

----
  freemem   Libera un bloque de momoria del DOS asignado previamente con
=========   allocmen().

Sintaxis:

```
   int freemem (unsigned segx);
```

Devuelve 0 si tiene éxito; -1 en caso de error y se pone en errno el código de error.

----
  geninterrupt   Macro que genera una interrupción software.
==============

Sintaxis:

```
   void geninterrupt (int num_interrup);
```

El estado de todos los registros después de la llamada es dependiente de la interrupción llamada. Cuidado: las interrupciones pueden dejar registros usados por Turbo C en un estado impredecible.

---
  getcbrk   Obtiene el estado de control-break.
=========

Sintaxis:
   int getcbrk (void);

Devuelve 0 si el chequeo de control-break está off y 1 si el chequeo
está on.

Ejemplo:

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

  int main (void)
  {
    if (getcbrk ())
      printf ("Cntrl-brk está on\n");
    else
      printf ("Cntrl-brk está off\n");

    return 0;
  }
```

---
  getdate   Obtiene fecha del sistema (DOS).
=========

Sintaxis:
   void getdate (struct date *pfecha);

Ejemplo:

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

  int main (void)
  {
    struct date d;

    getdate (&d);
    printf ("El día actual es: %d\n", d.da_day);
    printf ("El mes actual es: %d\n", d.da_mon);
    printf ("El año actual es es: %d\n", d.da_year);
    return 0;
  }
```

---
  getdfree   Obtiene el espacio libre de disco.
==========

Sintaxis:
   void getdfree (unsigned char drive, struct dfree *dtable);

En caso de error, a df_sclus en la estructura dfree se la da el valor de
0xFFFF.

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

```
  #include <dos.h>

  int main (void)
  {
    struct dfree free;
    long disponible;
    int drive;

    drive = getdisk ();
    getdfree (drive+1, &free);
    if (free.df_sclus == 0xFFFF)
      {
        printf ("Error en la llamada a getdfree()\n");
        exit (1);
      }

     disponible = (long) free.df_avail * (long) free.df_bsec *
                  (long) free.df_sclus;
     printf ("El drive %c tiene %ld bytes disponibles\n", 'A' + drive,
             disponible);

     return 0;
  }
```

----
  getdta   Obtiene la dirección de transferenciad de disco.
========

Sintaxis:
```
   char far *getdta (void);
```

Devuelve un puntero a la dirección de transferencia de disco actual.

Ejemplo:

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

  int main (void)
  {
    char far *dta;

    dta = getdta ();
    printf ("La dirección de transferencia del disco actual es: %Fp\n", dta);
    return 0;
  }
```

----
  getfat   Obtiene información sobre la tabla de asignación de ficheros
========   para la unidad dada.

Sintaxis:
```
   void getfat (unsigned char drive, struct fatinfo *dtable);
```

----
  getfatd   Obtiene información sobre la tabla de asignación de fichero.
=========

Sintaxis:
```
   void getfatd (struct fatinfo *dtable);
```

---
  getftime   Obtiene la fecha y hora de un fichero.
==========

Sintaxis:
   int getftime (int descriptor, struct ftime *pftime);

Devuelve 0 en caso de éxito, y -1 en caso de error y se pone en errno el
código de error.

---
  getpsp   Obtiene el prefijo de segmento de programa.
========

Sintaxis:
   unsigned getpsp (void);

getpsp() sólo puede ser llamada usando DOS 3.0 o superior.

---
  gettime   Obtiene la hora del sistema.
=========

Sintaxis:
   void gettime (struct time *phora);

Ejemplo:

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

  int main (void)
  {
    struct  time t;

    gettime (&t);
    printf ("La hora actual es: %2d:%02d:%02d.%02d\n",
            t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
    return 0;
  }
```

---
  getvect   Obtiene un vector de interrupción.
=========

Sintaxis:
   void interrupt (*getvect (int num_interrupcion)) ();

Devuelve el valor de 4 bytes almacenado en el vector de interrupción
nombrado por num_interrupcion.

---
  getverify   Obtiene el estado de verificacion.
==========

Sintaxis:
   int getverify (void);

Devuelve 0 si el estado de verificación está off, y 1 si el estado de

verificación está on.

Ejemplo:

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

  int main (void)
  {
    if (getverify ())
      printf ("El estado de verificacion del DOS está on\n");
    else
      printf ("El estado de verificación del DOS está off\n");
    return 0;
  }
```

---
harderr Establece un manejador de error hardware.

Sintaxis:

```
   void harderr (int (*manejador) ());
```

La función apuntada por manejador será llamada cuando el DOS encuentre un error crítico (INT 0x24).

Ejemplo:

```
  /*
    Este programa atrapa los errores de disco y pregunta al usuario la
    acción a realizar. Intenta ejecutarlo con ningún disco en la unidad A:
    para invocar sus funciones.
  */

  #include <stdio.h>
  #include <conio.h>
  #include <dos.h>

  #define IGNORAR    0
  #define REINTENTAR 1
  #define ABORTAR    2

  int buf[500];

  /* define los mensajes de error para los problemas de disco */
  static char *err_msj[] =
    {
      "protección contra escritura",
      "unidad desconocida",
      "unidad no preparada",
      "comando desconodido",
      "error de datos (CRC)",
      "respuesta mala",
      "error de búsqueda",
      "tipo de medio desconocido",
      "sector no encontrado",
      "impresora sin papel",
      "fallo de escritura",
      "fallo de lectura",
      "fallo general",
      "reservado",
      "reservado",
      "cambio de disco inválido"
```

```
  };

int error (char *msj)
{
  int valdev;

  cputs (msj);

  /* pide al usuario que presione una tecla para abortar, reintentar o
     ignorar */
  while (1)
    {
      valdev = getch ();
      if (valdev == 'a' || valdev == 'A')
        {
          valdev = ABORTAR;
          break;
        }
      if (valdev == 'r' || valdev == 'R')
        {
          valdev = REINTENTAR;
          break;
        }
     if (valdev == 'i' || valdev == 'I')
       {
         valdev = IGNORAR;
         break;
       }
   }

  return (valdev);
}

/* pragma warn -par reduce los warnings (avisos) que ocurren debidos al
   no uso de los parámetros errval, bp y si en manejador() */
#pragma warn -par

int manejador (int errval, int ax, int bp, int si)
{
  static char msj[80];
  unsigned di;
  int unidad;
  int numerror;

  di= _DI;
  /*si no es un error de disco entonces fue otro problema de dispositivo*/
  if (ax < 0)
    {
      /* informa del error */
      error ("Error de dispositivo");
      /* y vuelve al programa directamente requiriendo abortar */
      hardretn (ABORTAR);
    }
  /* en otro caso fue un error de disco */
  unidad = ax & 0x00FF;
  numerror = di & 0x00FF;
  /* informa del error */
  sprintf (msj, "Error: %s en unidad %c\r\nA)bortar, R)eintentar, "
           "I)gnorar: ", err_msj[numerror], 'A' + unidad);
  /* vuelve al programa vía interrupción dos 0x23 con abortar, reintentar
     o ignorar según elige usuario */
  hardresume (error (msj));
  return ABORTAR;
}
```

```
  #pragma warn +par

  int main (void)
  {
    /* instala nuestro manejador de la interrupción de problemas hardware */
    harderr (manejador);
    clrscr ();
    printf ("Asegúrate que no hay ningún disco en unidad A:\n");
    printf ("Presiona una tecla ...\n");
    getch ();
    printf ("Intentando acceder a la unidad A:\n");
    printf ("fopen() devolvió %p\n", fopen ("A:temp.dat", "w"));
    return 0;
  }
```

---

hardresume Función de manejador de error hardware.

Sintaxis:

```
   void hardresume (int axret);
```

El manejador de error establecido por harderr() puede devolver el control de la ejecución a la rutina del COS que provocó el error crítico vía esta función. El valor en axret es devuelto al DOS.

Valor devuelto	Significado
0	ignorar
1	reintentar
2	abortar

Ver ejemplo en la función harderr().

---

hardretn Manejador de error hardware.

Sintaxis:

```
   void hardretn (int retn);
```

El manejador de error establecido por harderr puede volver directamente al programa de aplicación llamando a hardretn().

El valor en retn es devuelto al programa de usuario en lugar del valor normal devuelto de la función DOS que generó el error crítico.

Ver ejemplo de función harderr().

---

inp Macro que lee un byte de un puerto hardware.

Sintaxis:

```
   int inp (int portid);
```

---

inportb Lee un byte de un puerto hardware.

Sintaxis:
```
   unsigned char inportb (int portid);
```

---
int86 Interrupción de software 8086.

Sintaxis:
```
   int int86 (int intno, union REGS *inregs, union REGS *outregs);
```

Esta función carga los registros de la CPU con los valores almacenados en inregs, ejecuta la interrupción intno, y almacena los valores resultados de los registros de la CPU en outregs.

Ejemplo:

```
  #include <stdio.h>
  #include <conio.h>
  #include <dos.h>

  #define VIDEO 0x10

  void movetoxy (int x, int y)
  {
    union REGS regs;

    regs.h.ah = 2;  /* pone la posición del cursor */
    regs.h.dh = y;
    regs.h.dl = x;
    regs.h.bh = 0;  /* página de vídeo 0 */
    int86 (VIDEO, ®s, ®s);
  }

  int main (void)
  {
    clrscr ();
    movetoxy (35, 10);
    printf ("Hola");
    return 0;
  }
```

---
int86x Interfase de interrupción software 8086.

Sintaxis:
```
   int int86x (int intno, union REGS *inregs, union REGS *outregs,
               struct SREGS *segregs);
```

Esta función carga los registros de la CPU con los valores almacenados en inregs y segregs, ejecuta la interrupción intno, y almacena los valores resultados de los registros de la CPU en outregs y segregs.

---
intdos Interfase de interrupción DOS.

Sintaxis:
```
   int intdos (union REGS *inregs, union REGS *outregs);
```

Esta función carga los registros de la CPU con los valores almacenados en

inregs, ejecuta la interrupción DOS (int 33 o 0x21), y almacena los valores resultados de los registros de la CPU en outregs.

---
intdosx Interfase de interrupción DOS.

Sintaxis:
```
int intdosx (union REGS *inregs, union REGS *outregs,
             struct SREGS *segregs);
```

Esta función carga los registros de la CPU con los valores almacenados en inregs y segregs, ejecuta la interrupción DOS (int 0x21), y almacena los valores resultados de los registros de la CPU en outregs y segregs.

---
intr Interfase de interrupción software 8086.

Sintaxis:
```
void intr (int intno, struct REGPACK *preg);
```

Esta función carga los registros de la CPU con los valores almacenados en preg, ejecuta la interrupción intno, y almacena los valores resultados de los registros de la CPU en preg.

---
keep Termina y queda residente.

Sintaxis:
```
void keep (unsigned char estado, unsigned tamanio);
```

Esta función vuelve al DOS con valor de salida en estado, pero el programa queda en memoria. La porción de programa residente ocupa tamanio párrafos y la memoria del programa restante es liberada.

---
nosound Desactiva el altavoz del PC.

Sintaxis:
```
void nosound (void);
```

---
outp Macro que escribe un byte en un puerto hardware.

Sintaxis:
```
int outp (int portid, int byte_value);
```

---
outport Escribe una palabra en un puerto hardware.

Sintaxis:
```
void outport (int portid, int value);
```

---
  outportb   Escribe un byte en un puerto hardware.
==========

Sintaxis:
   void outportb (int portid, unsigned char value);

---
  parsfnm   Analiza un nombre de fichero y construye un bloque de control
=========   de fichero (FCB).

Sintaxis:
   char *parsfnm (const char *cmdline, struct fcb *fcb, int opt);

Después de analizar con éxito el nombre del fichero, parsfnm() devuelve un
puntero al byte siguiente después del final del nombre del fichero. Si no
se produce ningún error en el análisis del nombre del fichero, devuelve 0.

---
  peek   Devuelve la palabra que hay en la localización de memoria
======   especificada por segmento:desplazamiento.

Sintaxis:
   int peek (unsigned segmento, unsigned desplazamiento);

---
  peekb   Devuelve el byte que hay en la localización de memoria
=======   especificada por segmento:desplazamiento.

Sintaxis:
   char peekb (unsigned segmento, unsigned desplazamiento);

---
  poke   Almacena un valor entero en la posición de memoria especificada
======   por segmento:desplazamiento.

Sintaxis:
   void poke(unsigned segmento, unsigned desplazamiento, int valor);

---
  pokeb   Almacena un byte en la posición de memoria especificada
=======   por segmento:desplazamiento.

Sintaxis:
   void pokeb (unsigned segmento, unsigned desplazamiento, char valor);

---
  randbrd   Lee bloque aleatorio.
=========

Sintaxis:
   int randbrd (struct fcb *fcb, int rcnt);

Si devuelve 0 todos los registros han sido leídos correctamente, si
devuelve otro valor ha habido algún problema.

---
  randbwr   Escribe bloque aleatorio usando el bloque de control de
=========   fichero (FCB).

Sintaxis:

```
   int randbwr (struct fcb *fcb, int rcnt);
```

Si devuelve 0 todos los registros han sido escritos correctamente, si devuelve otro valor ha habido algún problema.

---
segread Lee registros del segmento.

Sintaxis:

```
   void segread (struct SREGS *segp);
```

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

  int main (void)
  {
    struct SREGS segs;

    segread (&segs);
    printf ("Valores de los registros del segmento actual\n");
    printf ("CS: %X   DS: %X\n", segs.cs, segs.ds);
    printf ("ES: %X   SS: %X\n", segs.es, segs.ss);

    return 0;
  }
```

---
setblock Modifica el tamaño de un bloque asignado previamente.

Sintaxis:

```
   int setblock (unsigned segx, unsigned nuevotam);
```

Usa los bloques asignados con allocmem(). Devuelve -1 si tiene éxito.

En caso de error, devuelve el tamaño del bloque posible más grande y a _doserrno se le asigna el código de error.

---
setcbrk Pone el estado de control-break.

Sintaxis:

```
   int setcbrk (int valorcbrk);
```

Si valorcbrk es 1, chequea el Ctrl-Break en cada llamada al sistema. Si es 0, chequea sólo en las llamadas de E/S de consola, impresora y comunicaciones.

Devuelve el valor pasado en valorcbrk.

---
setdate Pone la fecha del DOS.

Sintaxis:

```
   void setdate (struct date *pfecha);
```

```
  setdta    Pone la dirección de transferencia de disco.
========

Sintaxis:
   void setdta (char far *dta);

---------
  settime    Pone la hora del sistema.
=========

Sintaxis:
   void settime (struct time *phora);

---------
  setvect    Pone entrada de un vector de interrupción.
=========

Sintaxis:
   void setvect (int interruptno, void interrupt (*isr) ( ));
```

isr apunta a una función que será llamada cuando ocurra el número de interrupción interrruptno. Si isr es una función C, debería ser definida con la palabra clave interrupt.

Ejemplo:

```
  /* NOTA: Cuando se utilizan rutinas de servicio de interrupción,
     no se puede compilar el programa con la opción de testear el
     desbordamiento de pila puesta a on y obtener un programa eje-
     cutable que opere correctamente */

  #include <stdio.h>
  #include <dos.h>
  #include <conio.h>

  #define INTR 0X1C  /* interrupción de tick de reloj */

  void interrupt (*viejo_manejador) (void);

  int cont = 0;

  void interrupt manejador (void)
  {
    /* incrementa el contador global */
    cont++;

    /* llama a la rutina original */
    viejo_manejador ();
  }

  int main (void)
  {
    /* salva el vector de interrupción original */
    viejo_manejador = getvect (INTR);

    /* instala el nuevo manejador de interrupción */
    setvect (INTR, manejador);

    /* bucle hasta mientras el contador no exceda de 20 */
    while (cont < 20)
      printf ("cont es %d\n", cont);

    /* vuelve a poner el manejador de interrupción original */
    setvect (INTR, viejo_manejador);
```

```
    return 0;
  }
```

---
  setverify   Pone el estado de verificación.
==========

Sintaxis:
   void setverify (int valor);

Si valor=1, cada operación de escritura en disco será seguida por una
operación de lectura para asegurar resultados correctos. (0 significa
no lectura de comprobación.)

---
  sleep   Suspende la ejecución durante un intervalo (en segundos).
=======

Sintaxis:
   void sleep (unsigned segundos);

Ejemplo:

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

  int main (void)
  {
     int i;

     for (i = 1; i < 5; i++)
       {
         printf ("Ejecución suspendida durante %d segundos.\n", i);
         sleep (i);
       }
     return 0;
  }
```

---
  sound   Activa el altavoz del PC a una frecuencia especificada.
=======

Sintaxis:
   void sound (unsigned frequencia);

frecuencia está en herzios (ciclos por segundo)

---
  unixtodos   Convierte fecha y hora de formato UNIX a formato DOS.
==========

Sintaxis:
   void unixtodos (long time, struct date *d, struct time *t);

---
  unlink   Borra un fichero.
========

Sintaxis:
```
   int unlink (const char *nombre_de_fichero);
```

Si el fichero nombre_de_fichero tiene el atributo de sólo lectura, unlink() fallará. En este caso es necesario llamar primero a chmod() para cambiar el atributo del fichero.

Devuelve 0 si tiene éxito; -1 en caso de error.

El prototipo de esta función también se encuentra en los ficheros io.h y stdio.h

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:

int _8087 (variable global)

Chip de coprocesador actual.

Si el programa está ejecutándose en una máquina con coprocesador matemático, _8087 es distinto de cero:

valor _8087	Coprocesador matemático
1	8087
2	80287
3	80387
0	(no detectado)

_argc (variable global)

Contiene el número de argumentos en la línea de comandos.

```
  extern int _argc;
```

_argc tiene el valor de argc pasado a main cuando empieza el programa.

_argv (variable global)

Array de punteros a los argumentos de la línea de comandos.

```
  extern char *_argv[]
```

BYTEREGS (struct)
WORDREGS (struct)

Estructuras para almacenamientos de registros de bytes y palabras.

```
  struct  BYTEREGS
    {
```

```
      unsigned char  al, ah, bl, bh;
      unsigned char  cl, ch, dl, dh;
    };

  struct  WORDREGS
    {
      unsigned int  ax, bx, cx, dx;
      unsigned int  si, di, cflag, flags;
    };
```

COUNTRY (struct)

La estructura COUNTRY especifica cómo se van a formatear ciertos datos dependientes del país.

```
  struct COUNTRY
    {
      int   co_date; /* formato de fecha */
      char  co_curr[5]; /* símbolo de moneda */
      char  co_thsep[2]; /* separador de millar */
      char  co_desep[2]; /* separador decimal */
      char  co_dtsep[2]; /* separador de fecha */
      char  co_tmsep[2]; /* separador de tiempo */
      char  co_currstyle; /* estilo de moneda */
      char  co_digits; /* dígitos significativos en moneda */
      char  co_time;
      long  co_case;
      char  co_dasep[2]; /* separador de datos */
      char  co_fill[10];
   };
```

DATE (struct)

Estructura de la fecha usada por las funciones dostounix(), setdate(), getdate() y unixtodos().

```
  struct date
    {
      int   da_year;
      char  da_day;
      char  da_mon;
    };
```

DEVHDR (struct)

Estructura de cabecera para los controladores de dispositivos de MS-DOS.

```
  struct devhdr
    {
      long            dh_next;
      short           dh_attr;
      unsigned short  dh_strat;
      unsigned short  dh_inter;
      char            dh_name[8];
    };
```

DFREE (struct)

La estructura de la información devuelta por la función getdfree().

```
  struct dfree
    {
      unsigned df_avail; /* Clusters disponibles */
      unsigned df_total; /* Clusters totales */
      unsigned df_bsec;  /* Bytes por sector */
      unsigned df_sclus; /* Sectores por cluster */
    };
```

_doserrno (variable global)

Variable que contiene el código de error de DOS actual.

```
  int _doserrno;
```

Cuando en una llamada al sistema MS-DOS ocurre un error, a _doserrno se le asigna el código de error DOS actual.

Los mnemotécnicos para los códigos de error de DOS actual que pueden ser asignados a _doserrno son:

Mnemotécnico	Código de error del DOS.
E2BIG	Entorno malo
EACCES	Acceso denegado
EACCES	Acceso malo
EACCES	Es directorio corriente
EBADF	Manejador malo
EFAULT	Reservado
EINVAL	Datos malos
EINVAL	Función mala
EMFILE	Demasiados ficheros abiertos
ENOENT	No es fichero ni directorio
ENOEXEC	Formato malo
ENOMEM	MCB destruido
ENOMEM	Fuera de memoria
ENOMEM	Bloque malo
EXDEV	Unidad mala
EXDEV	No es el mismo dispositivo

Esta variable también está declarada en los ficheros errno.h y stdlib.h.

---
dosSearchInfo (tipo)

```
typedef struct
   {
     char   drive;
     char   pattern [13];
     char   reserved [7];
     char   attrib;
     short  time;
```

```
     short  date;
     long   size;
     char   nameZ [13];
   } dosSearchInfo;
```

---

environ (variable global)

Array de cadenas usado para acceder y alterar el entorno del proceso.

```
  extern char **environ
```

También se encuentra declarada en el fichero stdlib.h.

FATINFO (struct)

La estructura de información rellenada por las funciones getfat() y getfatd().

```
  struct fatinfo
    {
      char  fi_sclus; /* sectores por cluster */
      char  fi_fatid; /* el byte identificador de la FAT */
      int   fi_nclus; /* número de clusters */
      int   fi_bysec; /* bytes por sector */
    };
```

FCB (struct)

La estructura de los bloques de control de ficheros de MS-DOS.

```
  struct fcb
    {
      char   fcb_drive;
      char   fcb_name[8],  fcb_ext[3];
      short  fcb_curblk,   fcb_recsize;
      long   fcb_filsize;
      short  fcb_date;
      char   fcb_resv[10], fcb_currec;
      long   fcb_random;
    };
```

FA_xxxx (#defines)

Atributos de fichero de MS-DOS.

```
  FA_RDONLY   Atributo de sólo lectura.
  FA_HIDDEN   Fichero oculto.
  FA_SYSTEM   Fichero de sistema.
  FA_LABEL    Etiqueta de la unidad.
  FA_DIREC    Directorio.
  FA_ARCH     Archivo.
```

  _heaplen (variable global)
==========

Tamaño inicial del montón en bytes.

```
  unsigned _heaplen
```

El valor de _heaplen al principio de la ejecución del programa determina el tamaño del montón near que será asignado. El valor de 0, por defecto, hace un montón de tamaño máximo.

_heaplen no es usado en los modelos de datos grandes.

  NFDS (#define)
==========

Número máximo de descriptores de fichero.

  _osmajor (variable global)
  _osminor (variable global)
  _version (variable global)
==========

La versión de MS-DOS bajo la cual está corriendo el programa actualmente.

```
  unsigned char _osmajor /* número de versión mayor */
  unsigned char _osminor /* número de versión menor */
  unsigned int  _version /* número de versión completa */
```

  _psp (variable global)
==========

Dirección del segmento del PSP (Prefijo de Segmento de Programa) del programa.

```
  extern unsigned int _psp;
```

También está declarada en los ficheros process.h y stdlib.h.

  REGPACK (struct)
==========

La estructura de los valores pasados y devueltos en la función intr().

```
  struct REGPACK
    {
      unsigned  r_ax, r_bx, r_cx, r_dx;
      unsigned  r_bp, r_si, r_di;
      unsigned  r_ds, r_es, r_flags;
    };
```

  REGS (union)

==========

La unión REGS es usada para pasar y recibir información en las funciones intdos(), intdosx(), int86() y int86x().

```
  union REGS
    {
      struct  WORDREGS  x;
      struct  BYTEREGS  h;
    };
```

  SREGS (struct)
==========

La estructura de los registros de segmentos pasadas y rellenadas en las funciones intdosx(), int86x() y segread().

```
  struct SREGS
    {
      unsigned int es;
      unsigned int cs;
      unsigned int ss;
      unsigned int ds;
    };
```

  _stklen (variable global)
==========

Variable con longitud de la pila.

```
  extern unsigned _stklen;
```

_stklen especifica el tamaño de la pila. El tamaño de la pila por defecto es 4K.
_stklen es usado antes de que se llame a la función main().

Para poner un tamaño de pila diferente, debes declarar _stklen en tu fichero fuente como una variable global. (Colócala fuera de todas las funciones.)

Por ejemplo, para poner un tamaño de pila de 20000 bytes, usa la siguiente declaración:

```
  unsigned _stklen = 20000;
```

  TIME (struct)
==========

Estructura de la hora usada por las funciones dostounix(), gettime(), settime() y unixtodos().

```
  struct time
    {
      unsigned char ti_min;
      unsigned char ti_hour;
      unsigned char ti_hund;
      unsigned char ti_sec;
    };
```

```
------------------
  BYTEREGS (struct)
  WORDREGS (struct)
==================
```

Estructuras para el almacenamiento de registros de bytes y palabras.

```
  struct BYTEREGS
    {
      unsigned char  al, ah, bl, bh;
      unsigned char  cl, ch, dl, dh;
    };

  struct WORDREGS
    {
      unsigned int  ax, bx, cx, dx;
      unsigned int  si, di, cflag, flags;
    };
```

```
---------------
  XFCB (struct)
===============
```

La estructura de bloque de control de fichero extendido de MS-DOS.

```
  struct xfcb
    {
      char         xfcb_flag;
      char         xfcb_resv[5];
      char         xfcb_attr;
      struct  fcb  xfcb_fcb;
    };
```

FUNCIONES DE LA ROM BIOS

Cada una de las funciones declaradas en el fichero de cabecera **<bios.h>** de Turbo C representa una rutina de servicio de la ROM BIOS. Todas estas funciones se pueden implementar con las funciones int86() y int86x() del fichero de cabecera <dos.h>. Para conocer más detalles sobre las funciones del fichero bios.h tendrás que consultar un manual en el que se describan las funciones de la ROM BIOS.

FICHERO DE CABECERA BIOS.H (TC)

```
---------
  bioscom    E/S de comunicaciones RS-232
=========
```

Sintaxis:

```
   int bioscom (int cmd, char byte, int puerto);
```

Los valores para cmd son los siguientes:

```
  þ 0  pone los parámetros de comunicación con el valor en byte
```

```
  þ 1  envía el carácter que está en byte a la línea de comunicaciones
  þ 2  recibe un carácter de la línea de comunicaciones (en los 8 bits
       menos significativos del valor devuelto)
  þ 3  devuelve el estado actual del puerto de comunicaciones

El puerto 0 es COM1, el 1 es COM2, etc.

Los 8 bits más significativos del valor devuelto son los bits de estado.
Los 8 bits menos significativos dependen del cmd especificado.


----------
  biosdisk    Servicios de disco de la BIOS.
==========

Sintaxis:
   int biosdisk (int cmd, int unidad, int cabeza, int pista, int sector,
                 int nsects, void *buffer);

Devuelve 0 si tiene éxito; en otro caso devuelve un código de error.

Para saber los valores que puede tomar cmd y el significado de cada uno
de ellos consulta la interrupción 13 hex. de la ROM BIOS.


-----------
  biosequip    Chequea equipo.
===========

Sintaxis:
   int biosequip (void);

Devuelve los indicadores de equipo de la BIOS.


---------
  bioskey    Interface de teclado.
=========

Sintaxis:
   int bioskey (int cmd);

cmd                                   Acción
================================================================================
  0  | Devuelve el código de exploración de la tecla que hay en el buffer
     | y la quita de él. Espera la pulsación de la tecla si el buffer
     | está vacío
     |
  1  | Devuelve el código de exploración de la tecla que hay en el buffer
     | pero no la quita de él. Devuelve 0 si el buffer está vacío. Si fue
     | presionada la combinación de teclas Ctrol-Break, bioskey() devuelve
     | -1 (0xFFFF).
     |
  2  | Devuelve los indicadores de estado de las teclas de cambio de la BIOS.
     | (teclas shift izquierda y derecha, control, alt)


------------
  biosmemory    Devuelve el tamaño de la memoria.
============

Sintaxis:
   int biosmemory (void);
```

El valor devuelto es el tamaño de la memoria en bloques de 1K.

Ejemplo:

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

  void main (void)
  {
    printf("Tamaño de la memoria RAM: %d Kbytes\n", biosmemory ());
  }
```

---
  biosprint   E/S de impresora usando directamente la BIOS.
==========

Sintaxis:
```
   int biosprint (int cmd, int byte, int puerto);
```

Si cmd es 0, si imprime el byte.
Si cmd es 1, se inicializa el puerto de impresora.
Si cmd es 2, se lee el estado actual de la impresora.

Se devuelve el estado actual de la impresora para cualquier valor de cmd.

---
  biostime   Rutina del servicio de reloj de la BIOS.
==========

Sintaxis:
```
   long biostime (int cmd, long nuevahora);
```

Si cmd es 0, lee la hora de la BIOS.
Si cmd es 1, pone la hora de la BIOS.
La hora está en pulsos de reloj desde la medianoche.
Un segundo tiene 18.2 pulsos.

Ejemplo:

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

  void main (void)
  {
    printf ("El número de pulsos de reloj desde la medianoche es: %lu\n",
            biostime (0, 0L));
  }
```

LECCIÓN

INDICE DE LA LECCION 12

- Primera parte:

  * Funciones varias (**stdlib.h**).
  * Funciones de fecha y hora (**time.h**).
  * Funciones relacionadas con información geográfica (**locale.h**).

- Segunda parte:

  * Función y estructura de hora actual (**sys\timeb.h**).
  * Funciones de información de ficheros (**sys\stat.h**).
  * Constantes simbólicas para compatibilidad con UNIX (**values.h**).
  * Funciones de coma flotante (**float.h**).
  * Conexión de Turbo C con ensamblador (**#pragma inline** y **asm**).

FUNCIONES VARIAS

El fichero **<stdlib.h>** es una especie de miscelánea de funciones, es decir, en él se encuentra declaradas una gran variedad de funciones: funciones de conversión de tipos, de ordenación, de búsqueda, de generación de números pseudoaleatorios, etc.

La cabecera stdlib.h también define los tipos **div_t** y **ldiv_t** que son los valores devueltos por **div()** y **ldiv()**, respectivamente. Además define las macros: **ERANGE** (valor asignado a errno si se produce un error de rango), **HUGE_VAL** (mayor valor representable por rutinas en coma flotante) y **RAND_MAX** (máximo valor que puede ser devuelto por la función rand()).

FICHERO DE CABECERA STDLIB.H

GLOSARIO:
****

---
abort Termina anormalmente un programa.
=======
---
abs Devuelve el valor absoluto de un entero.
=====
---
atexit Registra una función de terminación.
========
---
atof Convierte una cadena a un punto flotante.
======
---
atoi Convierte una cadena en un entero.
======
---
atol Convierte un string a un long.

======
----
  bsearch   Búsqueda binaria.
=========
----
  calloc   Asigna memoria principal.
========
-----
  div   Divide dos enteros.
=====
----
  ecvt y fcvt   (TC) Convierte número en coma flotante a cadena.
===============

------
  exit   Termina programa.
======

  _exit   Termina programa.
=======
------
  fcvt   (TC) [Ver ecvt()]
======

------
  free   Libera bloques asignados con malloc() o calloc().
======
------
  gcvt   (TC) Convierte un número en coma flotante a string.
======

----
  getenv   Obtiene un string del entorno.
========
------
  itoa   Convierte un entero a una cadena.
======
------
  labs   Calcula el valor absoluto de un long.
======
------
  ldiv   Divide dos longs, devuelve el cociente y el resto.
======
----
  lfind and lsearch   (TC) Ejecuta búsqueda lineal.
===================
-------------------   (TC)
  _lrotl and _lrotr   Rota un valor long a la izquierda (_lrotl).
===================   Rota un valor long a la derecha (_lrotr).
----
  lsearch   (TC) [Ver lfind()]
=========

------
  ltoa   Convierte un long a una cadena.
======
----
  malloc   Asigna memoria principal.
========
----
  max y min   (TC) Macros que generan código en línea para encontrar el
=============   valor máximo y mínimo de dos enteros.
----
  putenv   (TC) Añade una cadena al entorno actual.

```
========

-------
  qsort    Ordena usando el algoritmo quicksort (ordenación rápida).
=======
------
  rand    Generador de números aleatorios.
======
--------
  random    (TC) Macro que devuelve un entero.
========
-----------
  randomize    (TC) Macro que inicializa el generador de números aleatorios.
===========

---------
  realloc    Reasigna memoria principal.
=========
-----------------   (TC)
  _rotl and _rotr    Rota un valor unsigned int a la izquierda (_rotl).
=================   Rota un valor unsigned int a la derecha (_rotr).

-------
  srand    Inicializa el generador de números aleatorios.
=======
--------
  strtod    Convierte cadena a double.
========
--------
  strtol    Conviete cadena a long usando la base fijada.
========
---------
  strtoul    Convierte una cadena a un unsigned long con la base fijada.
=========
------
  swab    (TC) Intercambia bytes.
======

--------
  system    Ejecuta un comando DOS.
========
-------
  ultoa    (TC) Convierte un unsigned long a una cadena.
=======



FUNCIONES:
**********

-------
  abort    Termina anormalmente un programa.
=======

Sintaxis:
   void abort (void);

Ejemplo:

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

  int main (void)
  {
```

```
    printf ("Llamando a abort()\n");
    abort ();
    return 0; /* Esta línea nunca es ejecutada */
  }
```

---
  abs   Devuelve el valor absoluto de un entero.
=====

Sintaxis:
```
   int abs (int x);
```

Ejemplo:

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

  int main (void)
  {
    printf ("\nabs(4): %d", abs (4));
    printf ("\nabs(-5): %d", abs (-5));
    return 0;
  }
```

---
  atexit   Registra una función de terminación.
========

Sintaxis:
```
   int atexit (atexit_t func);
```

La función atexit() fija la función apuntada por func como la función a ser llamada una vez alcanzada la terminación normal del programa. Es decir, al final de la ejecución de un programa, se llama a la función especificada.

La función atexit() devuelve 0 si la función queda establecida como función de terminación; en cualquier otro caso devuelve un valor distinto de cero.

Se pueden fijar varias funciones de terminación siendo llamadas en orden inverso al de su establecimiento. En otras palabras, la naturaleza del proceso de registro es como una pila.

El tipo atexit_t está declarado del siguiente modo:

```
typedef void (* atexit_t) (void);
```

Ejemplo:

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

  void exit_fn1 (void)
  {
    printf ("Llamada función de salida número 1\n");
  }

  void exit_fn2 (void)
  {
    printf ("Llamada función de salida número 2\n");
  }

  int main (void)
```

```
  {
     atexit (exit_fn1);
     atexit (exit_fn2);
     return 0;
  }
```

```
------
  atof    Convierte una cadena a un punto flotante.
======
```

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

La función atof() convierte la cadena apuntada por s a un valor de tipo double. La cadena debe contener un número válido en como flotante. Si no es este el caso, se devuelve el valor 0.

Ejemplo:

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

  int main (void)
  {
    float f;
    char *str = "123.4ABC";

    f = atof (str);
    printf ("string = %s float = %f\n", str, f);
    return 0;
  }
```

```
------
  atoi    Convierte una cadena en un entero.
======
```

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

La función atoi() convierte la cadena apuntada por s a un valor int. La cadena debe contener un número entero válido. Si no es este el caso, se devuelve el valor 0.

Ejemplo:

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

  int main (void)
  {
    int n;
    char *str = "12345.67";

    n = atoi (str);
    printf ("string = %s integer = %d\n", str, n);
    return 0;
  }
```

```
------
  atol    Convierte un string a un long.
======
```

Sintaxis:
```
   long atol (const char *s);
```

La función atol() convierte la cadena apuntada por s a un valor long int. La cadena debe contener un número entero de tipo long válido. Si no es este el caso, se devuelve el valor 0.

Ejemplo:

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

  int main (void)
  {
     long l;
     char *str = "98765432";

     l = atol (str);
     printf ("string = %s integer = %ld\n", str, l);
     return (0);
  }
```

---
bsearch Búsqueda binaria.

Sintaxis:

```
  void *bsearch (const void *clave, const void *base, unsigned int *num,
    unsigned int tam, int (*compara) (const void *arg1, const void *arg2));
    unsigned int tam, int (*compara) (void *arg1, void *arg2));
```

La función bsearch() realiza una búsqueda binaria en el array ordenado apuntado por base y devuelve un puntero al primer elemento que se corresponde con la clave apuntada por clave. El número de elementos en el array está especificado por num y el tamaño (en bytes) de cada elemento está descrito por tam.

La función apuntada por compara se utiliza para comparar un elemento del array con la clave. La forma de la función de comparación debe ser:

```
  nombre_func (void *arg1, void *arg2);
```

Debe devolver los siguientes valores:

```
  Si arg1 es menor que arg2, devuelve un valor menor que 0.
  Si arg1 es igual que arg2, devuelve 0.
  Si arg1 es mayor que arg2, devuelve un valor mayor que 0.
```

El array debe estar ordenado en orden ascendente con la menor dirección conteniendo el elemento más pequeño. Si el array no contiene la clave, se devuelve un puntero nulo.

Esta función está implementada en uno de los ejemplos de la lección 3.

Ejemplo:

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

  #define NUM_ELEMENTOS(array) (sizeof(array) / sizeof(array[0]))
```

```
  int array_de_numeros[] = { 123, 145, 512, 627, 800, 933, 333, 1000 };

  int comparacion_de_numeros (const int *p1, const int *p2)
  {
    return (*p1 - *p2);
  }

  int buscar (int clave)
  {
     int *puntero_a_elemento;

     /* El molde (int (*) (const void *, const void *)) es necesario para
        evitar un error de tipo distinto en tiempo de compilación. Sin
        embargo, no es necesario: puntero_a_elemento = (int *) bsearch (...
        debido a que en este caso es el compilador el que realiza la
        conversión de tipos */
     puntero_a_elemento = bsearch (&clave, array_de_numeros,
       NUM_ELEMENTOS (array_de_numeros), sizeof (int),
       (int (*) (const void *, const void *)) comparacion_de_numeros);

     return (puntero_a_elemento != NULL);
  }

  int main (void)
  {
    if (buscar (800))
      printf ("800 está en la tabla.\n");
    else
      printf ("800 no está en la tabla.\n");

    return 0;
  }
```

---

calloc Asigna memoria principal.

Sintaxis:

```
   void *calloc (size_t nelems, size_t tam);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h.

Asigna espacio para nelems elementos de tam bytes cada uno y almacena cero en el área.

Devuelve un puntero al nuevo bloque asignado o NULL si no existe bastante espacio.

Ejemplo:

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

  int main (void)
  {
    char *str = NULL;

    /* asigna memoria para el string */
    str = calloc (10, sizeof (char));

    /* copia "Hola" en string */
    strcpy (str, "Hello");
```

```
    /* visualiza string */
    printf ("El string es %s\n", str);

    /* libera memoria */
    free (str);

    return 0;
  }
```

---

div Divide dos enteros.

Sintaxis:

```
   div_t div (int numer, int denom);
```

La función div() divide dos enteros y devuelve el cociente y el resto como un tipo div_t. Los parametros numer y denom son el numerador y el denominador respectivamente. El tipo div_t es una estructura de enteros definida (con typdef) en stdlib.h como sigue:

```
  typedef struct
    {
      long int quot; /* cociente */
      long int rem;  /* resto    */
    } div_t;
```

Ejemplo:

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

  int main (void)
  {
    div_t x = div (10, 3);
    printf ("10 div 3 = %d resto %d\n", x.quot, x.rem);
    return 0;
  }
```

---

ecvt y fcvt (TC) Convierte número en coma flotante a cadena.

Sintaxis:

```
   char *ecvt (double valor, int ndig, int *dec, int *sign);
   char *fcvt (double valor, int ndig, int *dec, int *sign);
```

Para ecvt(), ndig es el número de dígitos a almacenar, mientras que en fcvt() es el número de dígitos a almacenar depués del punto decimal. El valor devuelto apunta a un área estática que es sobreescrita en la próxima llamada.

Ejemplo:

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

  int main (void)
  {
    char *string;
```

```
    double valor;
    int dec, sign;
    int ndig = 10;

    clrscr ();
    valor = 9.876; /* número regular */
    string = ecvt (valor, ndig, &dec, &sign);
    printf ("string = %s\tdec = %d\tsign = %d\n", string, dec, sign);

    valor = -123.45; /* número negativo */
    ndig = 15;
    string = ecvt (valor, ndig, &dec, &sign);
    printf ("string = %s\tdec = %d\tsign = %d\n", string, dec, sign);

    valor = 0.6789e5; /* notación científica */
    ndig = 5;
    string = ecvt (valor, ndig, &dec, &sign);
    printf ("string = %s\tdec = %d\tsign = %d\n", string, dec, sign);

    return 0;
  }
```

---
  exit   Termina programa.
======

Sintaxis:
```
   void exit (int estado);
```

Antes de terminar, la salida buffereada es volcada, los ficheros son cerrados y las funciones exit() son llamadas.

En Turbo C, el prototipo de esta función también se encuentra en el fichero process.h.

Ejemplo:

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

  int main (void)
  {
    int estado;

    printf ("Introduce 1 ó 2\n");
    estado = getch ();
    /* Pone el errorlevel del DOS */
    exit (estado - '0');

    return 0; /* esta línea nunca es ejecutada */
  }
```

  _exit   Termina programa.
=======

Sintaxis:
```
   void _exit (int estado);
```

```
------
  fcvt    (TC) [Ver ecvt()]
======
```

```
------
  free    Libera bloques asignados con malloc() o calloc().
======

Sintaxis:
   void free (void *bloque);

Ejemplo:

  #include <string.h>
  #include <stdio.h>
  #include <alloc.h>

  int main (void)
  {
    char *str;

    /* asigna memoria para el string */
    str = malloc (10);

    /* copia "Hola" en string */
    strcpy (str, "Hola");

    /* visualiza string */
    printf ("El string es %s\n", str);

    /* libera memoria */
    free (str);

    return 0;
  }
```

```
------
  gcvt    (TC) Convierte un número en coma flotante a string.
======

Sintaxis:
   char *gcvt (double valor, int ndec, char *buf);

Devuelve la dirección del string apuntado por buf.
```

```
--------
  getenv    Obtiene un string del entorno.
========

Sintaxis:
   char *getenv (const char *nombre);

La función getenv() devuelve un puntero a la información de entorno
asociada con la cadena apuntada por nombre en la tabla de información
de entorno definida por la implementación. La cadena devuelta no debe
ser cambiada nunca por el programa.

El entorno de un programa puede incluir cosas como nombres de caminos
```

y los dispositivos que están conectados. La naturaleza exacta de estos datos viene definida por la implementación.

Si se hace una llamada a getenv() con un argumento que no se corresponde con ninguno de los datos del entorno, se devuelve un puntero nulo.

Ejemplo:

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

  int main (void)
  {
    char *s;

    s = getenv ("COMSPEC");
    printf ("Procesador de comando: %s\n",s);

    return 0;
  }
```

---

itoa Convierte un entero a una cadena.

Sintaxis:

```
   char *itoa (int valor, char *cad, int radix);
```

La función itoa() convierte el entero valor a su cadena equivalente y sitúa el resultado en la cadena apuntada por cad. La base de la cadena de salida se determina por radix, que se encuentra normalmente en el rango de 2 a 16.

La función itoa() devuelve un puntero a cad. Lo mismo se puede hacer con sprintf().

Ejemplo:

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

  int main (void)
  {
    int numero = 12345;
    char cadena[25];

    itoa (numero, cadena, 10);
    printf ("intero = %d cadena = %s\n", numero, cadena);
    return 0;
  }
```

---

labs Calcula el valor absoluto de un long.

Sintaxis:

```
   long int labs (long int x);
```

Ejemplo:

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

```
  int main (void)
  {
    long resultado;
    long x = -12345678L;

    resultado = labs (x);
    printf ("número: %ld valor absoluto: %ld\n", x, resultado);

    return 0;
  }
```

---
  ldiv   Divide dos longs, devuelve el cociente y el resto.
======

Sintaxis:

```
   ldiv_t ldiv (long int numer, long int denom);
```

La función div() divide dos longs y devuelve el cociente y el resto como un tipo ldiv_t. Los parametros numer y denom son el numerador y el denominador respectivamente. El tipo ldiv_t es una estructura de enteros definida (con typdef) en stdlib.h como sigue:

```
  typedef struct
    {
      long int quot; /* cociente */
      long int rem;  /* resto    */
    } ldiv_t;
```

Ejemplo:

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

  int main (void)
  {
    ldiv_t lx;
    lx = ldiv (100000L, 30000L);
    printf ("100000 div 30000 = %ld resto %ld\n", lx.quot, lx.rem);
    return 0;
  }
```

---
  lfind and lsearch   (TC) Ejecuta búsqueda lineal.
==========

Sintaxis:

```
  void *lfind (const void *clave, const void *base,
               size_t *num, size_t anchura,
               int (*func_de_comp) (const void *elem1, const void *elem2));

  void *lsearch (const void *clave, void *base, size_t *num, size_t anchura,
                 int (*func_de_comp) (const void *elem1, const void *elem2));
```

Estas funciones utilizan una rutina definida por el usuario (func_de_comp) para la búsqueda de la clave, en un array de elementos secuenciales.

El array tiene num elementos, cada uno de tamaño anchura bytes y comienza en la dirección de memoria apuntada por base.

Devuelve la dirección de la primera entrada en la tabla que coincida con la clave buscada. Si la clave buscada no se encuentra, lsearch la añade a

la lista; lfind devuelve 0.

La rutina *func_de_comp debe devolver cero si *elem1 == *elem2, y un valor distinto de cero en caso contrario.

Ejemplo de la función lfind:

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

  int comparar (int *x, int *y)
  {
    return (*x - *y);
  }

  int main (void)
  {
    int array[5] = { 5, -1, 100, 99, 10 };
    size_t nelem = 5;
    int clave;
    int *resultado;

    clave = 99;
    resultado = lfind (&clave, array, ≠lem, sizeof (int),
                (int (*) (const void *, const void *)) comparar);
    if (resultado)
      printf ("Número %d encontrado\n", clave);
    else
      printf ("Número %d no encontrado.\n", clave);

    return 0;
  }
```

Ejemplo de la función lsearch:

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

  int numeros[10] = { 3, 5, 1 };
  int nnumeros = 3;

  int comparar_numeros (int *num1, int *num2)
  {
    return (*num1 - *num2);
  }

  int aniadir_elemento (int numero_clave)
  {
    int viejo_nnumeros = nnumeros;

    lsearch ((void *) νmero_clave, numeros,
             (size_t *) &nnumeros, sizeof (int),
             (int (*) (const void *, const void *)) comparar_numeros);

    return (nnumeros == viejo_nnumeros);
  }

  int main (void)
  {
    register int i;
    int clave = 2;

    if (aniadir_elemento (clave))
      printf ("%d está ya en la tabla de números.\n", clave);
```

```
    else
      printf ("%d está añadido a la tabla de números.\n", clave);

    printf ("Números en tabla:\n");
    for (i = 0; i < nnumeros; i++)
      printf ("%d\n", numeros[i]);

    return 0;
  }
```

(TC)

_lrotl and _lrotr

Rota un valor long a la izquierda (_lrotl).
Rota un valor long a la derecha (_lrotr).

Sintaxis:

```
   unsigned long _lrotr (unsigned long val, int cont);
   unsigned long _lrotl (unsigned long val, int cont);
```

Las dos funciones devuelven el valor de val rotado cont bits.

Ejemplo:

```
  /* ejemplo lrotl */

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

  int main (void)
  {
    unsigned long resultado;
    unsigned long valor = 100;

    resultado = _lrotl (valor, 1);
    printf ("El valor %lu rotado un bit a la izquierda es: %lu\n",
            valor, resultado);

    return 0;
  }

  /* ejemplo lrotr */

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

  int main (void)
  {
    unsigned long resultado;
    unsigned long valor = 100;

    resultado = _lrotr (valor, 1);
    printf ("El valor %lu rotado un bit a la derecha es: %lu\n",
            valor, resultado);

    return 0;
  }
```

---

lsearch

(TC) [Ver lfind()]

```
------
  ltoa    Convierte un long a una cadena.
======

Sintaxis:
   char *ltoa (long valor, char *cadena, int radix);

Para una representación decimal, usa radix=10. Para hexadecimal, usa
radix=16.

Devuelve un puntero al argumento cadena.

Ejemplo:

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

  int main (void)
  {
    char cadena[25];
    long valor = 123456789L;

    ltoa (valor, cadena, 10);
    printf ("número = %ld  cadena = %s\n", valor, cadena);

    return 0;
  }


--------
  malloc    Asigna memoria principal.
========

Sintaxis:
   void *malloc (size_t tam);

El parámetro tam está en bytes. Devuelve un puntero al nuevo bloque
asignado, o NULL si no existe suficiente espacio para el nuevo bloque.
Si tam == 0, devuelve NULL.

Ejemplo:

  #include <stdio.h>
  #include <string.h>
  #include <alloc.h>
  #include <stdlib.h>

  int main (void)
  {
    char *str;

    if ((str = (char *) malloc (10)) == NULL)
      {
        printf ("Memoria insuficiente\n");
        exit (1);
      }

    strcpy (str, "Hola");

    printf ("El string es %s\n", str);

    free (str);

    return 0;
```

```
  }


-------------
  max  y  min    (TC) Macros que generan código en línea para encontrar el
=============   valor máximo y mínimo de dos enteros.

Sintaxis:
   max(a,b)  máximo de dos enteros a y b
   min(a,b)  mínimo de dos enteros a y b


--------
  putenv    (TC) Añade una cadena al entorno actual.
========

Sintaxis:
   int putenv (const char *nombre);

En caso de éxito, putenv() devuelve 0; en caso de fallo, devuelve -1.



-------
  qsort    Ordena usando el algoritmo quicksort (ordenación rápida).
=======

Sintaxis:

   void qsort (void *base, size_t num, size_t tam,
               int (*compara) (const void *, const void *));

La función qsort() ordena el array apuntado por base utilizando el
método de ordenación de C.A.R. Hoare (este método se ha explicado en
el ejemplo 3 de la lección 5). El número de elementos en el array se
especifica mediante num, y el tamaño en bytes de cada elemento está
descrito por tam.

La función compara se utiliza para comparar un elemento del array con
la clave. La comparación debe ser:

  int nombre_func (void *arg1, void *arg2);

Debe devolver los siguientes valores:

  Si arg1 es menor que arg2, devuelve un valor menor que 0.
  Si arg1 es igual a arg2 devuelve 0.
  Si arg1 es mayor que arg2, devuelve un valor mayor que 0.

El array es ordenado en orden ascendente con la dirección más pequeña
conteniendo el menor elemento.

Veamos un ejemplo de la utilización de esta función, donde podemos
apreciar además, dos formas posibles de declaración y utilización
de la función de comparación requerida por la función qsort().

  #include <stdio.h>  /* printf () */
  #include <stdlib.h> /* qsort () */

  void main (void)
  {
    int num[10] = { 3, 2, 8, 9, 2, 2, 1, -2, 3, 2 };
    register int i;
    int comparar_creciente (const void *elem1, const void *elem2);
```

```
    int comparar_decreciente (const int *elem1, const int *elem2);

    printf ("\nArray desordenado: ");
    for (i = 0; i < 10; i++)
      printf ("%d ", num[i]);

    qsort (num, 10, sizeof (int), comparar_creciente);

    printf ("\nArray ordenado en orden creciente: ");
    for (i = 0; i < 10; i++)
      printf ("%d ", num[i]);

    /*
       el molde del cuarto argumento convierte el tipo
         (int (*) (const int *, const int *))
       al tipo
         (int (*) (const void *, const void *))
       que es el que requiere la función qsort
    */
    qsort (num, 10, sizeof (int),
           (int (*) (const void *, const void *)) comparar_decreciente);

    printf ("\nArray ordenado en orden decreciente: ");
    for (i = 0; i < 10; i++)
      printf ("%d ", num[i]);
  }

  int comparar_creciente (const void *elem1, const void *elem2)
  {
    /* para acceder al contenido de un puntero del tipo (void *)
       necesitamos moldearlo a un tipo base que no sea void */
    return (*(int *)elem1 - *(int *)elem2);
  }

  int comparar_decreciente (const int *elem1, const int *elem2)
  {
    return (*elem2 - *elem1);
  }


------
  rand    Generador de números aleatorios.
======

Sintaxis:
   int rand (void);

Devuelve números aleatorios entre 0 y RAND_MAX. RAND_MAX está definido en
stdlib.h

Ejemplo:

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

  int main (void)
  {
    int i;
    printf ("Diez números aleatorios entre 0 y 99\n\n");
    for (i = 0; i < 10; i++)
      printf ("%d\n", rand() % 100);
    return 0;
  }
```

```
--------
  random    (TC) Macro que devuelve un entero.
========
```

```
Sintaxis:
   int random (int num);
```

Devuelve un entero entre 0 y (num-1).

Ejemplo:

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

  int main (void)
  {
    randomize ();
    printf ("Número aleatorio en el rango 0-99: %d\n", random (100));
    return 0;
  }
```

```
-----------
  randomize    (TC) Macro que inicializa el generador de números aleatorios.
===========
```

```
Sintaxis:
   void randomize (void);
```

Inicializa el generador de números aleatorios con un valor aleatorio. Esta función usa la función time(), así que debemos incluir time.h cuando usemos esta rutina.

Ejemplo:

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

  int main (void)
  {
    int i;
    randomize ();
    printf("Diez números aleatorios entre 0 y 99\n\n");
    for (i = 0; i < 10; i++)
      printf ("%d\n", rand() % 100);
    return 0;
  }
```

```
---------
  realloc    Reasigna memoria principal.
=========
```

```
Sintaxis:
   void *realloc (void *bloque, size_t tam);
```

El prototipo de esta función también se encuentra en el fichero de cabecera stdlib.h.

Intenta achicar o expandir el bloque asignado previamente a tam bytes.

Devuelve la dirección del bloque reasignado, la cual puede ser diferente
de la dirección original.

Si el bloque no puede ser reasignado o tam == 0, reallo() devuelve NULL.

Ejemplo:

```
  #include <stdio.h>
  #include <alloc.h>
  #include <string.h>

  void main (void)
  {
    char *str;

     /* asigna memoria para string */
    str = malloc (10);

    /* copia "Hola" en string */
    strcpy (str, "Hola");

    printf ("El string es %s\n  Está en la dirección %p\n", str, str);
    str = realloc (str, 20);
    printf ("El string is %s\n  Está en la nueva dirección %p\n", str, str);

    /* libera memoria */
    free (str);
  }
```

----------------- (TC)
_rotl and _rotr Rota un valor unsigned int a la izquierda (_rotl).
================ Rota un valor unsigned int a la derecha (_rotr).

Sintaxis:

```
   unsigned _rotl (unsigned val, int cont);
   unsigned _rotr (unsigned val, int cont);
```

Estas dos funciones devuelven el valor de val rotado cont bits.

Ejemplo:

```
  /* ejemplo de _rotl */

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

  int main (void)
  {
    unsigned valor, resultado;

    valor = 32767;
    resultado = _rotl (valor, 1);
    printf ("El valor %u rotado un bit a la izquierda es: %u\n",
            valor, resultado);
    return 0;
  }

  /* ejemplo de _rotr */

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

  int main (void)
```

```
  {
    unsigned valor, resultado;

    valor = 32767;
    resultado = _rotr (valor, 1);
    printf ("El valor %u rotando un bit a la derecha es: %u\n",
            valor, resultado);
    return 0;
  }
```

---
  srand   Inicializa el generador de números aleatorios.
=======

Sintaxis:
```
   void srand (unsigned semilla);
```

La función srand() utiliza semilla para fijar un punto de partida para el flujo generado por rand(), que devuelve números pseudoaleatorios.

La función srand() se utiliza normalmente para permitir que ejecuciones múltiples de un programa utilicen diferentes flujos de números pseudoaleatorios.

Ejemplo:

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

  int main (void)
  {
    int i;
    time_t t;

    srand ((unsigned) time (&t));
    printf("Diez números aleatorios entre 0 y 99\n\n");
    for (i = 0; i < 10; i++)
      printf ("%d\n", rand() % 100);
    return 0;
  }
```

---
  strtod   Convierte cadena a double.
========

Sintaxis:
```
   double strtod (const char *inic, char **fin);
```

La función strtod() convierte la representación de cadena de un número almacenado en la cadena apuntada por inic a un valor double y devuelve el resultado.

La función strtod() trabaja de la siguiente forma. Primero, se elimina cualquier carácter en blanco de la cadena apuntada por inic. A continuación cada carácter que constituye el número es leído. Cualquier carácter que no pueda ser parte de un número en coma flotante dará lugar a que el proceso se detenga. Esto incluye el espacio en blanco, signos de puntuación distintos del punto, y caracteres que no sean E o e. Finalmente fin se deja apuntando al resto, si lo hay, de la cadena original. Esto supone que si strtod() se llama con "10.10abc", se de-

vuelve el valor de 10.10 y fin apunta a la 'a' de "abc".

Si se produce un error de conversión, strtod() devuelve HUGH_VAL para desbordamiento por arriba (overflow) o HUGN_VAL para desbordamiento por abajo (underflow). Si no se produce la conversión se devuelve 0.

La cadena debe tener el siguiente formato:

```
  [sb] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
                         ===
  sb   =  espacios en blanco|
  sn   =  signo (+ o -)     |   Todos los elementos
  ddd  =  dígitos           Ç-  que están entre []
  fmt  =  e o E             |   son opcionales
   .   =  punto decimal     |
                         ===
```

Ejemplo:

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

  int main (void)
  {
    char entrada[80], *ptrfinal;
    double valor;

    printf ("Entra un número en coma flotante: ");
    gets (entrada);
    valor = strtod (entrada, &ptrfinal);
    printf ("El string es %s y el número es %lf\n", entrada, valor);
    return 0;
  }
```

---

strtol Conviete cadena a long usando la base fijada.

Sintaxis:

```
   long strtol (const char *inic, char **final, int radix);
```

La función strtol() convierte la representación en cadena de caracteres de un número (almacenada en la cadena apuntada por inic) en un número de tipo long int y devuelve el resultado. La base del número está determinada por radix. Si radix es 0, la base viene determinada por las reglas que gobiernan la especificación de constantes. Si radix es distinto de 0, debe estar en el rando de 2 a 36.

La función strtol() trabaja de la siguiente forma. Primero, elimina cualquier espacio en blanco de la cadena apuntada por inic. A continuación, se lee cada uno de los caracteres que constituyen el número. Cualquier carácter que no pueda formar parte de un número de tipo long int finaliza el proceso. Finalmente, fin se deja apuntando al resto, si lo hay, de la cadena original. Esto supone que si strtol() se llama con "100abc", se devuelve el valor 100L y fin apunta al carácter 'a' de la cadena "abc".

Si se produce un error de conversión, strtol() devuelve LONG_MAX en caso de desbordamiento por arriba o LONG_MIN en caso de desbordamiento por abajo. Si no se produce la conversión, se devuelve 0.

La cadena debe tener el siguiente formato:

```
  [sb] [sn] [0] [x] [ddd]

                          ===
  sb   =  espacios en blanco|   Todos los elementos
  sn   =  signo (+ o -)     Ç-  que están entre []
  ddd  =  dígitos           |   son opcionales
                          ===
```

Ejemplo:

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

  int main (void)
  {
    char *cadena = "87654321", *ptrfinal;
    long numero_long;

    numero_long = strtol (cadena, &ptrfinal, 10);
    printf ("cadena = %s  long = %ld\n", cadena, numero_long);

    return 0;
  }
```

---

strtoul — Convierte una cadena a un unsigned long con la base fijada.

Sintaxis:

```
   unsigned long strtoul (const char *inic, char **final, int radix);
```

La función strtoul() convierte la representación de la cadena de un número almacenada en la cadena apuntada por inic en un unsigned long int y devuelve el resultado. La base del número está determinada por radix. Si radix es 0, la base viene determinada por la regla que gobierna la especificación de constantes. Si radix está especificada, debe tener un valor en el rango de 2 a 36.

La función strtoul() trabaja de la siguiente forma. Primero, cualquier espacio en blanco en la cadena apuntada por inic es eliminado. A continuación, se lee cada carácter que constituye el número. Cualquier carácter que no pueda formar parte de un unsigned long int da lugar a que el proceso se detenga. Esto incluye espacios en blanco, signos de puntuación y caracteres. Finalmente, fin se deja apuntando al resto, si lo hay, de la cadena original. Esto supone que si strtoul() se llama con "100abc", el valor que se devuelve es 100L y fin apunta a la 'a' de "abc".

Si se produce un error de conversión, strtoul() devuelve ULONG_MAX para el desbordamiento por encima o ULONG_MIN para desbordamiento por abajo. Si la conversión no tiene lugar se devuelve 0.

Ejemplo:

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

  int main (void)
  {
    char *cadena = "87654321", *ptrfinal;
    unsigned long numero_long_unsigned;

    numero_long_unsigned = strtoul (cadena, &ptrfinal, 10);
```

```
    printf ("cadena = %s  long = %lu\n", cadena, numero_long_unsigned);

    return 0;
  }
```

---
  swab   (TC) Intercambia bytes.
======

Sintaxis:
```
   void swab (char *fuente, char *destino, int nbytes);
```

Copia nbytes bytes de fuente a destino intercambiando cada par de bytes adyacentes durante la transferencia.

```
  fuente[0] = destino[1]
  fuente[1] = destino[0]
        ...
```

nbytes debería ser un número par.

Ejemplo:

```
  /* Este programa imprime:  Este es destino: Frank Borland  */

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

  char fuente[15] = "rFna koBlrna d";
  char destino[15];

  int main (void)
  {
    swab (fuente, destino, strlen (fuente));
    printf ("Este es destino: %s\n", destino);
    return 0;
  }
```

---
  system   Ejecuta un comando DOS.
========

Sintaxis:
```
   int system (const char *comando);
```

El prototipo de esta función también se encuentra en el fichero stdlib.h.

comando puede ejecutar un comando interno del DOS tales como DIR, un fichero de programa .COM o .EXE, o un fichero batch .BAT.

Devuelve 0 en caso de éxito, -1 en caso de error y se le asigna a errno uno de los siguientes valores: ENOENT, ENOMEM, E2BIG o ENOEXEC.

La función system() también se encuentra declarada en los ficheros process.h y system.h. En el fichero system.h sólo se encuentra el prototipo de la función system().

Ejemplo:

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

```
  void main (void)
  {
    system ("dir");
  }
```

---
  ultoa   (TC) Convierte un unsigned long a una cadena.
=======

Sintaxis:
```
   char *ultoa (unsigned long valor, char *cadena, int radix);
```

Devuelve un puntero a la cadena. No devuelve error.

Ejemplo:

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

  int main (void)
  {
    unsigned long numero_unsigned_long = 3123456789L;
    char cadena[25];

    ultoa (numero_unsigned_long, cadena, 10);
    printf ("cadena = %s  unsigned long = %lu\n",
            cadena, numero_unsigned_long);

    return 0;
  }
```

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
****

---
  div_t (tipo)
==========

Tipo devuelto por la división entera.

```
  typedef struct { int quot, rem; } div_t;
```

  _doserrno (variable global)   (TC)
==========

Variable que indica el código de error del DOS actual.

```
  int _doserrno;
```

Cuando en una llamada al sistema MS-DOS ocurre un error, a _doserrno se
le asigna el código de error DOS actual.

Los mnemotécnicos para los códigos de error de DOS actual que pueden ser
asignados a _doserrno son:

Mnemotécnico | Código de error del DOS.
==========
    E2BIG | Entorno malo

```
    EACCES    | Acceso denegado
    EACCES    | Acceso malo
    EACCES    | Es directorio corriente
    EBADF     | Manejador malo
    EFAULT    | Reservado
    EINVAL    | Datos malos
    EINVAL    | Función mala
    EMFILE    | Demasiados ficheros abiertos
    ENOENT    | No es fichero ni directorio
    ENOEXEC   | Formato malo
    ENOMEM    | MCB destruido
    ENOMEM    | Fuera de memoria
    ENOMEM    | Bloque malo
    EXDEV     | Unidad mala
    EXDEV     | No es el mismo dispositivo
```

Esta variable también está declarada en los ficheros errno.h y dos.h.

---

environ (variable global) (TC)

Array de cadenas usado para acceder y alterar el entorno del proceso.

```
  extern char **environ
```

También se encuentra declarada en el fichero dos.h.

---

errno (global variable) (TC)

Variable que indica el tipo de error.

```
  int errno;
```

Siempre que ocurre un error en una llamada al sistema, a errno se le asigna un código de error que indica el tipo de error ocurrido.

En Turbo C, definido también en los ficheros errno.h y stddef.h.

EXIT_xxxx (#defines) (TC)

Constantes que definen condiciones de salida para las llamadas a la función exit().

```
  EXIT_SUCCESS   Terminación normal de un programa.
  EXIT_FAILURE   Terminación anormal de un programa.
```

_fmode (variable global) (TC)

Modo por defecto de traslación de fichero.

```
  int _fmode;
```

Al comienzo tiene el valor de O_TEXT por defecto.

Definida también en el fichero fcntl.h.

---
ldiv_t (type)

Tipo devuelto por la división de enteros largos.

```
typedef struct { long quot, rem; } ldiv_t;
```

NULL (#define)

Valor de puntero nulo.

En Turbo C, definido también en los ficheros alloc.h, mem.h, stddef.h y stdio.h.

_psp (variable global) (TC)

Dirección del segmento del PSP (Prefijo de Segmento de Programa) del programa.

```
extern unsigned int _psp;
```

También está declarada en los ficheros process.h y dos.h.

RAND_MAX (#define)

Valor máximo devuelto por la función rand().

---
size_t (tipo)

Tipo usado para los tamaños de objeto de memoria y contadores de bucles.

En Turbo C, definido también en los ficheros alloc.h, mem.h, stddef.h, stdio.h y string.h.

---
sys_errlist (variable global) (TC)

Array de cadenas de mensajes.

```
char *sys_errlist[]
```

Esta variable es un array de cadenas de mensajes que corresponden a errno y es usada por la función perror().

Los mnemotécnicos y sus significados para los valores almacenados en sys_errlist son:

Mnemotécnico	Significado
E2BIG	Lista de argumentos demasiado larga
EACCES	Permiso denegado
EBADF	Número malo de fichero
ECONTR	Bloques de memoria destruidos
ECURDIR	Intento de quitar el directorio actual
EDOM	Error de dominio
EEXIST	El fichero ya existe
EFAULT	Error desconocido
EINVACC	Código de acceso no válido
EINVAL	Argumento no válido
EINVDAT	Datos no válidos
EINVDRV	Unidad especificada no válida
EINVENV	Entorno no válido
EINVFMT	Formato no válido
EINVFNC	Número de función no válido
EINVMEM	Dirección de bloque de memoria no válido
EMFILE	Demasiados ficheros abiertos
ENMFILE	No más ficheros
ENODEV	No como dispositivo
ENOENT	No como fichero o directorio
ENOEXEC	Error de formato de exec
ENOFILE	No como fichero o directorio
ENOMEM	No hay suficiente memoria
ENOPATH	Path no encontrado
ENOTSAM	No es el mismo dispositivo
ERANGE	Resultado fuera de ranto
EXDEV	Cross-device link
EZERO	Error 0

---

sys_nerr (variable global) (TC)

Número de cademas de mensajes de error.

```
int sys_nerr;
```

Esta variable contiene el número de cadenas de mensajes de error en sys_errlist.

FUNCIONES DE HORA Y FECHA

El estándar ANSI define varias funciones que utilizan la fecha y hora del sistema al igual que el tiempo transcurrido. Estas funciones requieren la cabecera **<time.h>** en la declaración de funciones y también define algunos tipos. Los tipos **clock_t** y **time_t** permiten representar la hora y fecha del sistema como un entero extendido. El estándar ANSI se refiere a este tipo de representación como hora de calendario. El tipo de estructura **tm** mantiene la fecha y la hora separada en sus com-

ponentes. Además, time.h define la macro **CLK_TCK** que es el número de pulsos de reloj del sistema por segundo.

FICHERO DE CABECERA TIME.H

GLOSARIO:
****

---
asctime Convierte fecha y hora a ASCII.
=========
---
clock Devuelve el número de pulsos de reloj desde el comienzo del
======= programa.
---
ctime Convierte fecha y hora a una cadena.
=======
---
difftime Calcula la diferencia entre dos horas.
==========
---
gmtime Convierte fecha y hora a hora de Greenwich.
========
---
localtime Convierte fecha y hora a una estructura.
===========
---
mktime Convierte hora a formato de calendario.
========
---
stime (TC) Pone fecha y hora del sistema.
=======

---
strftime Formatea hora para salida.
==========
---
time Obtiene la hora actual.
======
---
tzset (TC) Para compatibilidad con hora de UNIX; da valores a las
======= variables globales daylight, timezone y tzname.

FUNCIONES:
****

---
asctime Convierte fecha y hora a ASCII.
=========

Sintaxis:
```
char *asctime (const struct tm *punt);
```

La función asctime() devuelve un puntero a una cadena que convierte la información almacenada en la estructura apuntada por punt de la siguiente forma:

nombre_de_dia nombre_del_mes dia_del_mes horas:minutos:segundos año\n\0

El puntero a estructura pasado a asctime() se obtiene normalmente de localtime() o gmtime().

El buffer utilizado por asctime() para mantener la cadena de salida con formato se sitúa estáticamente en un array de caracteres y se sobreescribe cada vez que se llama a la función. Si se desea salvar el contenido de la cadena, es necesario copiarlo en otro lugar.

Ejemplo:

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main (void)
{
  struct tm t;
  char str[80];

  t.tm_sec    = 1;  /* Segundos */
  t.tm_min    = 30; /* Minutos */
  t.tm_hour   = 9;  /* Hora */
  t.tm_mday   = 22; /* Día del mes */
  t.tm_mon    = 11; /* Mes */
  t.tm_year   = 56; /* Año - no incluye centenas */
  t.tm_wday   = 4;  /* Día de la semana */
  t.tm_yday   = 0;  /* No mostrado en asctime */
  t.tm_isdst  = 0;  /* Es horario de verano; no mostrado en asctime */

  strcpy (str, asctime (&t));
  printf ("%s\n", str);

  return 0;
}
```

---

clock

Devuelve el número de pulsos de reloj desde el comienzo del programa.

Sintaxis:

```
clock_t clock (void);
```

Devuelve el tiempo de procesador usado desde el comienzo de la ejecución del programa medido en pulsos de reloj. Para transformar este valor en segundos, se divide entre CLK_TCK. Se devuelve el valor -1 si el tiempo no está disponible.

Ejemplo:

```
#include <time.h>
#include <stdio.h>
#include <dos.h>

int main (void)
{
  clock_t comienzo, final;

  comienzo = clock ();

  delay (2000);

  final = clock ();
```

```
    printf ("El tiempo transcurrido ha sido: %f segundos.\n",
            (final - comienzo) / CLK_TCK);

    return 0;
  }
```

---
ctime   Convierte fecha y hora a una cadena.
=======

Sintaxis:
```
   char *ctime (const time_t *time);
```

Esta función es equivalente a:
```
   asctime (localtime (hora));
```

Ejemplo:

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

  int main (void)
  {
    time_t t;

    time (&t);
    printf ("La fecha y hora de hoy es: %s\n", ctime(&t));
    return 0;
  }
```

---
difftime   Calcula la diferencia entre dos horas.
==========

Sintaxis:
```
   double difftime (time_t hora2, time_t hora1);
```

Devuelve la diferencia, en segundos, entre hora1 y hora2. Es decir
hora2-hora1.

Ejemplo:

```
  #include <time.h>
  #include <stdio.h>
  #include <dos.h>

  int main (void)
  {
    time_t primero, segundo;

    primero = time (NULL); /* Obtiene hora del sistema */
    delay (2000); /* Espera 2 segundos */
    segundo = time (NULL); /* Obtiene otra vez la hora del sistema */

    printf ("La diferencia es: %f segundos\n", difftime (segundo, primero));

    return 0;
  }
```

---
gmtime   Convierte fecha y hora a hora de Greenwich.

========

Sintaxis:
```
   struct tm *gmtime (const time_t *hora);
```

La función gmtime() devuelve un puntero a la forma de hora en una estructura tm. La hora está representada en hora de Greenwich. El valor de hora se obtiene normalmente a través de una llamada a time().

La estructura utilizada por gmtime() mantiene la hora separada en una posición estática y se sobreescribe en ella cada vez que se llama a la función. Si se desea guardar el contenido de la estructura, es necesario copiarlo a otro lugar.

Ejemplo:

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

  int main (void)
  {
    struct tm *local, *gm;
    time_t t;

    t = time (NULL);
    local = localtime (&t);
    printf ("Hora y fecha local: %s", asctime (local));
    gm = gmtime (&t);
    printf ("Hora y fecha según Greenwich: %s", asctime (gm));

    return 0;
  }
```

---
  localtime   Convierte fecha y hora a una estructura.
==========

Sintaxis:
```
   struct tm *localtime (const time_t *hora);
```

La función localtime() devuelve un puntero a la forma esperada de hora en la estructura tm. La hora se representa en la hora local. El valor hora se obtiene normalmente a través de una llamada a time().

La estructura utilizada por localtime() para mantener la hora separada está situada de forma estática y se reescribe cada vez que se llama a la función. Si se desea guardar el contenido de la estructura, es necesario copiarla en otro lugar.

Ejemplo:

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

  int main (void)
  {
    time_t hora;
    struct tm *bloquet;

    /* obtiene hora actual */
    hora = time (NULL);

    /* convierte fecha/hora a una estructura */
```

```
    bloquet = localtime (&hora);

    printf ("La hora local es: %s", asctime (bloquet));

    return 0;
  }
```

---

mktime Convierte hora a formato de calendario.

Sintaxis:

```
   time_t mktime (struct tm *t);
```

La función mktime() devuelve la hora de calendario equivalente a la hora separada que se encuentra en la estructura apuntada por hora. Esta función se utiliza principalmente para inicializar el sistema. Los elementos tm_wday y tm_yday son activados por la función, de modo que no necesitan ser definidos en el momento de la llamada.

Si mktime() no puede representar la información como una hora de calendario válida, se devuelve -1.

Ejemplo:

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

  char *dia_semana[] = { "Domingo", "Lunes", "Martes", "Miércoles",
                         "Jueves", "Viernes", "Sábado", "Desconocido" };

  int main (void)
  {
    struct tm chequeo_de_hora;
    int anio, mes, dia;

    /* lee dia, mes y anio para encontrar el día de la semana */
    printf ("Día:   ");
    scanf ("%d", &dia);
    printf ("Mes: ");
    scanf ("%d", &mes);
    printf ("Año:  ");
    scanf ("%d", &anio);

    /* carga la estructura chequeo_de_hora con los datos */
    chequeo_de_hora.tm_year = anio - 1900;
    chequeo_de_hora.tm_mon  = mes - 1;
    chequeo_de_hora.tm_mday = dia;
    chequeo_de_hora.tm_hour = 0;
    chequeo_de_hora.tm_min  = 0;
    chequeo_de_hora.tm_sec  = 1;
    chequeo_de_hora.tm_isdst = -1;

    /* llama a mktime() para rellenear el campo weekday de la estructura */
    if (mktime (&chequeo_de_hora) == -1)
      chequeo_de_hora.tm_wday = 7;

    /* imprime el día de la semana */
    printf ("Este día es %s\n", dia_semana [chequeo_de_hora.tm_wday]);

    return 0;
  }
```

```
-------
  stime    (TC) Pone fecha y hora del sistema.
=======

Sintaxis:
   int stime (time_t *pt);

Devuelve el valor 0.

Ejemplo:

  #include <stdio.h>
  #include <time.h>

  int main (void)
  {
    time_t t;

    t = time (NULL);
    printf ("Número de segundos desde 1-1-1970: %ld\n", t);
    printf ("Hora local: %s", asctime (localtime (&t)));

    t++;
    printf ("Añadido un segundo: %s", asctime (localtime (&t)));

    t += 60;
    printf ("Añadido un minuto : %s", asctime (localtime (&t)));

    t += 3600;
    printf ("Añadido una hora  : %s", asctime (localtime (&t)));

    t += 86400L;
    printf ("Añadido un día    : %s", asctime (localtime (&t)));

    t += 2592000L;
    printf ("Añadido un mes    : %s", asctime (localtime (&t)));

    t += 31536000L;
    printf ("Añadido un año    : %s", asctime (localtime (&t)));

    return 0;
  }



----------
  strftime    Formatea hora para salida.
==========

Sintaxis:
   size_t strftime (char *cad, size_t maxtam, const char *fmt,
                    const struct tm *t);

La función strftime() sitúa la hora y la fecha (junto con otra información)
en la cadena apuntada por cad. La información se sitúa de acuerdo a las ór-
denes de formato que se encuentran en la cadena apuntada por fmt y en la
forma de hora separada t. Se sitúan un máximo de maxtam caracteres en cad.

La función strftime() trabaja de forma algo parecida a sprintf() en el que
se reconoce un conjunto de órdenes de formato que comienzan con el signo de
porcentaje (%) y sitúa su salida con formato en una cadena. Las órdenes de
formato se utilizan para especificar la forma exacta en que se representan
diferentes informaciones de hora y fecha en cad. Cualquier otro carácter
```

que se encuentre en la cadena de formato se pone en cad sin modificar. La hora y fecha presentadas están en hora local. Las órdenes de formato se presentan en la siguiente tabla. Observa que muchas órdenes distinguen entre mayúsculas y minúsculas.

La función strftime() devuelve el número de caracteres situados en la cadena apuntada por cad. Si se produce un error, la función devuelve 0.

Orden	Substituida por
%a	Día de la semana en abreviatura
%A	Día de la semana completo
%b	Abreviatura del mes
%B	Nombre del mes completo
%c	Cadena de hora y fecha estándar
%d	Día del mes en número (1-31)
%H	Hora, rango (0-23)
%I	Hora, rango (1-12)
%j	Día del año en número (1-366)
%m	Mes en número (1-12)
%M	Minuto en número (0-59)
%p	Equivalencia de lugar de AM y PM
%S	Segundos en número (0-59)
%U	Semana del año, domingo primer día (0-52)
%w	Día de la semana (0-6, domingo provoca 0)
%W	Semana del año, lunes primer día (0-52)
%x	Cadena de fecha estándar
%X	Cadena de hora estándar
%y	Año en número sin centenas (00-99)
%Y	Año completo en número
%Z	Nombre de zona temporal
%%	Signo de tanto por ciento

Ejemplo:

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

  int main(void)
  {
    struct tm *hora_act;
    time_t segs_act;
    char str[80];

    time (&segs_act);
    hora_act = localtime (&segs_act);
    strftime (str, 80, "Ahora son las %H %p del %d-%m-%y.", hora_act);
    printf ("%s\n", str);

    return 0;
  }
```

---

time — Obtiene la hora actual.

Sintaxis:

```
time_t time (time_t *hora);
```

La función time() devuelve la hora actual de calendario del sistema. Si el sistema no tiene hora, devuelve -1.

La función time() puede llamarse con un puntero nulo o con un puntero a una

variable de tipo time_t. Si se utiliza este último, el argumento es también asignado a la hora de calendario.

Ejemplo:

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

  int main (void)
  {
    time_t t;

    t = time (NULL);
    printf ("El número de segundos transcurridos desde el 1 de Enero "
            "de 1970 es %ld", t);

    return 0;
  }
```

---
tzset (TC) Para compatibilidad con hora de UNIX; da valores a las
======= variables globales daylight, timezone y tzname.

Sintaxis:
```
   void tzset (void);
```

tzset() espera encontrar una cadena de entorno de la forma

```
  TZ=...
```

que define la zona horaria.

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

  int main (void)
  {
    time_t td;

    /* Pacific Standard Time & Daylight Savings */
    putenv ("TZ=PST8PDT");
    tzset ();
    time (&td);
    printf ("Hora actual = %s\n", asctime (localtime (&td)));

    return 0;
  }
```

CONSTANTES, TIPOS DE DATOS, Y VARIABLES GLOBALES:
****

---
clock_t (tipo)
==========

Este tipo de datos devuelto por la función clock() almacena un tiempo transcurrido medido en pulsos de reloj.

La constante CLK_TCK define el número de pulsos de reloj por segundo.

daylight (global variable)

Indica si se harán ajustes de horario de verano.

```
int daylight;
```

Las funciones de hora y fecha usan dylight.

time_t (tipo)

Este tipo de variable define el valor usado por las funciones de tiempo.

En Turbo C, también está declarado en sys\types.h. El fichero sys\types.h únicamente contiene la declaración de este tipo.

timezone (variable global)

Diferencia en segundos entre hora local y hora del meridiano de Greenwich.

```
extern long timezone;
```

Usada por las funciones de hora y fecha.

TM (struct)

Describe una estructura que contiene una fecha y hora separada en sus componentes.

```
struct tm
  {
    int tm_sec;   /* segundos, 0-59                    */
    int tm_min;   /* minutos, 0-59                     */
    int tm_hour;  /* horas, 0-23                       */
    int tm_mday;  /* día del mes, 1-31                 */
    int tm_mon;   /* mes, 0-11, Enero = 0              */
    int tm_year;  /* año, año actual menos 1900        */
    int tm_wday;  /* día de la semana, 0-6, Domingo = 0 */
    int tm_yday;  /* día del año, 0-365, 1 de Enero = 0 */
    int tm_isdst; /* indicador de horario de verano    */
  };
```

tzname (variable global) (TC)

Array de punteros a cadenas.

```
extern char * tzname[2]
```

NOTA: esta variable está declarada en el fichero io.h, no en éste (time.h).

Un array de punteros a cadenas conteniendo las abreviaciones para los nombres de zonas horarias.

FUNCIONES RELACIONADAS CON INFORMACION GEOGRAFICA

El fichero de cabecera **<locale.h>** declara funciones que proporcionan información específica de los lenguajes y países.

FICHERO DE CABECERA LOCALE.H

---
localeconv (TC) Devuelve un puntero a la estructura de lugar actual.
==========

Sintaxis:
```
struct lconv *localeconv (void);
```

localeconv pone la moneda específica del país y otros formatos numéricos. Sin embargo, Borland C++ actualmente sólo soporta el lugar C.

---
setlocale Selecciona el lugar.
==========

Sintaxis:
```
char *setlocale (int categoria, char *lugar);
```

La función setlocale() permite al usuario pedir o activar ciertos parámetros que son sensibles al lugar donde se utiliza el programa. Por ejemplo, en Europa, la coma se utiliza en lugar del punto decimal; del mismo modo los formatos de la hora y la fecha difieren.

Si lugar es nulo, setlocale() devuelve un puntero a la actual cadena de localización. En cualquier otro caso, setlocale() intenta utilizar la cadena de localización especificada para poner los parámetros de lugar según se especifica en categoria.

En el momento de la llamada, categoria debe de ser una de las siguientes macros:

- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME

LC_ALL hace referencia a todas las categorías de localización. LC_COLLATE afecta a la operación de la función strcoll(). LC_CTYPE modifica la forma de trabajo de las funciones de caracteres. LC_NUMERIC cambia el carácter del punto decimal para las funciones de entrada/salida con formato. Finalmente, LC_TIME determina el comportamiento de la función strftime().

El estándar define dos posibles cadenas para lugar. La primera es "C", que especifica el mínimo entorno para la compilación de C. El segundo es " ", la cadena vacía, que especifica el entorno de implementación definido por defecto. El resto de los valores para locale() están definidos por la implementación y afectan a la portabilidad.

En Borland C++ (compilador con el que se ha desarrollado este programa), el único lugar soportado es "C".

Ejemplo:

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

int main (void)
{
  printf ("Lugar activo actualmente: %s", setlocale (LC_ALL, "C"));

  return 0;
}
```

LCONV (struct)
==========

```
struct lconv
  {
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
  };
```

partir

FUNCION Y ESTRUCTURA DE HORA ACTUAL

El fichero de cabecera **<sys\timeb.h>** de Turbo C declara la función **ftime()** y la estructura **timeb** para obtener la hora actual.

FICHERO DE CABECERA SYS\TIMEB.H (TC)

```
-------
  ftime    Almacena hora actual en la estructura timeb.
=======

Sintaxis:
   void ftime (struct timeb *buf);


----------------
  TIMEB (struct)
================

Información de hora actual rellenada por la función ftime().

  struct timeb
    {
      long time ; /* segundos desde 1-1-70 hora Greenwitch */
      short millitm ; /* fracción de segundo (en milisegundos) */
      short timezone ; /* diferencia entre hora local y hora Greenwitch */
      short dstflag ; /* 0 si no es horario de verano */
    } ;
```

FUNCIONES DE INFORMACION DE FICHEROS

El fichero de cabecera **<sys\stat.h>** de Turbo C declara dos funciones que obtienen información de un fichero abierto: **fstat()** y **stat()**. Además define la estructura **stat** que contiene la información sobre los ficheros y también define una serie de constantes para identificar el estado de los ficheros.

FICHERO DE CABECERA SYS\STAT.H (TC)

```
-------
  fstat    Obtiene información del fichero abierto.
=======

Sintaxis:
   int fstat (int descriptor, struct stat *statbuf);

Devuelve 0 si tiene éxito; o -1 en caso de error y se pone en errno el
número de error.

  #include <sys\stat.h>
  #include <stdio.h>
  #include <time.h>

  int main (void)
  {
    struct stat statbuf;
    FILE *stream;

    if ((stream = fopen ("DUMMY.FIL", "w+")) == NULL)
```

```
      {
        fprintf (stderr, "No se puede abrir el fichero.\n");
        return (1);
      }
    fprintf (stream, "Esto es una prueba.");
    fflush (stream);

    fstat (fileno (stream), &statbuf);
    fclose (stream);

    if (statbuf.st_mode & S_IFCHR)
      printf ("El descriptor se refiere a un dispositivo.\n");
    if (statbuf.st_mode & S_IFREG)
      printf ("El descriptor se refiere a un fichero ordinario.\n");
    if (statbuf.st_mode & S_IREAD)
      printf ("El usuario tiene permiso de lectura sobre el fichero.\n");
    if (statbuf.st_mode & S_IWRITE)
      printf ("El usuario tiene permiso de escritura sobre el fichero.\n");

    printf ("Letra de la unidad del fichero: %c\n", 'A'+statbuf.st_dev);
    printf ("Tamaño del fichero en bytes: %ld\n", statbuf.st_size);
    printf ("Hora de la última apertura del fichero: %s\n",
            ctime (&statbuf.st_ctime));

    return 0;
  }


------
  stat    Obtiene información acerca del fichero abierto.
======

Sintaxis:
   int stat (char *path, struct stat *statbuf);

Devuelve 0 si tiene éxito; o -1 en caso de error y se pone en errno el
número de error.

Ejemplo:

  #include <sys\stat.h>
  #include <stdio.h>
  #include <time.h>

  #define NOMBREFICHERO "TEST.$$$"

  int main (void)
  {
    struct stat statbuf;
    FILE *stream;

    if ((stream = fopen (NOMBREFICHERO, "w+")) == NULL)
      {
        fprintf (stderr, "No se puede abrir el fichero.\n");
        return (1);
      }

    stat (NOMBREFICHERO, &statbuf);

    fclose (stream);

    if (statbuf.st_mode & S_IFCHR)
      printf ("El descriptor se refiere a un dispositivo.\n");
    if (statbuf.st_mode & S_IFREG)
```

```
      printf ("El descriptor se refiere a un fichero ordinario.\n");
    if (statbuf.st_mode & S_IREAD)
      printf ("El usuario tiene permiso de lectura sobre el fichero.\n");
    if (statbuf.st_mode & S_IWRITE)
      printf ("El usuario tiene permiso de escritura sobre el fichero.\n");

    printf ("Letra de la unidad del fichero: %c\n", 'A'+statbuf.st_dev);
    printf ("Tamaño del fichero en bytes: %ld\n", statbuf.st_size);
    printf ("Hora de la última apertura del fichero: %s\n",
            ctime (&statbuf.st_ctime));

    return 0;
  }
```

S_Ixxxx (#defines)
==========

Definiciones usadas por las funciones de directorio y estado de fichero.

S_IFMT	Máscara de tipo de fichero
S_IFDIR	Directorio
S_IFIFO	FIFO especial
S_IFCHR	Carácter especial
S_IFBLK	Bloque especial
S_IFREG	Fichero regular
S_IREAD	Poseedor puede leer
S_IWRITE	Poseedor puede escribir
S_IEXEC	Poseedor puede ejecutar

STAT (struct)
==========

Describe una estructura que contiene información acerca de un fichero o directorio.

```
  struct stat
    {
      short  st_dev,   st_ino;
      short  st_mode,  st_nlink;
      int    st_uid,   st_gid;
      short  st_rdev;
      long   st_size,  st_atime;
      long   st_mtime, st_ctime;
    };
```

CONSTANTES SIMBOLICAS PARA COMPATIBILIDAD CON UNIX

El fichero de cabecera **<values.h>** de Turbo C define constantes compatibles con UNIX para valores límite de los tipos float y double.

FICHERO DE CABECERA VALUES.H (TC)

```
----------------------
  BITSPERBYTE (#define)
======================

Número de bits en un byte.


---------------------------------
  Límites float y double (#defines)
=================================

UNIX System V compatible:
=============================================
_LENBASE   | Base a la que aplicar exponente


Límites para valores float y double
=================================================
_DEXPLEN   | Número de bits en exponente
DMAXEXP    | Máximo exponente permitido
DMAXPOWTWO | Potencia de dos más grande permitida
DMINEXP    | Mínimo exponente permitido
DSIGNIF    | Número de bits significativos
MAXDOUBLE  | Valor double más grande
MINDOUBLE  | Valor double más pequeño


Límites para valores float
================================================
_FEXPLEN   | Número de bits en exponente
FMAXEXP    | Máximo exponente permitido
FMAXPOWTWO | Potencia de dos más grande permitida
FMINEXP    | Mínimo exponente permitido
FSIGNIF    | Número de bits significativos
MAXFLOAT   | Valor float más grande
MINFLOAT   | Valor float más pequeño
```

FUNCIONES DE COMA FLOTANTE

En el fichero de cabecera **<float.h>** de Turbo C están definidas una serie de funciones relacionadas con el coprocesador matemático o con el emulador en su defecto, esto es, con las operaciones de coma flotante.

FICHERO DE CABECERA FLOAT.H (TC)

```
----------
  _clear87    Borra el estado de coma flotante.
==========

Sintaxis:
   unsigned int _clear87 (void);

Los bits del valor devuelto indican el viejo estado de coma flotante.
Para obtener información acerca de los estados, ver las constantes definidas
en el fichero de cabecera float.h.
```

Ejemplo:

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

  int main (void)
  {
    float x;
    double y = 1.5e-100;

    printf ("\nEstado antes del error: %X\n", _status87 ());

    x = y; /* crea desbordamiento por abajo y pérdida de precisión */
    printf ("Estado después del error: %X\n", _status87 ());

    _clear87 ();
    printf ("Estado después de borrar el error: %X\n", _status87 ());

    y = x;

    return 0;
  }
```

_fpreset Reinicializa el paquete matemático de coma flotante.

Sintaxis:

```
   void  _fpreset (void);
```

Esta función se suele usar junto con las funciones de systema exec y spawn puesto que un proceso hijo podría alterar el estado de coma flotante del proceso padre.

Ejemplo:

```
  #include <stdio.h>
  #include <float.h>
  #include <setjmp.h>
  #include <signal.h>
  #include <process.h>
  #include <conio.h>

  jmp_buf reentrada;

  /* define un manejador para atrapar errores de coma flotante */
  void trampa_matematica (int sig)
  {
    printf ("Atrapando error de coma flotante.\n");
    printf ("La señal es: %d\n", sig);
    printf ("Presiona una tecla para continuar.\n");
    getch ();

    /* reinicializa el chip 8087 o emulador para borrar cualquier error */
    _fpreset ();

    /* vuelve al lugar del problema */
    longjmp (reentrada, -1);
  }

  int main (void)
  {
    float uno = 3.14, dos = 0.0;
```

```
    /* instala manejador de señal para las excepciones de coma flotante */
    if (signal (SIGFPE, trampa_matematica) == SIG_ERR)
      {
        printf ("Error al instalar manejador de señal.\n");
        exit (3);
      }

    printf ("Presiona una tecla para generar un error matemático.\n");
    getch ();

    if (setjmp (reentrada) == 0)
       uno /= dos;

    printf ("Volviendo del menajeador de la señal.\n");
    return 0;
  }
```

_control87 Cambia palabra de control de coma flotante.
==========

Sintaxis:

```
   unsigned int _control87 (unsigned int nuevo, unsigned int mascara);
```

Si un bit en la máscara es 1, el correspondiente bit en nuevo contiene el nuevo valor para el mismo bit en la palabra de control. Si mascara es 0, la palabra de control no es alterada.

_status87 Obtiene el estado de coma flotante.
==========

Sintaxis:

```
   unsigned int _status87 (void);
```

Los bits del valor devuelto contienen el estado de coma flotante.

Ejemplo:

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

  int main (void)
  {
     float x;
     double y = 1.5e-100;

     printf ("Estado del 87 antes del error: %x\n", _status87 ());

     x = y;  /* <-- fuerza a que se produzca un error */
     y = x;

     printf ("Estado del 87 después del error: %x\n", _status87 ());

     return 0;
  }
```

CW_DEFAULT (#define)
==========

Palabra de control por defecto para el comprocesador matemático 8087/80287.

CONEXION DE TURBO C CON ENSAMBLADOR

El C es un lenguaje muy potente, pero hay veces que puede interesar escribir algunas intrucciones en ensamblador. Hay tres razones para ello:

- Aumentar la velocidad y la eficiencia.
- Realizar una función específica de la máquina que no está disponible en C.
- Utilizar una rutina en lenguaje ensamblador empaquetada de propósito general.

Aquí no nos vamos a extender mucho en este tema, sólo nos vamos a limitar a comentar cómo se pueden incluir instrucciones ensamblador en un programa de C en el caso de Turbo C.

Antes de escribir instrucciones en ensamblador debemos incluir la siguiente directiva:

#pragma inline

la cual le dice al compilador que el programa contiene estamentos asm.

La palabra clave asm tiene la siguiente sintaxis:

asm codigo_de_operacion operandos punto_y_coma_o_nueva_linea

Ejemplo:

```
int var = 10;
asm mov ax, var
```

Si se quiere incluir varias instrucciones en ensamblador, se crea un bloque (con dos llaves) después de asm.

Ejemplo:

```
asm
  {
    pop ax; pop ds
    iret
  }
```

Para profundizar más en el tema consulta el manual de usuario de tu compilador de C.

Ejemplo:

```
/*
   Este ejemplo instala una rutina manejadora para la señal SIGFPE,
   atrapa una condición de desbordamiento de entero, hace un reajuste
   del registro AX y vuelve.
*/

#pragma inline

#include <stdio.h>
#include <signal.h>

#pragma argsused /* para evitar aviso de argumentos sig y tipo no usados */
```

```
void manejador (int sig, int tipo, int *listreg)
{
   printf ("\nInterrupción atrapada.");
   *(listreg + 8) = 3; /* hace devolver AX = 3 */
}

void main (void)
{
   signal (SIGFPE, manejador);

   asm mov ax, 07FFFH /* AX = 32767 */
   asm inc ax /* causa desbordamiento (overflow) */
   asm into /* activa manejador */

   /* El manejador pone AX a 3 a la vuelta. Si no se hubiera incluido en
      el código la línea anterior (instrucción del ensamblador into), se
      activaría el manejador en la instrucción into que han después del
      decremento */

   asm dec ax /* ahora no se produce desbordamiento (overflow) */
   asm into /* ahora no se activa el manejador */
}
```

Otra forma de hacer el ejemplo anterior:

```
/*
   Este ejemplo instala una rutina manejadora para la señal SIGFPE,
   atrapa una condición de desbordamiento de entero, hace un reajuste
   del registro AX y vuelve.
*/

#include <dos.h> /* para utilizar la pseudovariable _AX */

#include <stdio.h>
#include <signal.h>

#pragma argsused /* para evitar aviso de argumentos sig y tipo no usados */

void manejador (int sig, int tipo, int *listreg)
{
   printf ("\nInterrupción atrapada.");
   *(listreg + 8) = 3; /* hace devolver AX = 3 */
}

void main (void)
{
   signal (SIGFPE, manejador);

   _AX = 0x07FFF; /* AX = 32767 */
   _AX++; /* causa desbordamiento (overflow) */
   geninterrupt (4); /* activa manejador */

   _AX--;
   geninterrupt (4);

   /* El manejador pone AX a 3 a la vuelta. Si no se hubiera incluido en
      el código la línea anterior (instrucción del ensamblador into), se
      activaría el manejador en la instrucción into que han después del
      decremento */

   _AX--;
   geninterrupt (4);
}
```

Las pseudovariables registros en Turbo C son las siguientes:

```
  _AX    _AL    _AH    _SI    _ES
  _BX    _BL    _BH    _DI    _SS
  _CX    _CL    _CH    _BP    _CS
  _DX    _DL    _DH    _SP    _DS
  _FLAGS
```

Estas pseudovariables registros corresponden a los registros de los procesadores 80x86.

```
    f f f f    i i i i    n     n    a a a a    l
    f             i       n n   n    a     a    l
    f f f         i       n  n  n    a a a a    l
    f             i       n   n n    a     a    l
    f          i i i i    n     n    a     a    l l l l
```

```
  D E    L A    T E O R I A    D E L    L E N G U A J E
```