Introducción al Lenguaje C

Armando Serrano

Contenido

1
Introducción

Breve historia

El Lenguaje C fue creado en 1972 por Dennis Ritchie en un PDP-11 de Digital Equipment Corporation bajo el sistema operativo UNIX. Fue el resultado final de un proyecto que comenzó con un lenguaje llamado BCPL (Basic Combined Programming Language) diseñado por Martin Richards en 1967, que a su vez estaba influenciado por el lenguaje CPL (Combined Programming Language) desarrollado por las universidades de Cambridge y Londres. A partir del BCPL, Ken Thompson creó un lenguaje llamado B, que fue el que condujo al desarrollo del Lenguaje C.

Durante muchos años el estándar para C fue el que se suministraba con la versión 5 de UNIX. Pero con la creciente popularidad de los microordenadores aparecieron muchas implementaciones diferentes (Quick C de Microsoft, Turbo C de Borland, etc.) que, aunque eran altamente compatibles entre sí, tenían algunas diferencias. Por ello, en 1983 se creó un comité que elaboró el documento que define el estándar ANSI de C.

El Lenguaje C es un lenguaje de nivel medio, es decir, sin ser un lenguaje de alto nivel como COBOL, BASIC o Pascal, tampoco es un Lenguaje Ensamblador.

Las principales características del Lenguaje C son:

- Tiene un conjunto completo de instrucciones de control.
- Permite la agrupación de instrucciones.
- Incluye el concepto de puntero (variable que contiene la dirección de otra variable).
- Los argumentos de las funciones se transfieren por su valor. Por ello, cualquier cambio en el valor de un parámetro dentro de una función no afecta al valor de la variable fuera de ella.
- La E/S no forma parte del lenguaje, sino que se proporciona a través de una biblioteca de funciones.
- Permite la separación de un programa en módulos que admiten compilación independiente.

Originalmente el Lenguaje C estuvo muy ligado al sistema operativo UNIX que, en su mayor parte, está escrito en C. Más adelante se comenzó a utilizar en otros sistemas operativos para programar editores, compiladores, etc. Aunque se le conoce como un lenguaje de programación de sistemas, no se adapta mal al resto de aplicaciones. De hecho, hoy en día un alto porcentaje de software para ordenadores personales está escrito en Lenguaje C. Por ejemplo, el sistema operativo MS-DOS.

En este capítulo realizaremos un rápido recorrido por algunas de las características del lenguaje a través de unos ejemplos muy sencillos. En los siguientes capítulos estudiaremos con mucho más detalle la mayor parte de los aspectos del Lenguaje C. Este estudio lo basaremos en la implementación de Borland: el Turbo C. Estos programas pueden ejecutarse desde el entorno integrado de Turbo C o compilándolos y enlazándolos desde la línea de órdenes del DOS (Capítulo 12).

Ejemplo 1: #include, main(), printf()

Comenzaremos por un ejemplo sencillo: un programa que muestra en pantalla una frase.

```
/* Ejemplo 1. Programa DOCENA.C */

#include <stdio.h>

main ()
{
    int docena;

    docena = 12;
    printf ("Una docena son %d unidades\n", docena);
}
```

Este programa hace aparecer en pantalla la frase "Una docena son 12 unidades". Veamos el significado de cada una de las líneas del programa.

/* Ejemplo 1. Programa DOCENA.C */

Es un comentario. El compilador de Turbo C ignora todo lo que está entre los símbolos de comienzo (**/***) y fin (***/**) de un comentario. Los comentarios delimitados por estos símbolos pueden ocupar varias líneas.

#include <stdio.h>

Le dice a Turbo C que en el proceso de compilación incluya un archivo denominado **stdio.h**. Este fichero se suministra como parte del compilador de Turbo C y contiene la información necesaria para el correcto funcionamiento de la E/S de datos.

La sentencia **#include** no es una instrucción C. El símbolo # la identifica como una directiva, es decir, una orden para el preprocesador de C, responsable de realizar ciertas tareas previas a la compilación.

Los archivo ***.h** se denominan **archivos de cabecera**. Todos los programas C requieren la inclusión de uno o varios archivos de este tipo, por lo que normalmente es necesario utilizar varias líneas **#include**.

main ()

Es el nombre de una función. Un programa C se compone de una o más funciones, pero al menos una de ellas debe llamarse **main()**, pues los programas C empiezan a ejecutarse por esta función.

Los paréntesis identifican a **main()** como una función. Generalmente, dentro de ellos se incluye información que se envía a la función. En este caso no hay traspaso de información por lo que no hay nadaescrito en su interior. Aún así son obligatorios.

El **cuerpo de una función** (conjunto de sentencias que la componen) va enmarcado entre llaves **{** y **}**. Ese es el significado de las llaves que aparecen en el ejemplo.

int docena;

Es una sentencia declarativa. Indica que se va a utilizar una variable llamada **docena** que es de tipo entero. La palabra **int** es una palabra clave de C que identifica uno de los tipos básicos de datos que estudiaremos en el Capítulo 3. En C es obligatorio declarar todas las variables antes de ser utilizadas. El "**;**" identifica la línea como una sentencia C.

docena = 12;

Es una sentencia de asignación. Almacena el valor **12** a la variable **docena**. Obsérvese que acaba con punto y coma. Como en la mayoría de los lenguajes, el operador de asignación en C es el signo igual "=".

printf ("Una docena son %d unidades\n", docena);

Esta sentencia es importante por dos razones: en primer lugar, es un ejemplo de llamada a una función. Además ilustra el uso de una función estándar de salida: la función **printf()**.

La sentencia consta de dos partes:

- El nombre de la función: **printf()**.
- Los argumentos. En este caso hay dos separados por una coma:
  - **"Una docena son %d unidades\n"**
  - **docena**

Como toda sentencia C acaba con punto y coma.

La función **printf()** funciona de la siguiente forma: el primer argumento es una **cadena de formato**. Esta cadena será lo que, básicamente, se mostrará en pantalla. En la cadena de formato pueden aparecer **códigos de formato** y **caracteres de escape**.

Un **código de formato** comienza por el símbolo **%** e indica la posición dentro de la cadena en donde se imprimirá el segundo argumento, en este caso, la variable **docena**. Más adelante estudiaremos todos los códigos de formato de Turbo C. En este ejemplo, **%d** indica que en su lugar se visualizará un número entero decimal.

Un **carácter de escape** comienza por el símbolo \. Son caracteres que tienen una interpretación especial. La secuencia **\n** es el carácter **nueva línea** y equivale a la secuencia LF+CR (salto de línea + retorno de cursor).

La función **printf()** pertenece a la biblioteca estándar de C. Las definiciones necesarias para que funcione correctamente se encuentran en el archivo **stdio.h**, de ahí que sea necesaria la sentencia **#include <stdio.h>**.

Ejemplo 2: scanf()

El siguiente programa realiza la conversión de pies a metros usando la equivalencia:

1 pie = 0.3084 metros

El programa solicita por teclado el número de pies y visualiza en pantalla los metros correspondientes.

```
/* Ejemplo 2. Programa PIES.C */
```

```
#include <stdio.h>

main ()
{
    int pies;
    float metros;

    printf ("\n¿Pies?: ");
    scanf ("%d", &pies);

    metros = pies * 0.3084;

    printf ("\n%d pies equivalen a %f metros\n", pies, metros);
}
```

Estudiaremos ahora las novedades que aparecen en este programa.

float metros;

Es una sentencia declarativa que indica que se va a utilizar una variable llamada **metros**, que es del tipo **float**. Este tipo de dato se utiliza para declarar variables numéricas que pueden tener decimales.

printf ("\n¿Pies?: ");

Es la función **printf()** comentada antes. En esta ocasión sólo tiene un argumento: la cadena de control sin códigos de formato. Esta sentencia simplemente sitúa el cursor al principio de la siguiente línea (**\n**) y visualiza la cadena tal como aparece en el argumento.

scanf ("%d", &pies);

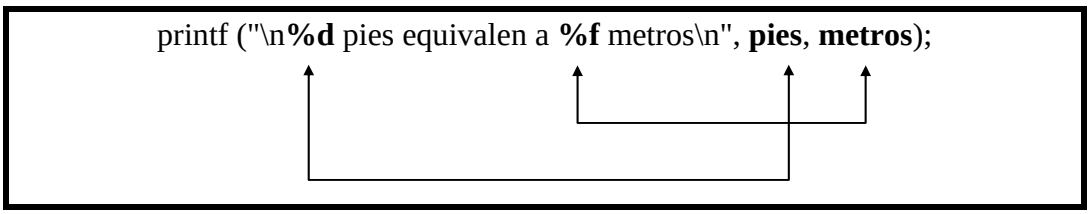
scanf() es una función de la biblioteca estándar de C (como **printf()**), que permite leer datos del teclado y almacenarlos en una variable. En el ejemplo, el primer argumento, **%d**, le dice a **scanf()** que tome del teclado un número entero. El segundo argumento, **&pies**, indica en qué variable se almacenará el dato leído. El símbolo **&** antes del nombre de la variable es necesario para que **scanf()** funcione correctamente. Aclararemos este detalle en capítulos posteriores.

metros = pies * 0.3084;

Se almacena en la variable **metros** el resultado de multiplicar la variable **pies** por **0.3084**. El símbolo ***** es el operador que usa C para la multiplicación.

printf ("\n%d pies equivalen a %f metros\n", pies, metros);

Aquí **printf()** tiene 3 argumentos. El primero es la cadena de control, con dos códigos de formato: **%d** y **%f**. Esto implica que **printf()** necesita dos argumentos adicionales. Estos argumentos encajan en orden, de izquierda a derecha, con los códigos de formato. Se usa **%d** para la variable **pies** y **%f** para la variable **metros**.



El código **%f** se usa para representar variables del tipo **float**.

Ejemplo 3: Funciones con argumentos

Veremos ahora dos ejemplos de programas que utilizan funciones creadas por el programador. Una función es una subrutina que contiene una o más sentencias C. Viene definida por un nombre, seguida de dos paréntesis **()** entre los que puede haber o no argumentos. Los argumentos son valores que se le pasan a la función cuando se llama.

Veamos, en primer lugar, un ejemplo de una función sin argumentos.

```
/* Ejemplo 3.1 - Programa FUNCION1.C */

#include <stdio.h>

main ()
{
    printf ("\nEste mensaje lo muestra la función main()");
    MiFuncion ();
}

/* Definición de la función MiFuncion() */
MiFuncion ()
{
    printf ("\nEste otro lo muestra MiFuncion()");
}
```

En este ejemplo se utiliza la función **MiFuncion()** para mostrar en pantalla una frase. Como se ve, **MiFuncion()** se invoca igual que **printf()** o **scanf()**, es decir, simplemente se escribe el nombre de la función y los paréntesis. La

definición de **MiFuncion()** tiene el mismo aspecto que **main()**: el nombre de la función con los paréntesis y, seguidamente, el cuerpo de la función encerrado entre llaves.

El siguiente ejemplo ilustra el uso de una función con argumentos. El programa visualiza el cuadrado de un número entero por medio de una función que recibe dicho número como argumento.

```
/* Ejemplo 3.2 - Programa FUNCION2.C */

#include <stdio.h>

main ()
{
    int num;

    printf ("\nTeclee un número entero: ");
    scanf ("%d", &num);
    cuadrado (num);
}

/* Definición de la función cuadrado() */
cuadrado (int x)
{
    printf ("\nEl cuadrado de %d es %d\n", x, x * x);
}
```

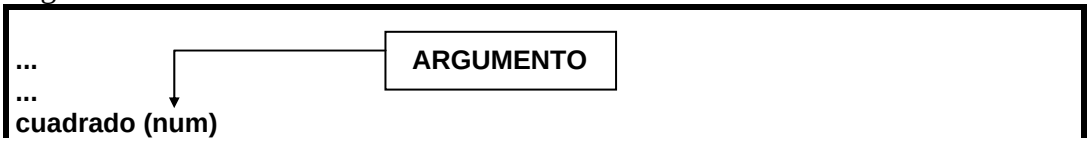
cuadrado (int x)

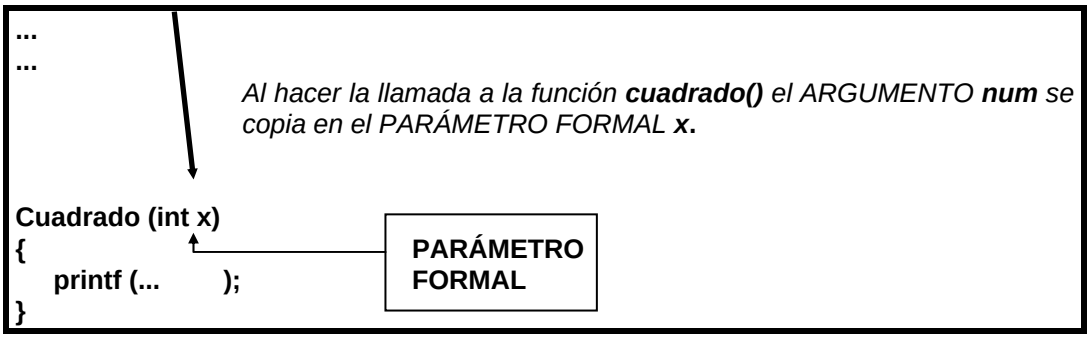
Es la declaración de la función **cuadrado()**. Dentro de los paréntesis se pone la variable que recibirá el valor pasado a **cuadrado()** y de qué tipo es. Así, si se teclea el valor **6**, se almacena en **num** y al hacer la llamada **cuadrado (num),** la variable **num** se copia en la variable **x**, que es con la que trabaja internamente la función **cuadrado()**.

Es importante mantener claros dos términos:

1. El término **ARGUMENTO** se refiere a la variable usada al llamar la función.
2. El término **PARÁMETRO FORMAL** se refiere a la variable de una función que recibe el valor de los argumentos.

También es importante tener claro que la copia de variables se hace sólo en una dirección: del argumento al parámetro formal. Cualquier modificación del parámetro formal realizado dentro de la función no tiene ninguna influencia en el argumento.





Otro detalle a tener en cuenta es que el tipo de argumento que se utiliza para llamar a una función debe ser el mismo que el del parámetro formal que recibe el valor. Así, no debe llamarse a la función cuadrado con un argumento de tipo **float** (más adelante veremos que C permite cierta flexibilidad en este aspecto).

Ejemplo 4: Funciones que devuelven valores

Para finalizar el capítulo veremos un ejemplo que utiliza una función que devuelve un valor. El siguiente programa lee dos números enteros del teclado y muestra su producto en pantalla. Para el cálculo se usa una función que recibe los dos números y devuelve el producto de ambos.

```
/* Ejemplo 4 - Programa MULT.C */

#include <stdio.h>

main ()
{
    int a, b, producto;

    printf ("\nTeclee dos números enteros: ");
    scanf ("%d %d", &a, &b);

    producto = multiplica (a, b);

    printf ("\nEl resultado es %d", producto);
}

/* Definición de la función multiplica() */
multiplica (int x, int y)
{
    return (x * y);
}
```

Las novedades que se presentan en este programa se comentan a continuación.

scanf ("%d %d", &a, &b);

La cadena de control de **scanf()** contiene dos códigos de formato. Al igual que ocurre en **printf()**, se precisan dos argumentos más, uno por cada código de formato. Los dos números se teclean separados por espacios en blanco, tabuladores o por la tecla Intro.

return (x * y);

La palabra clave **return** se usa dentro de las funciones para salir de ellas devolviendo un valor. El valor devuelto mediante **return** es el que asume la función. Eso permite tener sentencias como

producto = multiplica (a, b);

es decir, sentencias en las que la función está a la derecha del operador de asignación. Para nuestros propósitos actuales podemos decir (aunque esto no sea exacto) que después de la sentencia **return** la función **multiplica()** actúa como si fuese una variable que almacena el valor devuelto.

Los paréntesis son opcionales, se incluyen únicamente para clarificar la expresión que acompaña a **return**. No deben confundirse con los paréntesis de las funciones.

Ejercicios

1. Encuentra todos los errores del siguiente programa C:

```
include studio.h

/* Programa que dice cuántos días hay en una semana /*

main {}
(
    int d

    d := 7;
    print (Hay d días en una semana);
```

2. Indica cuál sería la salida de cada uno de los siguientes grupos de sentencias:

a) **printf ("Historias de cronopios y famas.");**
printf ("Autor: Julio Cortázar");

b) **printf ("¿Cuántas líneas \nocupa esto?");**

c) **printf ("Estamos \naprendiendo /naprogramar en C");**

d)
```
int num;
num = 2;
printf ("%d + %d = %d", num, num, num + num);
```

3. Escribe un programa que calcule el área de un círculo de radio R y la longitud de su circunferencia. Solicitar el valor de R por teclado, mostrando en la pantalla los mensajes necesarios ($S = \pi \cdot R^2$; $L = 2 \cdot \pi \cdot R$).

4. Sean dos cuadrados de lados L1 y L2 inscritos uno en otro. Calcula el área de la zona comprendida entre ambos, utilizando para ello una función (que se llamará **AreaCuadrado**) que devuelve el área de un cuadrado cuyo lado se pasa como argumento.

2
Elementos de un programa C

Introducción

Básicamente el C está compuesto por los siguientes elementos

- Constantes
- Identificadores
- Palabras reservadas
- Comentarios
- Operadores

Para representar estos elementos se utilizan los caracteres habituales (letras, números, signos de puntuación, subrayados, ...) aunque no todos los elementos pueden usar todos estos caracteres.

Una característica importante del Lenguaje C es que en todos los elementos anteriormente enumerados distingue letras mayúsculas y minúsculas. Así, **int** es una palabra reservada del lenguaje que sirve para declarar variables enteras, mientras que **Int** podría ser el nombre de una variable.

Constantes

Las constantes que se pueden usar en C se clasifican de la siguiente forma:

- Enteras
- Reales
- De carácter

Constantes enteras

Son números sin parte fraccionaria. Pueden expresarse en decimal, octal o hexadecimal.

Una constante octal debe comenzar con un cero:

$$016 \rightarrow 16 \text{ octal} = 14 \text{ decimal}$$

Una constante hexadecimal debe comenzar con un cero seguida de **x** ó **X**.

$$0xA3 \rightarrow A3 \text{ hex} = 163 \text{ decimal}$$

Esta constante se puede escribir también de cualquiera de las 3 formas siguientes:

0XA3 0xa3 0xA3

Las constantes enteras se consideran positivas a menos que vayan precedidas por el signo menos (-):

-150 -063 -0xA

Constantes Reales

También se denominan constantes de **coma flotante**. Tienen el siguiente formato:

[parte entera] [.parte fraccionaria] [exponente de 10]

Cualquiera de las 3 partes es opcional, pero si no hay parte entera debe haber parte fraccionaria y viceversa. El exponente de 10 tiene el formato

{E|e}exponente

pudiendo ser el exponente un número positivo o negativo. Son constantes válidas:

13.21 21.37E1 0.230001 32e2 -81e-8 -.39 -.39E-7

Constantes de caracteres

Pueden ser de 2 tipos:

- Simples
- Cadenas de caracteres

Simples: Están formadas por un solo carácter y se encierran entre comillas simples. Por ejemplo:

'a' 'A' '9'

Los caracteres ASCII no imprimibles se definen mediante la barra invertida (\) según el cuadro que se muestra a continuación. En él también se muestra la representación de los caracteres barra invertida, comilla simple y comillas dobles, que en C tienen un tratamiento especial.

CÓDIGO ASCII	***CARÁCTER BARRA***	***SIGNIFICADO***
7	*\a*	*Alarma (Beep)*
8	*\b*	*Retroceso (BS)*
9	*\t*	*Tabulador Horizontal (HT)*
10	*\n*	*Nueva Línea (LF)*
11	*\v*	*Tabulador Vertical (VT)*
12	*\f*	*Nueva Página (FF)*
13	*\r*	*Retorno*
34	*\"*	*Comillas dobles*
39	*\'*	*Comilla simple*
92	**	*Barra invertida*

También se pueden representar caracteres ASCII mediante su código octal o hexadecimal, usando el formato:

\numoctal o bien **x**numhexadecimal

que representan, respectivamente, el carácter cuyo código ASCII es **numoctal** o **numhexadecimal**. Así, la letra A puede representarse de cualquiera de las tres formas que se indican a continuación:

'A' '\101' '\x41'

No es válido '\X41'. Cualquier otro carácter después de \ se interpreta literalmente. Así \N se interpreta como la letra N.

Cadenas: Son secuencias de caracteres simples encerradas entre comillas dobles. A las cadenas de caracteres el compilador les añade un carácter nulo ('\0') de terminación, y los almacena como una matriz de caracteres. Así, la cadena "Hola" está compuesta por los 5 caracteres 'H', 'o', 'l', 'a','\0'.

Identificadores

Son los nombres dados a variables, funciones, etiquetas u otros objetos definidos por el programador. Un identificador puede estar formado por:

- Letras (mayúsculas o minúsculas)
- Números
- Carácter de subrayado

con la condición de que el primer carácter no sea un número. En determinados casos, que se estudiarán en el Capítulo 9, un identificador de un dato puede incluir el punto.

Ejemplos de identificadores válidos son:

Precio_Venta
Num1
_123
D_i_5

No son válidos:

Precio Venta	Lleva un espacio en blanco
1Num	Empieza por un número
Precio-Venta	Lleva un guión

De un identificador sólo son significativos los 32 primeros caracteres.

Palabras reservadas

Son palabras especiales que no pueden usarse para nombrar otros elementos del lenguaje. En el capítulo anterior vimos algunas de ellas, como **int** y **float**. El número de palabras reservadas en C es significativamente menor que el de otros lenguajes. En Turbo C hay 43, algunas más en Turbo C++ y menos en ANSI C. Durante el resto de capítulos se irán conociendo.

Es preciso insistir en que C hace distinción entre mayúsculas y minúsculas. Por lo tanto, la palabra reservada **for** no puede escribirse como **FOR**, pues el compilador no la reconoce como una instrucción, sino que la interpreta como un nombre de variable.

Comentarios

Como se vio en el capítulo anterior, el compilador reconoce como comentario cualquier grupo de caracteres situados entre /* y */, aunque estén en diferentes líneas. Por ejemplo,

```
/*  Este es un comentario que
    ocupa más de una línea */
```

Estos comentarios pueden anidarse en Turbo C++, aunque no es aconsejable para permitir la compatibilidad del código.

Se pueden definir comentarios de una sola línea mediante //.

```
//  Este comentario ocupa una sola línea
```

En el caso de comentarios de una sola línea no hay indicador de fin de comentario.

Operadores, expresiones, sentencias

Un operador es un símbolo que indica alguna operación sobre uno o varios objetos del lenguaje, a los que se denomina **operandos**.

Atendiendo al número de operandos sobre los que actúa un operador, estos se clasifican en:

- Unarios: actúan sobre un solo operando
- Binarios: " " 2 operandos
- Ternarios: " " 3 "

Atendiendo al tipo de operación que realizan, se clasifican en :

- Aritméticos
- Relacionales
- Lógicos
- De tratamiento de bits
- Especiales

Estudiaremos en este capítulo la mayor parte de ellos. Iremos viendo el resto a medida que se necesiten.

Los operadores, junto con los operandos, forman **expresiones**. En una expresión, los operandos pueden ser constantes, variables o llamadas a funciones que devuelvan valores (como la función **multiplica ()** que aparece en la página 8).

Una expresión se convierte en una **sentencia** cuando va seguida de un punto y coma. Cuando un grupo de sentencias se encierran entre llaves { }, forman un **bloque**, sintácticamente equivalente a una sentencia.

Operadores aritméticos

Los operadores aritméticos se exponen en el cuadro siguiente:

	OPERADOR	***DESCRIPCIÓN***
UNARIOS	-	*Cambio de signo*
	- -	*Decremento*
	++	*Incremento*
BINARIOS	-	*Resta*
	+	*Suma*
	*	*Producto*
	/	*División*
	%	*Resto de división entera*

Los operadores -, + y * funcionan del mismo modo que en el resto de los lenguajes de programación.

El valor devuelto por el operador / depende del tipo de los operandos. Si estos son enteros, devuelve la parte entera del cociente; si alguno de ellos es real, devuelve el resultado como número real. El operador % es equivalente al operador **mod** de Pascal o Quick-BASIC. Proporciona el resto de la división entera de los operandos, que han de ser enteros. Por ejemplo, dadas las sentencias

```
int x, y;

x = 9;
y = 2;
```

la operación **x / y** devuelve el valor **4**, mientras que la operación **x % y** devuelve **1**. Sin embargo, después de las sentencias

```
float x;
int y;
```

```
x = 9.0;
y = 2;
```

la operación **x / y** devuelve **4.5**, no pudiéndose aplicar, en este caso, el operador **%** puesto que uno de los operandos no es entero.

Los operadores ++ y -- aumentan o disminuyen, respectivamente, en una unidad el operando sobre el que actúan. Así, las expresiones

```
x++;                    x- -;
```

producen el mismo efecto sobre la variable **x** que

```
x = x + 1;              x = x - 1;
```

y el mismo que

```
++x;                    --x;
```

Es importante tener en cuenta la posición de los operadores ++ y -- cuando se encuentran dentro de una expresión más compleja. Si el operador está antes de la variable, la operación de incremento o decremento se realiza antes de usar el valor de la variable. Si el operador está después, primero se usa el valor de la variable y después se realiza la operación de incremento o decremento. Para aclarar esto, tomemos un ejemplo. Después de las sentencias

```
x = 10;
y = ++x;
```

los valores que se almacenan en las variables **x** e **y** son, en ambos casos, 11. Puesto que el operador está antes de la variable **x**, primero se incrementa ésta (asumiendo el valor 11), y después se asigna el valor de **x** a la variable **y**. Sin embargo, después de las sentencias

```
x = 10;
y = x++;
```

los valores para **x** e **y** serán, respectivamente, 11 y 10. En este caso, puesto que el operador está después de la variable **x**, primero se usa el valor de ésta (10) y se asigna a **y**; después se realiza la operación de incremento, pasando **x** a almacenar el valor 11.

Operadores relacionales

Se usan para expresar condiciones y describir una relación entre dos valores. En la página siguiente se muestra una tabla con todos ellos.

Estos operadores se usan en sentencias del tipo

```
if (a == b) printf ("Son iguales");
```

que debe leerse "si el contenido de la variable **a** es igual al de la variable **b** muestra en pantalla la frase *Son iguales*".

El resultado de una expresión relacional sólo puede ser *verdadero* o *falso*, lo que en C se identifica con los valores distinto de cero y cero, respectivamente. En la sentencia anterior, la expresión **a == b** se evaluará como 0 si **a** y **b** son diferentes, y como distinto de 0 si son iguales.

	OPERADOR	***DESCRIPCIÓN***
BINARIOS	>	*Mayor que*
	>=	*Mayor o igual que*
	<	*Menor que*
	<=	*Menor o igual que*
	==	*Igual que*
	!=	*Diferente que*

Operadores lógicos

Actúan sobre expresiones booleanas, es decir, sobre valores *verdadero* o *falso* generados por expresiones como las explicadas en el caso anterior. Son los siguientes:

	OPERADOR	***DESCRIPCIÓN***
UNARIOS	!	*not*
BINARIOS	&&	*and*
	||	*or*

El resultado de una operación lógica viene dado por su tabla de verdad. La tabla de verdad de los operadores **!**, **&&** y || se muestra a continuación:

a	***b***	***!a***	***a && b***	***a || b***
F	*F*	*V*	*F*	*F*
F	*V*	*V*	*F*	*V*
V	*F*	*F*	*F*	*V*
V	*V*	*F*	*V*	*V*

Dado que el lenguaje interpreta como falso el valor 0 y como cierto cualquier valor diferente de cero, se pueden emplear operadores aritméticos en las expresiones lógicas y de comparación. Por ejemplo, si **x**, **y** y **z** almacenan, respectivamente, los valores 20, 4 y 5, las expresiones siguientes son válidas:

x == y **Se interpreta como FALSO (0)**
x = y **Se interpreta como VERDADERO (4)**[1]
x == (y * z) **Se interpreta como VERDADERO (1)**

Operadores de tratamiento de bits

C incorpora ciertas operaciones sobre bits propias del Lenguaje Ensamblador, como desplazamientos o manipulación individual de bits. Los operadores que realizan estas operaciones son los siguientes:

	OPERADOR	***DESCRIPCIÓN***
UNARIOS	~	*not*
BINARIOS	&	*and*
	|	*or*
	^	*or exclusivo*
	>>	*desplazamiento a la derecha*
	<<	*desplazamiento a la izquierda*

Los operadores **&** (and), | (or) y **~** (not) se rigen por la misma tabla de verdad que gobierna a los operadores lógicos equivalentes (**&&**, ||, **!**). La diferencia entre unos y otros consiste en que **&**, | y **~** actúan a nivel de bits individuales y no sobre valores completos como **&&**, || y **!**. Así, si las variables **a** y **b** almacenan, respectivamente, los valores

a ← 0xA1B2
b ← 0xF0F0

las siguientes expresiones se evalúan como sigue:

a && b → 1 (Verdadero)
a || b → 1 (Verdadero)
!a → 0 (Falso)

a & b → 0xA0B0
a | b → 0xF1F2
~a → 0x5E4D

Los operadores **&**, |, y **~** son idénticos, respectivamente, a las instrucciones del Lenguaje Ensamblador AND, OR y NOT. Por tanto, el operador **&** permite poner ciertos bits a 0 dejando el resto como estaban, el operador | permite poner ciertos bits

[1] Veremos más adelante que el operador = devuelve el valor asignado. En este caso, la expresión .**x = y** devuelve el valor **4**, que se evalúa como cierto.

a 1 dejando el resto inalterado, y el operador ~ cambia los bits 1 por 0 y viceversa (Ver ejercicio 10 al final del capítulo).

El operador ^ (or exclusivo) es idéntico a la instrucción XOR de Lenguaje Ensamblador. Su tabla de verdad es la siguiente.

a	***b***	***a^b***
F	*F*	*F*
F	*V*	*V*
V	*F*	*V*
V	*V*	*F*

Por ejemplo, si **a** y **b** almacenan los valores

a ← 0xA1B2
b ← 0x1234

la siguiente expresión produce el resultado indicado

a ^ b → 0xB386

El operador ^ permite cambiar ciertos bits a su valor contrario, dejando el resto sin modificar (Ver ejercicio 10 al final del capítulo).

Los operadores de desplazamiento >> y <<, mueven todos los bits de una variable a la derecha o a la izquierda, respectivamente, un número de posiciones determinado. El formato general es:

variable << n o bien **variable >> n**

Así, la sentencia

```
a = b << 4;
```

almacena en **a** el contenido de **b**, después de realizar un desplazamiento de **b** de 4 bits a la izquierda. El contenido de **b** permanece inalterado.

Al igual que en Lenguaje ensamblador, C distingue entre desplazamientos aritméticos y lógicos:

- **Desplazamientos aritméticos:** Se realizan sobre datos enteros y mantienen el signo.
- **Desplazamientos lógicos:** Se realizan sobre datos declarados como *sin signo* (**unsigned**)[2] y simplemente añade ceros.

[2] **unsigned** es una palabra reservada que se puede aplicar al tipo de dato **int**. Las variables así declaradas se entienden siempre como positivas.

Veamos un ejemplo. Sea **a** una variable declarada como entera (con signo), que almacena el valor hexadecimal **A1B2**:

```
a ← 0xA1B2
```

a = a << 4	produce	**a ← 0x1B20**
a = a >> 4	produce	**a ← 0xFA1B**

Sin embargo, si **a** se declara como **unsigned**,

a = a << 4	produce	**a ← 0x1B20**
a = a >> 4	produce	**a ← 0x0A1B**

Operadores de asignación

Las asignaciones se realizan mediante el operador =. El uso de este operador tiene ciertos aspectos que lo distinguen del de otros lenguajes. En primer lugar, se puede emplear cualquier número de veces en una expresión. Así, podemos tener sentencias como

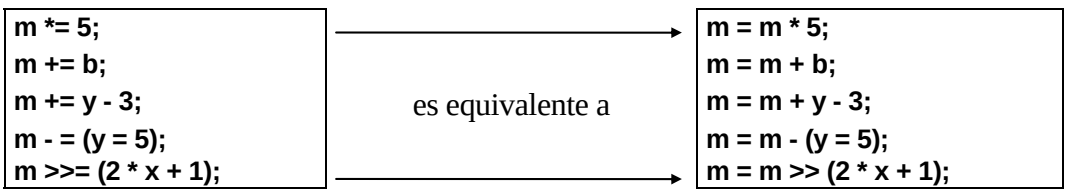
```
a = b = c = 3;
```

que asigna el valor 3 a las variables **a**, **b** y **c**. Esto es así porque la operación de asignación, además de asignar el valor, devuelve el valor asignado. Así, la expresión **c = 3** devuelve el valor **3**, que se asigna a **b**, y así sucesivamente. También son posibles sentencias como

```
x = 3;
y = x + (z = 6);
```

que asigna a **x**, **y**, y **z** los valores 3, 9 y 6, respectivamente.

El operador de asignación se combina con los operadores *****, /, **%**, +, -, <<, >>, **&**, |, ^, para operaciones acumulativas. Por ejemplo,



Operador condicional (?:)

Es un operador ternario que se usa para reemplazar sentencias simples del tipo **if...else**. El formato de este operador es:

<condición> ? <expresión_sí> : <expresión_no>

Funciona del siguiente modo: primero se evalúa **<condición>**; si es verdadera se evalúa **<expresión_sí>**, en caso contrario se evalúa **<expresión_no>**. Por ejemplo, en la sentencia

```
y = (x > 9 ? 100 : 200);
```

la variable **y** toma el valor 100 cuando **x > 9**, y el 200 en caso contrario.

Operadores de punteros (&, *)

Estos operadores, desafortunadamente, se identifican con el mismo símbolo que el AND de manipulación de bits y el producto, respectivamente. De cualquier forma no hay posibilidad de confusión, pues actúan en contextos diferentes.

	OPERADOR	***DESCRIPCIÓN***
UNARIOS	*&*	*Dirección de*
	***	*Contenido de*

El operador **&** actúa sobre una variable y devuelve su dirección de memoria. Después de

```
dir_a = &a;
```

la variable **dir_a** almacena la dirección de **a**.

El operador ***** actúa sobre variables que contienen direcciones (**punteros**) y devuelven el contenido de la posición de memoria almacenada en la variable sobre la que actúan. Si **dir_a** es una variable que almacena una dirección de memoria, la expresión

```
b = *dir_a;
```

almacena en la variable **b** el contenido de la posición de memoria apuntada por **dir_a**. De igual modo, la expresión

```
*dir_a = 80;
```

almacena el valor **80** en la dirección de memoria apuntada por **dir_a**.[3]

Después de la siguiente secuencia, las variables **m** y **z** almacenan los valores **4** y **2** respectivamente.

[3] Para que las sentencias en las que interviene la variable **dir_a** funcionen correctamente, ésta debe declararse de un modo especial, que explicaremos en el Capítulo 7, en el que se estudiarán los punteros.

```
m = 1;
n = 2;
direcc = &m;          (direcc ← dirección de m)
*direcc = 4;          ( m ← 4)
direcc = &n;          (direcc ← dirección de n)
z = *direcc;          (z ← 2)
```

Operador de tamaño (sizeof)

Es un operador unario que devuelve el tamaño en bytes del operando. La sintaxis es:

sizeof (m)

y devuelve los valores 1, 2 y 4 si **m** es, respectivamente, de tipo carácter, entero o float. Si **m** es una matriz devuelve el tamaño, en bytes, ocupado por ella. También devuelve el tamaño de ciertos objetos del C, denominados estructuras, que se estudiarán en el Capítulo 8.

Operador secuencial (,)

Se utiliza para concatenar expresiones. El lado izquierdo de la coma se evalúa primero. Por ejemplo, después de la expresión

```
x = ( y = 45, y++, y * 2);
```

las variables **x** e **y** almacenan, respectivamente, los valores 92 y 46.

Operadores . y ->

Se estudiarán detenidamente en el Capítulo 8, dedicado a uniones y estructuras.

Operador de moldeado (cast)

Permite cambiar el tipo de una expresión. La sintaxis es:

(tipo) expresión

donde **tipo** es uno de los tipos básicos de datos que estudiaremos en el próximo capítulo (en el anterior ya se vieron **int** y **float**). Por ejemplo, en la secuencia

```
int x;
x = 5;
y = x / 2;
```

el valor asignado a la variable **y** será 2, pues / realiza una división entera. Si se desea que **y** tenga parte fraccionaria, se utilizará el operador de moldeado:

```
int x;
x = 5;
y = (float) x / 2;
```

Ahora, la variable **y** almacena el valor 2.5.

De cualquier modo, hay que ser cuidadosos con este operador. En la secuencia

```
int x;
x = 5;
y = (float) (x / 2);
```

no se asigna a **y** el valor 2.5, sino **2**, ya que los paréntesis que envuelven a la expresión **x / 2** hacen que primero se efectúe la división entera y luego se convierta a float.

Operadores [ ] y ( )

Los corchetes se utilizan para acceder a los elementos de una matriz. Se estudiarán en el Capítulo 7.

Los paréntesis sirven para clarificar una expresión o para modificar las reglas de prioridad entre operadores. Estas reglas de prioridad quedan reflejadas en la tabla de la página siguiente.

Las prioridades de esta tabla pueden alterarse por medio de los paréntesis. Cuando alguna expresión se enmarca entre paréntesis, se evalúa primero. Si hay anidamiento, se evalúan primero los más internos. Por ejemplo, la expresión

```
x + 5 * y
```

se evalúa del siguiente modo:

1. Se calcula **5 * y**, pues el operador ***** tiene mayor prioridad que el operador **+**.
2. Se suma a **x** el resultado de la operación anterior.

Sin embargo, si escribimos

```
(x + 5) * y
```

los paréntesis modifican la prioridad, evaluándose en primer lugar la expresión **x + 5**, y multiplicando este valor por **y**.

Cuando en una expresión intervienen operadores con el mismo nivel de prioridad, se evalúan en el sentido indicado en la primera columna de la tabla.

Evaluación a igual nivel de prioridad	*Nivel de prioridad*	*Operadores*
→	*1º*	*( ) [ ] . ->*
←	*2º*	*! ~ ++ -- (cast) *[4] *&*[5] *sizeof*
→	*3º*	* / %
→	*4º*	+ -
→	*5º*	<< >>
→	*6º*	< <= > >=
→	*7º*	== !=
→	*8º*	&
→	*9º*	^
→	*10º*	|
→	*11º*	&&
→	*12º*	||
←	*13º*	*?:*
←	*14º*	*operadores de asignación*
→	*15º*	,

Efecto lateral

Cuando en una expresión se emplea más de una vez un mismo operando, y en alguna de ellas se modifica su valor, pueden producirse errores de efecto lateral. Por ejemplo, sea la secuencia

```
i = 2;
matriz[i] = x + (i = 4);
```

En la segunda sentencia, el elemento **matriz[i]** se refiere el elemento i-ésimo de un vector llamado **matriz**. Sin embargo, en esa expresión no se asegura que el resultado se almacene en **matriz[2]** o en **matriz[4]**.

Para evitar este tipo de problemas es necesario seguir las siguientes reglas:

- No utilizar operadores ++ o -- sobre variables que se empleen más de una vez en una expresión.

[4] Operador *contenido de*.
[5] Operador *dirección de*.

- No utilizar operadores ++ o -- sobre variables que se empleen más de una vez como argumento de una función.

Las directivas #include y #define

Estudiaremos ahora dos directivas que se utilizan prácticamente en todos los programas C.

#include

Indica al compilador que incluya un archivo fuente. El formato es

```
#include "nom_fich"
```

o bien,

```
#include <nom_fich>
```

siendo **nom_fich** un nombre válido de fichero DOS.

Puede decirse que cuando el compilador se encuentra una línea **#include**, reemplaza dicha línea por el fichero **nom_fich**.

El uso de comillas dobles "" o ángulos < > afecta a dónde se buscará el fichero **nom_fich**. En el primer caso se busca en el siguiente orden:

1. Directorio en el que se está trabajando.
2. Directorios indicados al compilar el programa desde la línea de órdenes del DOS.
3. Directorios indicados en la implementación de Turbo C.

En el segundo caso el orden de búsqueda es el siguiente:

1. Directorio definido con la opción -I del compilador.
2. Directorio indicado en el Menú Principal de Turbo C.
3. Directorios definidos en la implementación de Turbo C.

En ambos casos se supone que el nombre del fichero no va acompañado de ninguna trayectoria de directorio. En caso de que el nombre del fichero incluya una ruta de acceso, sólo se buscará en ese directorio. Así, en el caso

```
#include "C:\MIDIR\MIFICH.H"
```

o bien

```
#include <C:\MIDIR\MIFICH.H>
```

la búsqueda se realiza sólo en el directorio **C:\MIDIR**, produciéndose un mensaje de error si no se encuentra el archivo.

La directiva **#include** se usa principalmente para la inclusión de los denominados **archivos de cabecera**. La inclusión de estos archivos es necesaria cuando se utilizan las **funciones de biblioteca**. Estas funciones emplean tipos de datos y variables propias que se definen en esos archivos. Por ejemplo, el archivo de cabecera requerido por las funciones **printf()** y **scanf()** (junto con otras muchas más) se llama **stdio.h**, y por ello siempre que un programa maneje esas funciones deberá incluir una línea como

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

De cualquier modo, puede utilizarse para incluir cualquier archivo fuente.

#define

En su forma más simple se usa para asociar a un identificador una cadena de caracteres que lo sustituirá cada vez que lo encuentre el compilador en el programa. Al identificador se le denomina **macro** y al proceso de sustituir el identificador por la cadena de caracteres se le llama **sustitución de macro**. El formato general para esta directiva es

```
#define   macro  cadena
```

Por ejemplo, en

```
#define   CIERTO      1
#define   FALSO       0
```

cuando el compilador encuentra el nombre de macro **CIERTO** lo sustituye por el valor **1**. Cuando encuentra el nombre **FALSO** lo sustituye por el valor **0**. También es válido

```
#define   MENSAJE     "Error de E/S \n"
   ...
   ...
printf (MENSAJE);
```

que es equivalente a la sentencia

```
printf ("Error de E/S \n");
```

No se produce el proceso de sustitución de macro en el caso

```
printf ("MENSAJE");
```

que muestra en pantalla la cadena de caracteres **MENSAJE**. También se pueden escribir sentencias como

```
#define     TECLA     printf ("\nPulse cualquier tecla...")
    ...
    ...
TECLA;
```

Una vez definida una macro, puede utilizarse en la definición de macros posteriores:

```
#define     A       1
#define     B       2
#define     TRES    A + B
```

Si la definición de la macro es demasiado larga, se indica mediante una barra invertida \ y se continúa la definición en la siguiente línea.

```
#define     LARGO"Este mensaje es \
demasiado largo"
```

Por convenio, los nombres de macros se escriben en mayúscula.

Ejercicios

1. Representa de 3 maneras diferentes el carácter nueva línea.

2. Indica el significado de cada uno de los siguientes elementos

a) 12	b) 012	c) 0x12	d) 0X12
e) '\N'	f) '\x12'	g) '\x01b'	h) '\034'

3. Indica cuáles de los siguientes identificadores no son correctos y por qué.

a) contador	b) CONTADOR	c) _hola	d) hola_
e) dias2	f) 2dias	g) Suma_Total	h) Suma-Total

4. Sean **x**, **y**, **z**, **u**, **v** y **t**, variables que contienen, respectivamente, los valores 2, 3, 4, 5, 6 y 7, ¿qué almacenarán después de ejecutar las siguientes sentencias?

```
x++;
y = ++z;
t = - -v;
v = x + (y *= 3) / 2;
u = x + y / 2;
```

5. ¿Qué diferencia hay entre '\a' y '\A'? ¿Y entre 'A' y "A"?

6. Indica cuáles de las expresiones siguientes son verdaderas y cuáles falsas, suponiendo que **x**, **y**, **z** y **t**, almacenan, respectivamente los valores 20, 10, 5 y 2.

a) **x > y && z > t**
b) **x < y && z > t**
c) **x < y || z > t**
d) **!0**
e) **!1**
f) **0 != 1**
g) **0 != !1**
h) **0 == !1**
i) **200 || x**
j) **x * 0**
k) **x * y**
l) **!!0**
m) **!(0 == 0)**
n) **10 > 5 && !(10 < 9) || 3 <= 4**
ñ) **1 && !0 || 1**
o) **1 && !(0 || 1)**

7. Sea **x** una variable entera que almacena el valor **10** ¿Qué almacenará **y** después de las siguientes sentencias?

a) **y = (x > 9 ? ++x : -- x);**
b) **y = (x > 9 ? x++ : x--);**

¿Y si **x** almacenase el valor **8**?

8. Una temperatura en grados centígrados C, puede ser convertida en su valor equivalente de la escala Fahrenheit de acuerdo a la siguiente fórmula:

$$F = (9 / 5) C + 32$$

Escribe un programa C que solicite una temperatura en grados centígrados por teclado y presente en pantalla la temperatura Fahrenheit equivalente. (ATENCIÓN a la división 9/5).

9. Escribe un programa que lea un número hexadecimal entero del teclado (código de formato **%x**), intercambie los bytes alto y bajo, y presente el resultado en hexadecimal en la pantalla. (NOTA: Declara la variable que almacenará el valor hexadecimal como **unsigned int** y utiliza los operadores << y >>).

10. Escribe un programa que lea un número hexadecimal del teclado y manipule sus bits de acuerdo al siguiente criterio:

- Poner a 0 los bits 0, 3, 6 y 9.
- Poner a 1 los bits 2, 5, 8 y 11.
- Cambiar el estado del resto.

Presenta en pantalla el resultado en hexadecimal. (NOTA: Declara la variable que almacenará el valor hexadecimal como **unsigned int** y utiliza los operadores de tratamiento de bits).

11. Escribe un programa que lea dos números enteros del teclado y presente en pantalla el triple de la diferencia entre el mayor y el menor. (Para saber cuál es el mayor utiliza el operador **?:**)

12. Escribe un programa que capture por teclado un número entero de 4 cifras y lo redondee a la centena más próxima, presentando el resultado en pantalla.

3
Tipos básicos de datos

Tipos básicos de datos

En C es necesario declarar todas las variables antes de ser utilizadas. Al declarar una variable debemos asignarle uno de los 5 tipos básicos de datos. Estos tipos básicos son los que se muestran en la siguiente tabla.

TIPO	***PALABRA RESERVADA***	***TAMAÑO EN BYTES***
sin valor	*void*	*0*
carácter	*char*	*1*
entero	*int*	*2*
coma flotante (simple precisión)	*float*	*4*
coma flotante (doble precisión)	*double*	*8*

Toda variable declarada de uno de los tipos anteriores (salvo **void**) se supone que puede ser positiva o negativa a menos que se aplique uno de los modificadores que veremos más adelante.

El tipo **void** se usa para definir funciones que no devuelven ningún valor (no tienen sentencia **return**). Por ejemplo, ninguna de las funciones **main()** utilizadas hasta ahora devuelve valores. Esto provoca un "Warning" en la compilación, que se evita escribiendo **void main ()**, en lugar de escribir simplemente **main()**. Además, el tipo **void** se usa en ciertas operaciones de punteros que se estudiarán en el Capítulo 7.

Las variables de tipo **char** se usan para almacenar caracteres simples o números enteros de tamaño byte. Cuando se asigna un carácter simple a una variable de tipo **char**, se almacena su código ASCII. Esto permite hacer operaciones matemáticas con este tipo de variables. Por ejemplo, el siguiente programa

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

```
void main ()
{
    char caracter;

    caracter = '$';
    caracter *= 2;

    printf ("%c %d", caracter, caracter);
}
```

muestra en pantalla **H 72**, puesto que la asignación del carácter '$' a la variable **caracter** almacena en la variable su código ASCII (36). El **%c** es el código de formato para visualizar caracteres simples.

El tipo **int** se utiliza para declarar variables que almacenarán números enteros, sin decimales.

Los tipos **float** y **double** permiten almacenar números con decimales.

Cómo declarar variables

Una sentencia C que declare variables debe respetar la siguiente sintaxis:

[clase de almacenamiento][modificador] tipo variable[, variable, ...];

Las **clases de almacenamiento** y los **modificadores del tipo** se estudiarán en los próximos apartados del capítulo.

Ejemplos válidos de declaración de variables son:

```
char    letra1, letra2;
int     m, n;
float   p, q;
double  z;
```

Estas declaraciones deben situarse al principio de la función y antes de cualquier instrucción o llamada a función. Pueden ir precedidas por alguno de los modificadores de tipo que estudiaremos a continuación.

Modificadores del tipo de una variable

Un **modificador del tipo** es una palabra reservada que antecede a la definición del tipo de la variable, y permite cambiar el tamaño, rango y la característica de signo/sin signo. Los modificadores de tipo son:

- **short**
- **long**
- **signed**
- **unsigned**

Estos modificadores se aplican únicamente a los tipos **char** e **int**, salvo el **long**, que en un caso se puede aplicar al tipo **double**.

La tabla siguiente muestra los rangos y tamaños de cada tipo, según se apliquen los modificadores.

Tipo	***Modificadores***		***Rango***	***Ocupa***
char		*unsigned*	*0 a 255*	*1 byte*
		signed	*-128 a 127*	*1 byte*
int	*short*	*unsigned*	*0 a 65.535*	*2 bytes*
		signed	*-32.768 a 32.767*	*2 bytes*
	long	*unsigned*	*0 a 4.294.967.295*	*4 bytes*
		signed	*-2.147.483.648 a 2.147.483.647*	*4 bytes*
float			$\pm 3{,}4*10^{-38}$ *a* $\pm 3{,}4*10^{+38}$	*4 bytes*
double			$\pm 1{,}7*10^{-308}$ *a* $\pm 1{,}7*10^{+308}$	*8 bytes*
	long		$\pm 3{,}4*10^{-4932}$ *a* $\pm 1{,}1*10^{+4932}$	*10 bytes*

Respecto de estos modificadores hay que tener en cuenta los siguientes aspectos:

a) El modificador **signed** no es necesario. Todas las variables enteras, por defecto, se consideran con signo.

signed char equivale a **char**
signed int " **int**

b) El tipo de dato **char** no es afectado por los modificadores **short** y **long**.

short char equivale a **long char** y a **char**

c) Turbo C permite una notación breve para declarar enteros sin usar la palabra reservada **int**.

unsigned	equivale a	**unsigned int**
short	"	**short int**
long	"	**long int**
unsigned short	"	**unsigned short int**
unsigned long	"	**unsigned long int**

d) Los modificadores **signed** y **unsigned** provocan un error si se aplican a los tipos **float** y **double**.

e) Los modificadores **short** y **long** no afectan a **float**.

f) Al tipo **double** no le afecta el modificador **short**. El modificador **long** se puede aplicar a partir de la versión 2.00 de Turbo C.

g) La precisión que se puede obtener para los tipos de datos reales viene dada por la siguiente tabla:

TIPO	***PRECISIÓN***
float	*7 dígitos*
double	*15 dígitos*
long double	*19 dígitos*

Variables locales y variables globales

Dependiendo del lugar del programa en que se declare una variable, a ésta se le llama **local** o **global**. Una variable puede declararse en 4 lugares diferentes de un programa:

- Fuera de todas las funciones (**global**)
- Dentro de una función (**local** a la función)
- Dentro de un bloque enmarcado por llaves { } (**local** al bloque)
- Como parámetro formal (**local** a la función)

Una variable **global** es conocida por todo el programa, mientras que una variable **local** sólo es conocida por la función o bloque en donde está definida.

Variables globales

Cuando una variable está declarada fuera de todas las funciones, incluso fuera de **main ()**, se denomina **GLOBAL** y es conocida por todas las funciones del programa.

Variables locales

Una variable es **LOCAL** cuando se declara:

- dentro de una función
- como argumento de una función
- en un bloque de sentencias enmarcado por { }

En cualquiera de estos casos, la variable sólo es conocida por la función o bloque en donde está definida. Se crea al entrar a la función o bloque y se destruye al salir. Esto permite que dos variables locales a diferentes funciones pueden tener el mismo identificador, pero se consideran dos variables diferentes. Por claridad, no se recomienda utilizar nombres iguales para variables diferentes.

A continuación se muestra un programa que pretende ilustrar las diferencias entre variables locales y globales. En él se declara una variable global **y**, conocida por todas las partes del programa, y varias variables locales, todas con el mismo nombre **x**: una local a **main()**, otra local a un bloque dentro de **main()**, y otra local a una función.

```
#include <stdio.h>
int y;  /* Global. Conocida tanto por main() como por MiFuncion() */

main ()
{
    int x;                              /* Esta x es local a main () */
    y = 100;
    x = 1;
    printf ("x=%d, y=%d", x, y)         /* Visualiza x=1, y=100 */
    {                                   /* Comienza bloque */
        int x;                          /* Esta x es local al bloque */
        x = 2;
        printf ("x=%d, y=%d", x, y)     /* Visualiza x=2, y=100 */
        MiFuncion ()                    /* Visualiza x=3, y=100 */
        printf ("x=%d, y=%d", x, y)     /* Visualiza x=2, y=100 */
    }                                   /* Fin del bloque */
    printf ("x=%d, y=%d", x, y)         /* Visualiza x=1, y=100 */
}

MiFuncion ()
{
    int x;                              /* Local a MiFuncion() */
    x = 3;
    printf ("x=%d, y=%d", x, y)         /* Visualiza x=3, y=100 */
}
```

Clases de almacenamiento

Existen cuatro clases de almacenamiento, que se identifican con las palabras reservadas:

- **auto**
- **extern**
- **static**
- **register**

que anteceden al tipo de dato en la declaración de las variables.

El modo de almacenamiento de una variable determina

- qué partes del programa la conocen (**ámbito**)
- qué tiempo permanece en la memoria (**tiempo de vida**)

Variables automáticas (auto)

Una variable es de clase **auto** si es local a una función o bloque. La variable se crea cuando se llama a la función y se destruye al salir de ella.

El especificador **auto** es redundante, pues todas las variables locales son **automáticas** por defecto.

```
...
Funcion ( ...
{
    auto int a;        /* Declara la variable como local a Funcion */
...
```

Variables externas (extern)

Cuando una variable se define fuera de una función se clasifica como **externa** y tiene alcance global.

Una variable global puede volver a declararse dentro del cuerpo de una función, añadiendo la palabra reservada **extern**.

```
...
int a;                 /* Variable global */

main ()
{
    extern int a;      /* Es la misma variable */
...
```

Esta redeclaración es innecesaria cuando todas las funciones están en el mismo módulo. Sin embargo, si un programa consta de varios módulos diferentes, es

necesaria la palabra **extern** para que sea reconocida por módulos distintos del que contiene la declaración de la variable.

Por ejemplo, supongamos que PROG1.C y PROG2.C son dos módulos que se compilan separadamente y, posteriormente, se enlazan para formar un solo módulo ejecutable.

PROG1.C	***PROG2.C***
#include <stdio.h> **int x;** **int y;** **main ()** **{** **x = 10; y = 20;** **printf ("x=%d, y=%d", x, y);** **FuncionInterna ();** **FuncionExterna ();** **}** **FuncionInterna ()** **{** **printf ("x=%d, y=%d", x, y);** **}**	**#include <stdio.h>** **extern int x, y;** **FuncionExterna ()** **{** **printf ("x=%d, y=%d", x, y);** **}**

En el módulo PROG1.C (que se compilará a PROG1.OBJ) hay una llamada a **FuncionInterna()**. Esta función no necesita la redeclaración de **x** e **y** puesto que reside en el mismo módulo en el que estas variables están declaradas como globales. Sin embargo, hay una llamada a **FuncionExterna()**, y esta función reside en un módulo distinto (PROG2.C) que se compilará por separado (a PROG2.OBJ). Para que **FuncionExterna()** pueda reconocer a las variables **x** e **y** que están declaradas como globales en PROG1.C, es necesario redeclararlas como de clase **extern**. En caso contrario, PROG2.C "creería" que son variables locales propias no declaradas, por lo que se generaría un error en la compilación.

A partir de los módulos objeto PROG1.OBJ y PROG2.OBJ, mediante el enlazador, se crea un módulo ejecutable, por ejemplo PROG.EXE. Este programa visualizaría en 3 ocasiones la línea

x=10, y=20

puesto que para las tres funciones -**main()**, **FuncionInterna()** y **FuncionExterna()**- las variables **x** e **y** son las mismas.

Variables estáticas (static)

Una variable **estática** existe durante todo el tiempo de ejecución del programa.

Si se aplica la clase **static** a una variable **local**, Turbo C crea un almacenamiento permanente para ella, como si fuese global. La diferencia entre una variable **local estática** y una variable **global** es que la primera sólo es conocida por la función o bloque en que está declarada. Se puede decir que una variable **local estática** es una variable local que mantiene su valor entre llamadas.

Por ejemplo, cada vez que se llama a la función

```
incremento ()
{
    int n;

    n++;
    printf ("n=%d", n);
}
```

se visualiza **n=1**[1], mientras que si la función es

```
incremento ()
{
    static int n;

    n++;
    printf ("n=%d", n);
}
```

se visualiza, sucesivamente, **n=1**, **n=2**, ...

Cuando se aplica la clase **static** a una variable **global**, esta se convierte en *global sólo del archivo en que está declarada*. Así, en el ejemplo de la página anterior, si en PROG1.C en lugar de la línea declaramos **x** de la forma

```
static int x;
```

se produciría un error en el enlazado, incluso aunque en PROG2.C **x** esté declarada como **extern**.

Clase registro (register)

La clase **registro** es idéntica a la clase **auto**, por tanto sólo se puede aplicar a variables locales. Sólo se diferencian en el lugar en que se almacenan.

[1] En realidad no se puede saber cuál es el valor de **n** pues, como veremos, **n** no se inicializa con ningún valor. Pero para este ejemplo podemos suponer que se inicializa a 0.

Cualquier variable se almacena, por defecto, en la memoria. Si se declaran de clase **register**, Turbo C intentará mantenerlas en un registro de la CPU, en cuyo caso se acelerarán los procesos en los que intervengan. Puesto que el número de registros de la CPU es limitado, Turbo C mantendrá las variables en un registro, si es posible. En caso contrario la convertirá en **auto**.

Una variable de clase **registro** debe ser de los tipos **char** o **int**, y son ideales para contadores de bucles.

A estas variables no se les puede aplicar el operador **&** (dirección de).

Inicialización de variables

Al declarar una variable se le puede asignar un valor inicial, independientemente de que lo mantenga o no a lo largo de todo el programa. Así, son válidas sentencias como:

```
unsigned int x = 40000;
char letra = 'F';
register int b = 35;
char cadena[12] = "Buenos días";
```

Las variables globales o estáticas se inicializan a 0 si no se especifica ningún valor. Ambas deben inicializarse con expresiones constantes.

Las variables estáticas son inicializadas por el compilador una sola vez, al comienzo del programa.

Las variables automáticas y de registro tienen valores desconocidos hasta que se les asigne uno. Si tienen valores iniciales, se asignan cada vez que se ejecuta el bloque donde se definen.

Ejercicios

1. Indica el tipo de dato más adecuado para declarar variables que almacenen los valores cuyos rangos se muestran a continuación:

a) Números enteros del 0 al 70.000
b) Números enteros del 1 al 100
c) Números reales del 1 al 100
d) Números enteros del -100 al +100
e) Números reales del -100 al +100
f) Números enteros del 0 al 10.000
g) Cualquier letra mayúscula
h) Números enteros del 0 al 5.000.000.000
i) Números del -10.5 al 90.125

2. ¿Qué hace el siguiente programa?

```
#include <stdio.h>

void main ()
{
    unsigned char caracter;

    scanf ("%c", &caracter);
    caracter > 127 ? caracter++ : caracter--;
    printf ("%c", caracter);
}
```

3. Indica el tipo de variable (local/global), ámbito y tiempo de vida de las variables **a**, **b**, **c** y **d** de un programa que tenga la estructura que se muestra a continuación:

```
int a;

void main ()
{
    int b;
    ...
}

MiFuncion ()
{
    int c;
    static int d;
    ...
}
```

4
E/S básica

Tipos de E/S

Ya quedó dicho en el Capítulo 1 que C no tiene palabras reservadas específicas que se encarguen de la E/S. Estas operaciones se realizan mediante funciones de biblioteca. Estas funciones realizan operaciones de E/S que se clasifican del siguiente modo:

- **E/S de alto nivel**, definida por el estándar ANSI. También se denomina sistema de archivos con búffer.
- **E/S de bajo nivel**, o E/S de tipo UNIX.
- **E/S por consola**.

Los dos estándares ANSI y UNIX son redundantes. Turbo C admite los dos, pero no deben utilizarse dentro de un mismo programa funciones de ambos.

En el presente capítulo se estudiarán operaciones básicas de E/S por consola, entendiendo por ello las operaciones realizadas sobre la E/S estándar del sistema, por defecto teclado y pantalla. Veremos también que tanto la entrada como la salida puede redireccionarse a otros dispositivos, y cómo hacerlo, y también cómo realizar operaciones de salida a impresora. Por último, estudiaremos cómo realizar cierto control sobre la pantalla en modo texto (color, posicionamiento, etc.). Las operaciones de E/S a ficheros se estudiarán en los Capítulos 10 y 11.

E/S de caracteres

Funciones getche() y getch()

Son las dos funciones básicas que capturan caracteres simples por teclado. Los programas que utilizan estas funciones deben incluir el archivo de cabecera **conio.h** mediante una sentencia **#include**. Ambas funciones devuelven el carácter leído del teclado (sin esperar a pulsar la tecla ↵). La función **getche()** muestra en pantalla el

carácter tecleado. No así la función **getch()**. El uso correcto de estas funciones es mediante sentencias del tipo

```
char a;              /* También puede declararse como int */
...
...
a = getche ();       /* o bien   a = getch ();  */
```

Estas funciones no tienen argumentos. No es obligatorio capturar en una variable el carácter leído. Así, es correcto escribir:

```
printf ("\nPulse una tecla para continuar ...");
getch ();
```

que detiene el programa hasta que haya un carácter disponible en el teclado.

```
//Ejemplo: Visualiza el código ASCII de un carácter tecleado

#include <stdio.h>
#include <conio.h>

void main ()
{
    unsigned int caracter;

    printf ("\nPulse una tecla: ");
    caracter = getche ();
    printf ("\nSu código ASCII es %u", caracter);
}
```

Este programa produce un resultado casi idéntico utilizando la función **getch()**. La única diferencia es la no visualización del carácter que se teclea. El código de formato **%u** se utiliza en **printf()** para visualizar números declarados **unsigned**.

Función putchar()

Para la presentación de caracteres en pantalla se puede usar, además de **printf()** con el código de formato **%c**, la función **putchar()**. Los programas que usen esta función deben incluir el archivo de cabecera **stdio.h**. El uso correcto de **putchar()** es

```
char a;              /* También puede declararse como int */
    ...
    ...
putchar (a);
```

Si la variable **a** es de tipo **int** se visualiza en pantalla el carácter almacenado en el byte menos significativo. Las sentencias

```
char letra;
```

```
    ...
    ...
letra = 65;
putchar (letra);
```

provocan la visualización en pantalla del carácter **A**, cuyo código ASCII es 65. Es equivalente a

```
putchar ('A');
```

También son posibles sentencias como

```
putchar (getch ());
```

cuyo efecto es idéntico al que produce la función **getche()**.

E/S de cadenas de caracteres

En C no hay un tipo de dato específico para declarar cadenas de caracteres. Como ya quedó dicho, una cadena de caracteres se define como un vector de caracteres terminado con el carácter nulo '**\0**'. Así, para declarar una cadena de hasta 30 caracteres se debe escribir

```
char cadena[31];
```

Aunque se profundizará en el estudio de las cadenas de caracteres en el Capítulo 7, es necesario adelantar que C no realiza ningún control de límites sobre matrices. Ese cometido queda en manos del programador.

Función gets()

La función básica de entrada de cadenas caracteres por teclado es **gets()**. Esta función lee caracteres del teclado hasta pulsar la tecla ↵, almacenándolos en la cadena indicada en el argumento y añadiendo el carácter nulo al final.

```
char frase[31];
    ...
    ...
gets (frase);
```

La función **gets()** no hace comprobación del límite. En la secuencia anterior nada impedirá al usuario teclear más de 30 caracteres. La consecuencia de ello es imprevisible, aunque con toda seguridad el programa no funcionará correctamente.

Es importante tener en cuenta que no se permite la asignación de cadenas de caracteres mediante el operador =. Este tipo de operaciones se realizan mediante funciones de la biblioteca estándar. Alguna de ellas se estudiarán e el Capítulo 6.

```
frase = gets ();
```

Función puts()

La función básica de salida de cadenas de caracteres es **puts()**. Esta función escribe en pantalla la cadena de caracteres especificada en el argumento y provoca, además, un salto de línea. Es más rápida que **printf()** pero no permite formatear la salida.

Tanto **gets()** como **puts()** precisan la inclusión del fichero **stdio.h**.

```
//Ejemplo con gets() y puts()

#include <stdio.h>

void main ()
{
    char frase[31];

    puts ("\nTeclee una frase: ");
    gets (frase);
    printf ("\nUsted ha tecleado: %s", frase);
}
```

En el ejemplo anterior el código de formato **%s** se refiere a cadenas de caracteres. Un efecto similar al programa anterior lo produce la sentencia

```
puts (gets (frase));
```

E/S formateada

La E/S con formato se realiza por medio de las funciones **scanf()** y **prinft()** que ya conocemos. Las estudiaremos ahora con más detalle.

Función printf()

Como ya hemos visto, esta función precisa la inclusión del archivo de cabecera **stdio.h** y su formato general es:

printf (cadena de control, lista de argumentos);

La cadena de control determina como se presentarán el resto de argumentos mediante los caracteres de formato. Por cada carácter de formato de la cadena de control debe haber un argumento en la lista. En caso contrario el resultado no será correcto. Los caracteres de formato válidos vienen especificados en la tabla siguiente.

TIPO DE ARGUMENTO	***CARÁCTER DE FORMATO***	***FORMATO DE SALIDA***
Numérico	*%d*	*signed decimal int*
	%i	*signed decimal int*
	%o	*unsigned octal int*
	%u	*unsigned decimal int*
	%x	*unsigned hexadecimal int (con a, ..., f)*
	%X	*unsigned hexadecimal int (con A, ..., F)*
	%f	*[-]dddd.dddd*
	%e	*[-]d.dddd o bien e[+/-]ddd*
	%g	*el más corto de %e y %f*
	%E	*[-]d.dddd o bien E[+/-]ddd*
	%G	*el más corto de %E y %f*
Carácter	*%c*	*carácter simple*
	%s	*cadena de caracteres*
	%%	*el carácter %*
Punteros	*%n*	*se refieren a punteros y se*
	%p	*estudiarán en el Capítulo 7*

Mediante los caracteres de formato pueden controlarse ciertos aspectos de la presentación de los datos, como la longitud, el número de decimales y la justificación a izquierda o derecha.

Para indicar la longitud de un campo basta incluir un número entero entre el signo **%** y el carácter de formato. Así, **%5d** indica que se va a presentar un número entero en un campo de 5 posiciones, justificando a la derecha (mientras no se indique lo contrario la justificación es siempre a la derecha). Así,

printf ("\n[%5d]", 47);	visualiza	**[°°°47]**
printf ("\n[%4d]", 47);	visualiza	**[°°47]**
printf ("\n[%3d]", 47);	visualiza	**[°47]**

donde el símbolo **°** representará el resto del capítulo un espacio en blanco. Los corchetes se ponen sólo para ver el efecto de los caracteres de formato.

Si se desea justificación a la izquierda, se indica con un signo menos.

printf ("\n[%-5d]", 47);	visualiza	**[47°°°]**
printf ("\n[%-4d]", 47);	visualiza	**[47°°]**
printf ("\n[%-3d]", 47);	visualiza	**[47°]**

Cuando se presenta un valor numérico justificando a la derecha, puede hacerse que el campo se rellene con ceros en lugar de espacios en blanco, poniendo un 0 antes del indicador de tamaño. Así, la sentencia

printf ("\n[%05d]", 47); visualiza **[00047]**

Este efecto no se produce si se solicita justificación izquierda

printf ("\n[%-05d]", 47); visualiza **[47°°°]**

Cuando el dato sobrepasa el tamaño del campo, se imprime completo

printf ("\n[%3d]", 1234); visualiza **[1234]**

Si lo dicho anteriormente se aplica a cadenas de caracteres, el resultado es el que se indica a continuación:

printf ("\n[%5s]", "ABC"); visualiza **[°°ABC]**
printf ("\n[%05s]", "ABC"); visualiza **[°°ABC]**
printf ("\n[%-5s]", "ABC"); visualiza **[ABC°°]**
printf ("\n[%-05s]", "ABC"); visualiza **[ABC°°]**
printf ("\n[%5s]", "ABCDEF"); visualiza **[ABCDEF]**

Aplicado a caracteres simples

printf ("\n[%3c]", 'A'); visualiza **[°°A]**
printf ("\n[%-3c]", 'A'); visualiza **[A°°]**

Si se quiere dar formato a un número con parte decimal se utiliza el código de formato

%*m*.*n***f**

siendo **m** la longitud total del campo (incluyendo el punto) y **n** el número de decimales. Así, el especificador de formato **%7.2f** define un campo de 7 posiciones del siguiente modo: 4 para la parte entera, 1 para el punto y 2 para los decimales.

printf ("\n[%7.2f]", 12.3); visualiza **[°°12.30]**
printf ("\n[%07.2f]", 12.3); visualiza **[0012.30]**
printf ("\n[%-7.2f]", 12.3); visualiza **[12.30°°]**

Cuando el número de decimales es mayor que **n**, se redondea la última cifra

printf ("\n[%7.2f]", 12.348); visualiza **[°°12.35]**
printf ("\n[%7.2f]", 12.342); visualiza **[°°12.34]**

Si este formato se aplica a cadenas de caracteres o números enteros, **n** especifica el tamaño máximo del campo. Así, **%3.5s** define un campo de presentación para una cadena de al menos 3 caracteres y no más de 5. Si la cadena sobrepasa el tamaño máximo, se trunca.

printf ("\n[%3.5s]", "ABC"); visualiza **[ABC]**
printf ("\n[%-3.5s]", "ABC"); visualiza **[ABC]**
printf ("\n[%3.5s]", "ABCDEFG"); visualiza **[ABCDE]**
printf ("\n[%3.5s]", "AB"); visualiza **[°AB]**

printf ("\n[%5.3s]", "ABCDE"); visualiza **[°°ABC]**

Pueden utilizarse modificadores de formato para visualizar enteros **short** y **long**, poniendo, respectivamente, **h** y **l** antes de **d**, **u**, **o**, **x**, **X**.

%hd para visualizar variables **short int**
%ld para visualizar variables **long int**
%hu para visualizar variables **unsigned short int**
%lu para visualizar variables **unsigned long int**

El modificador **l** también se puede aplicar a **e**, **f**, **g**, **E**, **G**.

%lf para visualizar variables **long double**

Función scanf()

Es la función de entrada de datos con formato de propósito general que hemos visto en el Capítulo 1. La sintaxis es similar a la de **printf()**:

scanf (cadena de control, lista de argumentos);

aunque aquí la cadena de control no debe interpretarse igual que en **printf()**.

Clasificaremos los caracteres que pueden aparecer en la cadena de control en 6 categorías:

- Especificadores de formato
- Caracteres blancos
- Caracteres no blancos
- Carácter *
- Modificadores de longitud
- Juego de inspección

Especificadores de formato: Son los mismos que en **printf()** salvo **%g** y **%u** y añadiendo **%h**. Así, la sentencia

```
char a;                    /* También se puede declarar como int */
   ...
   ...
scanf ("%c", &a);
```

captura un carácter y lo almacena en la variable **a**. El operador **&** es necesario en **scanf()** para simular las *llamadas por referencia*[1], y hace que la función trabaje internamente con la dirección de la variable. No es necesario cuando el dato a capturar es una cadena de caracteres.

[1] En las *llamadas por referencia* a funciones, cualquier cambio operado sobre los parámetros dentro de la función tienen efecto sobre las variables utilizadas en la llamada. No ocurre así en las *llamadas por valor* en las que los cambios en los parámetros dentro de la función no se reflejan en las variables de la llamada. Estudiaremos esto con detalle en el Capítulo 6, dedicado a Funciones.

```
char cadena[80];
   ...
   ...
scanf ("%s", cadena);
```

aunque tendría el mismo efecto la sentencia

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

La razón por la que no es necesario el operador **&** cuando se trata de cadenas, es que el nombre de una cadena (en general, de una matriz) sin índices identifica la dirección del primer elemento.

Caracteres blancos: Son separadores de datos de entrada. Se entiende como carácter blanco cualquiera de los siguientes:

- Espacio(s) en blanco
- Tabulador(es)
- Return, Intro o salto de línea (↵)

Cuando se especifica un carácter blanco en la cadena de control, **scanf()** lee, pero no guarda, cualquier número de caracteres blancos hasta que encuentre el primero que no lo sea. La sentencia

```
scanf ("%c %d %s", &a, &b, &c);
```

almacena en **a**, **b** y **c**, respectivamente, datos introducidos de cualquiera de los siguientes modos:

Y ↵	Y <TAB> 19 ↵	Y 19 Hola ↵
19 ↵	Hola ↵	
Hola ↵		

y cualquier combinación similar.

Hay que tener precauciones con el uso de los espacios en blanco. Una sentencia como

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

no volverá hasta que no se teclee un carácter no blanco, puesto que el espacio que hay después de **%s** hace que se lean y descarten espacios, tabuladores y saltos de línea.

Caracteres no blancos: Se toman como separadores de los datos de entrada. Por ejemplo, en la sentencia

```
scanf ("%dHOLA%c", &num, &car);
```

se lee un entero y se almacena en **num**, los caracteres **HOLA** y se descartan (es necesario teclearlos), y un carácter cualquiera y se almacena en **car**. Debido a ello hay que poner especial cuidado en no usar la cadena de control de **scanf()** como en **printf()**, y NO ESCRIBIR sentencias como:

```
scanf ("Teclee un número: %d", &num);
```

Carácter *: Cuando se coloca un ***** entre el símbolo **%** y el carácter de formato, se lee el tipo de dato especificado pero no se asigna a ninguna variable. La sentencia

```
scanf ("%d %*c %d", &num1, &num2);
```

lee un entero y lo almacena en **num1**, lee un carácter y lo descarta (no se asigna a ninguna variable, y lee otro entero y lo guarda en **num2**. De este modo, si a una sentencia como la anterior se responde tecleando algo como

10-20 ↵

se almacenaría el valor **10** en **num1** y el valor **20** en **num2**.

Modificadores de longitud: Es posible limitar el número de caracteres que se admiten en un campo escribiendo un número después del símbolo **%** del carácter de formato. Así, la sentencia

```
scanf ("%5s", frase);
```

captura cualquier cadena de caracteres tecleada, pero sólo almacena en **frase** los 5 primeros. Sin embargo el resto de caracteres no se descartan, sino que quedan en el búffer del teclado disponibles para ser leídos por otra sentencia **scanf()**. El siguiente programa ilustra el uso de **scanf()** con modificadores de longitud.

```
//Ejemplo de scanf() con modificador de longitud
#include <stdio.h>

void main ()
{
  long int n1, n2;

  printf ("\nTeclee 2 números enteros: ");
  scanf ("%5ld %5ld", &n1, &n2);
  printf ("\nLos números tecleados son %ld y %ld", n1, n2);
}
```

Si en este programa a la sentencia **scanf()** se responde con

123456 ↵

se muestra en pantalla

Los números tecleados son 12345 y 6

Juego de inspección: Se utiliza sólo con **%s** y permite especificar qué caracteres se aceptarán. Se indican entre corchetes. Por ejemplo

```
char frase[30];
   ...
   ...
scanf ("%[abc]s", frase);
```

admite sólo los caracteres **a**, **b**, **c**. Cuando se teclea un carácter que no está en el juego de inspección finaliza la entrada para esa variable. Sin embargo, la sentencia continuará admitiendo caracteres hasta que se teclee ↵, dejando los caracteres no capturados en el búffer del teclado, disponibles para otra sentencia **scanf()** posterior. Por ello, si se tienen dos sentencias como

```
scanf ("%[abc]s", frase);
scanf ("%s", cadena);
```

y se teclea, por ejemplo

abcbbapabcfgts ↵

en **frase** se almacenará la cadena **abcbba**, mientras que en **cadena** se almacenará **abcfgts**. El carácter **p**, primero diferente de **[abc]**, actúa como finalizador de la entrada para la variable **frase**.

Pueden especificarse rangos. Por ejemplo

```
scanf ("%[a-z]s", frase);
```

admite cualquier carácter comprendido entre la **a** y la **z**. También, la sentencia

```
scanf ("%[0-9][p-t]s", frase);
```

admite cualquier dígito numérico y los caracteres **pqrst**.

Es posible definir juegos inversos de inspección mediante el carácter ^. Éste le indica a **scanf()** que debe aceptar cualquier carácter *que no esté* en el juego de inspección. Por ello, la sentencia

```
scanf ("%[^0-9]s", frase);
```

finaliza la entrada al teclear un dígito numérico.

La función fprintf()

Esta función es prácticamente idéntica a **printf()**. Pero **fprintf()** permite asociar la salida a diferentes dispositivos (pantalla, impresora, archivo de texto u otro). La sintaxis correcta para esta función es

fprintf (dispositivo, cadena de control, lista de argumentos);

es decir, incluye un primer argumento que relaciona la salida con un dispositivo. En realidad, ese primer argumento debe ser un puntero a una estructura predefinida en C llamada **FILE**, de la que hablaremos en el Capítulo 10. De cualquier modo, para nuestros propósitos actuales, nos basta con identificarlo de algún modo con un dispositivo.

Como veremos en su momento, C asocia los dispositivos a unas zonas o búffers de memoria denominados **canales**. Cuando un programa empieza a ejecutarse se abren automáticamente 5 canales, entre ellos el asociado a la impresora. A este dispositivo se le referencia mediante la palabra **stdprn**.

Por lo tanto, **fprintf()** direcciona su salida a la impresora cuando se escribe de la forma

fprintf (stdprn, cadena de control, lista de argumentos);

Todas las demás características de **fprintf()** son idénticas a las de **printf()**.

Control de la pantalla de texto

El control de la pantalla puede establecerse por dos vías. En primer lugar, Turbo C proporciona gran número de funciones para controlar la pantalla tanto en modo texto como en modo gráfico. Estudiaremos en este apartado algunas funciones de control en modo texto básicas. Todas ellas necesitan que se incluya el archivo de cabecera **conio.h**. Otra forma de controlar la pantalla es por medio de secuencias de escape ANSI.

Funciones de biblioteca

Función clrscr(): Borra la pantalla y sitúa el cursor en la esquina superior izquierda. No necesita argumentos ni devuelve ningún valor.

Función clreol(): Borra una línea de la pantalla desde la posición del cursor hasta el final de dicha línea. Tampoco necesita argumentos ni devuelve valor alguno.

Funciones insline() y delline(): La función **insline()** inserta una línea en blanco debajo de la del cursor, desplazando el resto una línea hacia abajo. Análogamente, la función **delline()** elimina la línea en donde está el cursor, desplazando las de debajo hacia arriba.

Funciones gotoxy(), wherex() y wherey(): La función **gotoxy()** se encarga del posicionamiento del cursor. Se escribe

```
int fila, columna;
```

```
    ...
    ...
gotoxy (columna, fila);
```

Las funciones **wherex()** y **wherey()** devuelven las coordenadas de la posición actual del cursor:

```
int fila, columna;
    ...
    ...
columna = wherex ();
fila = wherey ();
```

En estas 3 funciones, las variables **fila** y **columna** deben declararse de tipo **int**.

Función movetext(): Copia texto de una zona de la pantalla a otra. La sintaxis de esta función es

```
int x1, y1, x2, y2, x3, y3;
    ...
    ...
movetext (y1, x1, y2, x2, y3, x3);
```

siendo:

- **x1, y1**: columna, fila de la esquina superior izquierda de la zona de la pantalla a desplazar.
- **x2, y2**: ídem de la esquina inferior derecha.
- **x3, y3**: columna, fila de la esquina superior izquierda de la pantalla a donde se desplaza el texto.

La función no borra el texto de la posición original. Devuelve 0 si alguna coordenada está fuera de rango, y 1 en caso contrario.

Funciones highvideo(), lowvideo() y normvideo(): Las dos primeras ponen, respectivamente, el texto de la pantalla en alta y baja intensidad. La función **normvideo()** restaura la intensidad de la pantalla al estado en que se encontraba al iniciarse el programa.

Funciones textcolor(), textbackground() y textattr(): La función **textcolor()** establece el color del texto que se presenta a partir del momento en que se ejecuta la función.

```
textcolor (color_de_carácter);
```

El parámetro **color_de_carácter** es un número comprendido entre 0 y 15. Cada uno de estos números tiene asociado un color. Además, en el archivo **conio.h** se define una macro para cada color. Esto queda reflejado en la tabla siguiente:

VALOR	***COLOR***	***MACRO***

0	*Negro*	*BLACK*
1	*Azul*	*BLUE*
2	*Verde*	*GREEN*
3	*Turquesa*	*CYAN*
4	*Rojo*	*RED*
5	*Morado*	*MAGENTA*
6	*Marrón*	*BROWN*
7	*Blanco*	*LIGHTGRAY*
8	*Gris*	*DARKGRAY*
9	*Azul intenso*	*LIGHTBLUE*
10	*Verde intenso*	*LIGHTGREEN*
11	*Turquesa intenso*	*LIGHTCYAN*
12	*Rojo intenso*	*LIGHTRED*
13	*Morado intenso*	*LIGHTMAGENTA*
14	*Amarillo*	*YELLOW*
15	*Blanco intenso*	*WHITE*
128	*Parpadeo*	*BLINK*

Teniendo esto en cuenta, para conseguir que un texto se presente, por ejemplo, en color verde, previamente a su escritura debe hacerse

```
textcolor (2);
```

o bien

```
textcolor (GREEN);
```

Sin embargo, esto no tiene efecto si se utilizan las funciones **printf()**, **puts()**, **gets()** y **putchar()**. En su lugar, deben utilizarse las funciones **cprintf()**, **cputs()**, **cgets()** y **putch()**, respectivamente. Sigue siendo válida la función **getche()**.

Para conseguir el parpadeo del carácter debe hacerse una operación OR entre el color y el valor 128 (BLINK). Por lo tanto, si queremos presentar un texto en amarillo parpadeante debemos escribir

```
textcolor (YELLOW | BLINK);
```

La función **textbackground()** establece el color de fondo para todo texto que se escriba en pantalla a continuación.

```
textbackground (color_de_fondo);
```

siendo **color_de_fondo** un valor comprendido entre 0 y 7, correspondiente a la tabla anterior.

La función **textattr()** establece el byte de atributo completo (color de carácter, color de fondo, parpadeo si/no e intensidad alta/baja). Se escribe

```
textattr (atributo);
```

donde **atributo** es un byte cuyo significado se muestra a continuación

7	6	5	4	3	2	1	0
P	**F**	**F**	**F**	**I**	**C**	**C**	**C**

donde CCC son los bits que codifican el color del carácter, I es el bit de intensidad, FFF indican color de fondo, y P es el bit de parpadeo. La forma más cómoda de usar esta función es la siguiente: se multiplica el número del color de fondo de la tabla por 16 y se hace una operación OR con el color de carácter. Si además se quiere que el texto parpadee, debe hacerse una operación OR con el valor 128 (BLINK). Así, para obtener un texto en amarillo parpadeando sobre fondo azul, debe hacerse:

```
textattr (YELLOW | BLINK | BLUE * 16);
```

Función textmode(): Asigna un determinado modo de vídeo a la pantalla de texto:

```
textmode (modo_de_vídeo);
```

Los valores permitidos para **modo_de_vídeo** y las macros asociadas se muestran en la tabla siguiente.

MODO DE VÍDEO	***DESCRIPCIÓN***	***MACRO***
0	*Blanco y negro, 40 columnas*	*BW40*
1	*Color, 40 columnas*	*CO40*
2	*Blanco y negro, 80 columnas*	*BW80*
3	*Color, 80 columnas*	*CO80*
7	*Monocromo*	*MONO*
-1	*Modo anterior*	*LASTMODE*

Al asignar un modo de vídeo se inicializa la pantalla.

Secuencias de escape ANSI

El archivo ANSI.SYS es un controlador de pantalla y teclado que permite cambiar la presentación de textos y gráficos en pantalla, controlar el cursor y cambiar las asignaciones de las teclas. Cada una de estas funciones se define mediante una *secuencia de escape ANSI*, que es una cadena de caracteres que comienza por los caracteres **ESC [**, siendo **ESC** el carácter **ESCAPE**, de código ASCII 27 (033 octal). Estas secuencias de escape están descritas a continuación.

OPERACIÓN	***SECUENCIA DE ESCAPE***	***DESCRIPCIÓN***
Desplazamientos	*ESC[#;#H*	*Coloca el cursor en la posición de la pantalla*

del cursor		*señalada con #;# (el primer número indica la fila y el segundo la columna)*
	ESC[#A	*Mueve el cursor # líneas hacia arriba*
	ESC[#B	*Mueve el cursor # líneas hacia abajo*
	ESC[#C	*Mueve el cursor # columnas hacia la derecha*
	ESC[#D	*Mueve el cursor # columnas hacia la izquierda*
	ESC[s	*Guarda la posición actual del cursor*
	ESC[u	*Recupera la posición del cursor previamente guardada con la secuencia anterior*
Borrado	*ESC[2J*	*Borra la pantalla y coloca el cursor en la esquina superior izquierda de la pantalla*
	ESC[K	*Borra todos los caracteres desde la posición actual del cursor hasta el final de la línea*
Establecer modo de gráficos	*ESC[#;...;#m*	*Llama a las funciones gráficas especificadas mediante los dígitos #. Estas funciones permanecen activas hasta la siguiente aparición de esta secuencia de escape. Los valores permitidos para # se muestran en la tabla de la página siguiente*
Establecer modo de vídeo	*ESC[=#h*	*Cambia el ancho o el tipo de pantalla. Los valores permitidos para # se muestran en la tabla de la página siguiente*
	ESC[=#l	*Restablece el modo utilizando los mismos valores que utiliza la secuencia anterior, salvo el modo 7 que desactiva el ajuste automático de línea. El último carácter de la secuencia es la L minúscula*
Reasignación de cadenas para el teclado	*ESC[#;cad;#p*	*Permite cambiar la definición de las teclas a una cadena específica. El símbolo # representa el código o códigos generados por la tecla a reasignar.* ***cad*** *es el código ASCII correspondiente a un solo carácter o a una cadena entre comillas.* *Ejemplo 1º:* ***ESC[65;66p*** *convierte la A en B.* *Ejemplo 2º:* ***ESC[0;59;"CLS";13p*** *Asigna a la tecla F1 (que genera los códigos 0 y 59) la cadena CLS junto con ↵ (código 13).*

Valores permitidos para # en la secuencia de escape ***ESC[#;...;#m***	*ATRIBUTOS DE TEXTO*	*0*	*Desactiva todos los atributos*
		1	*Alta intensidad activada*
		4	*Subrayado (monitor monocromo)*
		5	*Parpadeo activado*
		7	*Vídeo inverso activado*
		8	*Oculto activado*
	COLOR DE CARÁCTER	*30*	*Negro*
		31	*Rojo*
		32	*Verde*
		33	*Amarillo*
		34	*Azul*
		35	*Magenta*
		36	*Violeta*
		37	*Blanco*

	COLOR DE FONDO	*40* *41* *42* *43* *44* *45* *46* *47*	*Negro* *Rojo* *Verde* *Amarillo* *Azul* *Magenta* *Violeta* *Blanco*
Valores permitidos para # en la secuencia de escape ***ESC[=#h***		*0* *1* *2* *3* *4* *5* *6* *7* *13* *14* *15* *16* *17* *18* *19*	*40x25 monocromo (texto)* *40x25 color (texto)* *80x25 monocromo (texto)* *80x25 color (texto)* *320x200 color (gráficos)* *320x200 monocromo (gráficos)* *640x200 monocromo (gráficos)* *Activa ajuste automático de línea* *320x200 color (gráficos)* *640x200 color (gráficos 16 colores)* *640x350 mono (gráficos 2 colores)* *640x350 color (gráficos 16 colores)* *640x480 mono (gráficos 2 colores)* *640x480 color (gráficos 16 colores)* *320x200 color (gráficos 256 colores)*

Una secuencia de escape ANSI se envía, por ejemplo, mediante la función **printf()**. Con ello no se visualizan los caracteres de la secuencia de escape, sino que se consigue el efecto descrito para cada una de ellas. Pero para que estas secuencias tengan el efecto esperado es necesario instalar el controlador ANSI.SYS en el archivo CONFIG.SYS mediante una sentencia DEVICE o DEVICEHIGH. La sintaxis de la instalación de este controlador es:

DEVICE = [unidad:] [ruta] **ANSI.SYS**

de modo que si el archivo ANSI.SYS está en el directorio \DOS de la unidad C:, en el fichero CONFIG.SYS debe haber una línea

device = c:\dos\ansi.sys

o, si se desea instalar en memoria superior

devicehigh = c:\dos\ansi.sys

Veamos algunos ejemplos que ilustran el uso de estas secuencias de escape.

Posicionamiento del cursor: La secuencia de escape para posicionar el cursor es

ESC [f**;**c**H** **(**f = fila, c = columna)

Para situar el cursor en la fila 5, columna 20, se hace

```
printf ("\033[5;20H");
```

y produce exactamente el mismo efecto que

```
gotoxy (20, 5);
```

No obstante para este tipo de operaciones es más conveniente definir una función que reciba en los parámetros las coordenadas:

```
pon_cursor (int fila, int columna)
{
    printf ("\033[%d;%dH", fila, columna);
}
```

En un programa se llamaría a esta función, por ejemplo, mediante

```
pon_cursor (5, 20);
```

Borrado de pantalla: La secuencia ANSI que borra la pantalla es

ESC [2J

Por tanto, la sentencia

```
printf ("\033[2J");
```

produce el mismo efecto que la función de biblioteca **clrscr()**. Se puede abreviar la escritura de esta secuencia mediante una macro como

```
#define     CLS     printf ("\033[2J")
```

y utilizar en el programa sentencias

```
CLS;
```

Redefinición de teclas: Mostraremos el uso de esta secuencia mediante un ejemplo que asigna a la tecla **F1** (que al ser pulsada genera los códigos 0 y 59) la cadena de caracteres **DIR C:/P** ↵. Esto se consigue con la secuencia

ESC [0;59;"DIR C:/P";13p

Con una función **printf()** esto se escribe

```
printf ("\033[0;59;\"DIR C:/P\";13p");
```

o también

```
printf ("\033[0;59;%c%s%c;13p", 34, "DIR C:/P", 34);
```

donde hay que tener en cuenta que el código ASCII del carácter comilla doble (") es 34.

Redireccionamiento de la E/S

Las operaciones de E/S realizadas por funciones referidas a la consola (**printf()**, **scanf()**, ...) pueden redirigirse a otros dispositivos mediante los operadores

> redirección de la salida
< redirección de la entrada

Redirección de la salida

Sea el programa de la página 2, compilado y linkado como DOCENA.EXE. Si desde el indicador del DOS escribimos:

C:\> docena >prn ↵

la frase "Una docena son 12 unidades" no se muestra en pantalla, sino que se dirige al dispositivo **prn** (impresora). Se puede también redireccionar la frase a un fichero de texto llamado, por ejemplo, FRASE.TXT, mediante

C:\> edad >frase.txt ↵

Esta orden crea en el directorio raíz un fichero de texto llamado FRASE.TXT que almacena la frase "Una docena son 12 unidades". Si el fichero ya existe, se pierde su contenido anterior.

Cada vez que se redirecciona la salida a un fichero, éste se crea de nuevo. Así, la secuencia

C:\> prog1 >fich.txt ↵
C:\> prog2 >fich.txt ↵

crea un archivo FICH.TXT que almacena la salida únicamente de PROG2. Si se desea añadir la salida de un programa a un fichero existente sin destruir su contenido, se utiliza el operador >>. La secuencia

C:\> prog1 >fich.txt ↵
C:\> prog2 >>fich.txt ↵

almacena en FICH.TXT la salida de ambos programas PROG1 y PROG2.

Redirección de la entrada

Sea ahora el programa PIES.C de la página 5 compilado y linkado como PIES.EXE. Creemos un fichero de texto llamado DATO.DAT que almacene el número 3. Esto se consigue desde el indicador del DOS mediante la secuencia

C:\> copy con dato.dat ↵
3 ↵
^Z ↵
C:\>

Si escribimos

C:\> pies <dato.dat ↵

la sentencia **scanf()** del programa ya no lee el dato del teclado. Al estar la entrada redireccionada, lo toma del archivo DATO.DAT. Veríamos entonces en pantalla los mensajes generados por **printf()**. Algo como

¿Pies?:
3 pies = 0.925200 metros

Redirección de la entrada y de la salida

Se puede redireccionar a la vez tanto la entrada como la salida. Si escribimos

C:\> pies <dato.dat >mensajes.txt ↵

no veremos en pantalla nada, y se crea un fichero de texto en el directorio raíz llamado MENSAJES.TXT que almacena los mensajes generados por las dos sentencias **printf()**.

Ejercicios[2]

1. Escribe un programa que lea del teclado un código ASCII (entero comprendido entre 0 y 255) y presente en pantalla el carácter correspondiente.

2. Escribe un programa que lea del teclado un carácter cualquiera y presenta en pantalla su código ASCII en decimal, octal y hexadecimal.

3. Escribe un programa que lea del teclado dos números en coma flotante cuya parte entera se asume que no supera 3 dígitos, y muestre en pantalla su suma ajustada a la derecha. Por ejemplo, si los números son 23.6 y 187.54 el programa debe mostrar:

```
  23,60
 187,54
----------
 211,14
```

4. Escribe un programa que lea 2 números enteros de 3 dígitos e imprima su producto. Por ejemplo, si los números son 325 y 426 se presentará en el formato

```
   325
 x 426
------------
  1950
  650
 1300
------------
 138450
```

5. La fecha de Pascua corresponde al primer Domingo después de la primera Luna llena que sigue al equinoccio de Primavera, y se calcula con las siguientes expresiones:

```
A   =   resto de (año / 19)
B   =   resto de (año / 4)
C   =   resto de (año / 7)
D   =   resto de (19 * A + 24) / 30
```

[2] Utilizar en todas las salidas a pantalla funciones de control de la pantalla de texto o secuencias de escape ANSI (borrado de pantalla, posicionamiento del cursor, colores, etc.)

E = resto de (2 * B + 4 * C + 6 * D + 5) / 7
N = 22 + D + E

en el que N indica el número del día del mes de Marzo (Abril si N > 31, en cuyo caso el día es N - 31) correspondiente al Domingo de Pascua. Escribe un programa que acepte un año por teclado y muestre en pantalla la fecha del primer Domingo de Pascua con el formato dd/mm/aaaa.

6. Escribe un programa que lea del teclado los siguientes datos de 2 personas: nombre, edad, altura y peso. El programa enviará a impresora un listado con el formato:

NOMBRE	EDAD	ALTURA	PESO
xxxxxxxxxxxxxxxxxxxxxx	xx	xx,x	xxx,xx
xxxxxxxxxxxxxxxxxxxxxx	xx	xx,x	xxx,xx
Media:	xx,xx	xx,xx	xxx,xx

5
Sentencias de control

La estructura if

La estructura **if** adopta una de las dos formas siguientes:

if (*expresión***)** *sentencia;*

o bien

if (*expresión***)** *sentencia;*
else *sentencia;*

en donde *expresión* es una sentencia que se evalúa como verdadera (devuelve un valor no nulo) o falsa (devuelve cero). La palabra *sentencia* puede ser una sentencia simple terminada con un punto y coma, o un grupo de sentencias encerradas entre llaves {}. Algunos ejemplos válidos de sentencias **if** son:

- ```
  if (x > 0) puts ("POSITIVO");
  ```
- ```
  if (x) puts ("Verdadero");
  else puts ("Falso");
  ```
- ```
  if (c >= 'a' && c <= 'z') {
      puts ("La variable c almacena un carácter alfabético");
      puts ("El carácter es una letra minúscula");
  }
  ```
- ```
  if (num <= 40000) {
      printf ("\nOctal: %o", num);
      printf ("\nHexadecimal: %X", num); }
  else {
      puts ("El número es mas grande que 40000");
      printf ("Su valor es %u", num);
  }
  ```

Las estructuras **if** pueden anidarse sin más que tomar un mínimo de precauciones. En las sentencias

```
if (x)
    if (y) puts ("1");
else puts ("2");
```

el **else** está asociado a **if (y)**. C siempre asocia los **else** al **if** más cercano que no tenga ya un **else**. Para que en la sentencia anterior el **else** se asocie a **if (x)**, hay que colocar adecuadamente las llaves {}.

```
if (x) {
    if (y) puts ("1"); }
else puts ("2");
```

En C también se dispone de la estructura

if (*condición1***)** *sentencia1***;**
else if (*condición2***)** *sentencia2***;**
else if (*condición3***)** *sentencia3***;**
...
else *sentenciaN***;**

que va evaluando, sucesivamente, *condición1*, *condición2*, ... Cuando encuentra una cierta, se ejecuta la sentencia correspondiente y se finaliza el **if**. Si ninguna es cierta se ejecuta la sentencia que acompaña al **else** (que no es obligatorio). Como siempre, *sentencia1*, *sentencia2*, ..., pueden ser una sola sentencia o un grupo de ellas, en cuyo caso se encierran entre llaves {}.

El siguiente programa usa esta estructura para determinar si un número es positivo, negativo o cero.

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

void main ()
{
    int n;

    clrscr ();
    printf ("Teclee un número entero: ");
    scanf ("%d", &n);

    if (n > 0) puts ("Positivo");
    else if (n < 0) puts ("Negaivo");
    else puts ("Cero");
}
```

La estructura switch

La estructura **switch** inspecciona una variable y la va comparando con una lista de constantes. Cuando encuentra una coincidencia, se ejecuta la sentencia o grupo de sentencias asociado. La forma de **switch** es

switch (*variable***) {**

```
    case cte1:   sentencia;
                 break;
    case cte2:   sentencia;
                 break;
    ...
    ...
    default:     sentencia;
}
```

donde *variable* es una variable o cualquier expresión que devuelva un valor. La sentencia **switch** compara la variable con *cte1*, *cte2*, ..., y si encuentra una coincidencia, ejecuta la sentencia correspondiente. Por *sentencia* debemos entender tanto una sentencia simple como un grupo de sentencias (que, en este caso, no se encierran entre llaves). Si no se encuentra ninguna coincidencia se ejecuta la sección **default** (que no es obligatoria).

El siguiente segmento de programa muestra cómo se utiliza la sentencia **switch**.

```
char c;
...
...
c = getche ();
switch (c) {
    case 'a':  funcion_a ();
               break;
    case 'b':  funcion_b ();
               break;
    default:   puts ("No se ha pulsado ni a ni b");
}
```

Fijémonos que la sentencia **switch** busca coincidencias exactas, por lo que no es una alternativa a programas como el de la página anterior, ya que NO ESTÁ PERMITIDO imponer condiciones de desigualdad. No es correcto, por tanto

```
int n;
...
...
switch (n) {
    case > 0:  puts ("Positivo");
               break;
    case < 0:  puts ("Negativo");
               break;
    default:   puts ("Cero");
}
```

La sentencia **break** es opcional. Cuando se encuentra, provoca la salida de **switch**. En caso contrario continua la siguiente secuencia **case** o **default** aunque no se cumpla la condición. Para aclarar esto, tomemos el siguiente ejemplo:

```
int c;
```

```
...
...
scanf ("%d", &c);
switch (c) {
    case 1:
    case 2:   Funcion2 ();
    case 3:   Funcion3 ();
              break;
    case 4:   Funcion4_1 ();
              Funcion4_2 ();
              break;
    case 5:   Funcion_5 ();
    default:  FuncionX ();
}
```

La siguiente tabla indica qué función se ejecuta dependiendo del valor de **c**.

Si se pulsa	***Se ejecuta***
1	*Funcion2() y Funcion3()*
2	*Funcion2() y Funcion3()*
3	*Funcion3()*
4	*Funcion4_1() y Funcion4_2()*
5	*Funcion5() y FuncionX()*
cualquier otra cosa	*FuncionX()*

La sentencia **default** es opcional. Cuando no está no se ejecuta ninguna acción al fallar todas las coincidencias. Simplemente se abandona el **switch** sin hacer nada. Si hay sentencia **default**, el bloque de sentencias asociado se ejecuta cuando fallan todas las comparaciones o no hay un **break** anterior que lo impida. Las sentencias **switch** pueden anidarse, con lo que se permiten estructuras del tipo:

```
switch (m) {
    case 1:   Funcion1 ();
              break;

    case 2:   switch (n) {
                  case 21:   Funcion21 ();
                             break;

                  default:   switch (p) {
                                 case 31:   Funcion31 ();
                                            break;
                                 case 31:   Funcion32 ();
                             }
              }
              break;
    default:  FuncionX ();
}
```

Bucles

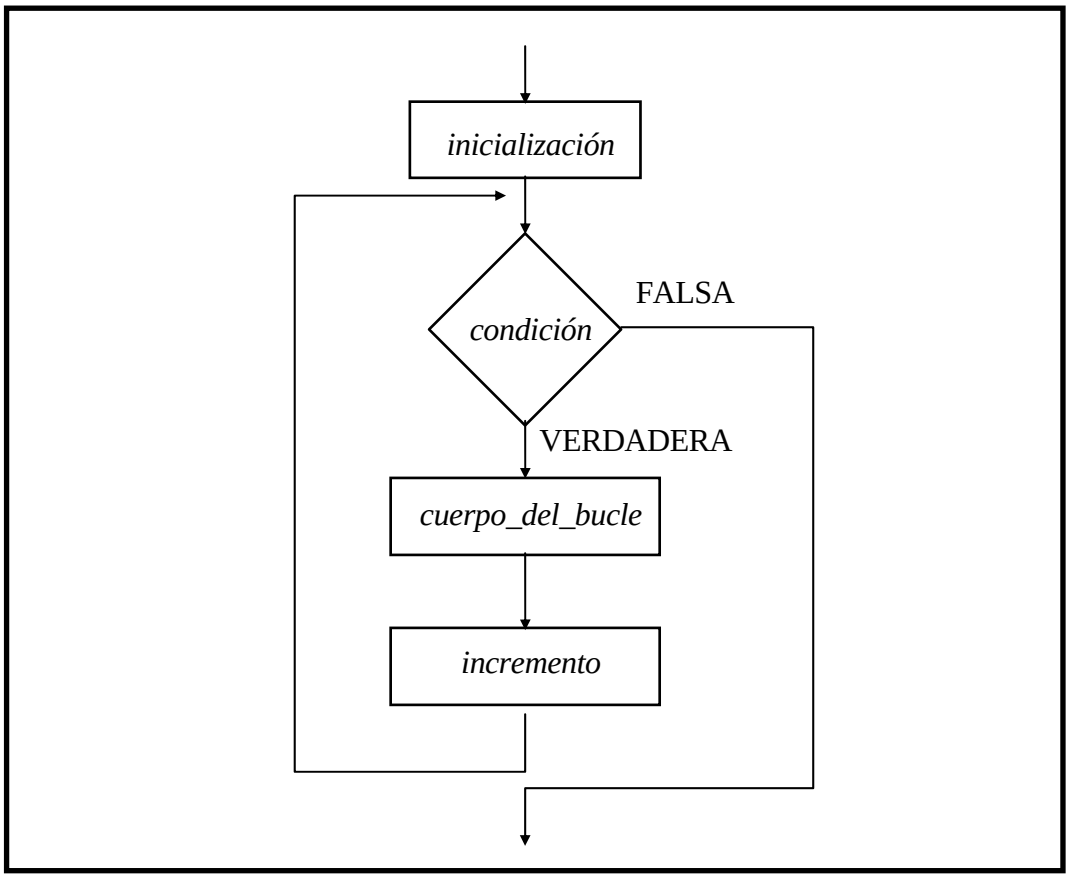
Existen en C tres tipos de bucles: **for**, **while** y **do/while**.

Bucles for

El bucle **for** es muy potente y flexible. Además de permitir las mismas operaciones que cualquier **for** de otros lenguajes, tiene características que lo diferencian claramente de ellos. En su formato tradicional este bucle tiene la forma

for (*inicialización***;** *condición***;** *incremento***)** *cuerpo_del_bucle***;**

Vemos que **for** tiene tres secciones: *inicialización*, en dónde se da un valor inicial a una variable de control del bucle; *condición*, que es una expresión que devuelve un valor verdadero o falso, y hace que el bucle se repita mientras sea cierta; e *incremento*, en dónde se determina la cuantía del incremento o decremento de la variable de control. Las tres secciones están separadas por punto y coma. El *cuerpo del bucle* puede estar formado por una o por varias sentencias. En este último caso deben encerrarse entre llaves {}. El flujo de sentencias en este bucle es el siguiente:



Fijémonos que el **for** se sigue ejecutando MIENTRAS la *condición* sea verdadera.

Veamos un par de ejemplos. En la siguiente secuencia se muestran en pantalla los números del 1 al 10 y sus cuadrados.

```
register int i;
...
...
for (i = 1; i <= 10; i++) {
    printf ("\nValor de i: %d", i);
    printf ("\nValor de i2: %d", i * i);
}
```

En esta, se muestran en pantalla las letras mayúsculas de la A a la Z.

```
char letra;
...
...
for (letra = 'A'; letra <= 'Z'; letra++) printf ("\n%c", letra);
```

Puede ponerse un incremento/decremento diferente de 1. El siguiente ejemplo muestra en pantalla los números pares comprendidos entre 1 y 100, descendentemente:

```
register int i;
...
...
for (i = 100; i >= 1; i = i - 2) printf ("\n%d", i);
```

Estudiaremos ahora algunas formas de **for** que se apartan del uso tradicional.

Es posible tener más de una variable de control del bucle: En el bucle **for** las secciones de *inicialización* e *incremento* pueden tener, a su vez, subsecciones, en cuyo caso van separadas por el operador secuencial (**,**). Un ejemplo es

```
register int i, j;
...
...
for (i = 0, j = 1; i + j < N; i++, j++) printf ("\n%d", i + j);
```

que visualiza los N primeros números impares. No debe confundirse esta sentencia con un anidamiento de bucles **for**. Un anidamiento tiene el siguiente aspecto:

```
register int i, j;
...
...
for (i = 0; i <= 100; i++) {
    ...
    for (j = 0; j <= 100; j++) {
            cuerpo_del_bucle;
    }
    ...
}
```

La condición de salida del bucle no tiene por qué referirse a la variable de control: Esto queda ilustrado en el siguiente ejemplo:

```
char a;
register int i;
...
...
for (i = 1; a != 's'; i++) {
    printf ("\n%d", i);
    a = getch ();
}
```

En este ejemplo se van mostrando en pantalla los números 1, 2, ... hasta que se teclee el carácter **s**.

El bucle for puede no tener cuerpo: Esta característica permite crear retardos en un programa. El bucle

```
register int i;
...
...
for (i = 1; i <= 100; i++);
```

provoca un retardo de 100 ciclos.

El bucle for puede tener vacía cualquier sección: En un bucle **for** puede faltar una, dos o las tres secciones. Por ejemplo, es correcto escribir

```
register int i;
...
...
for (i = 0; i != 10; ) {                    /* Falta la 3ª sección (incremento) */
    scanf ("%d", &i);
    printf  ("\n%d", i);
}
```

que va mostrando en pantalla los valores que se tecleen, finalizando al teclear el número 10 (que también se visualiza).

También es correcto un bucle como

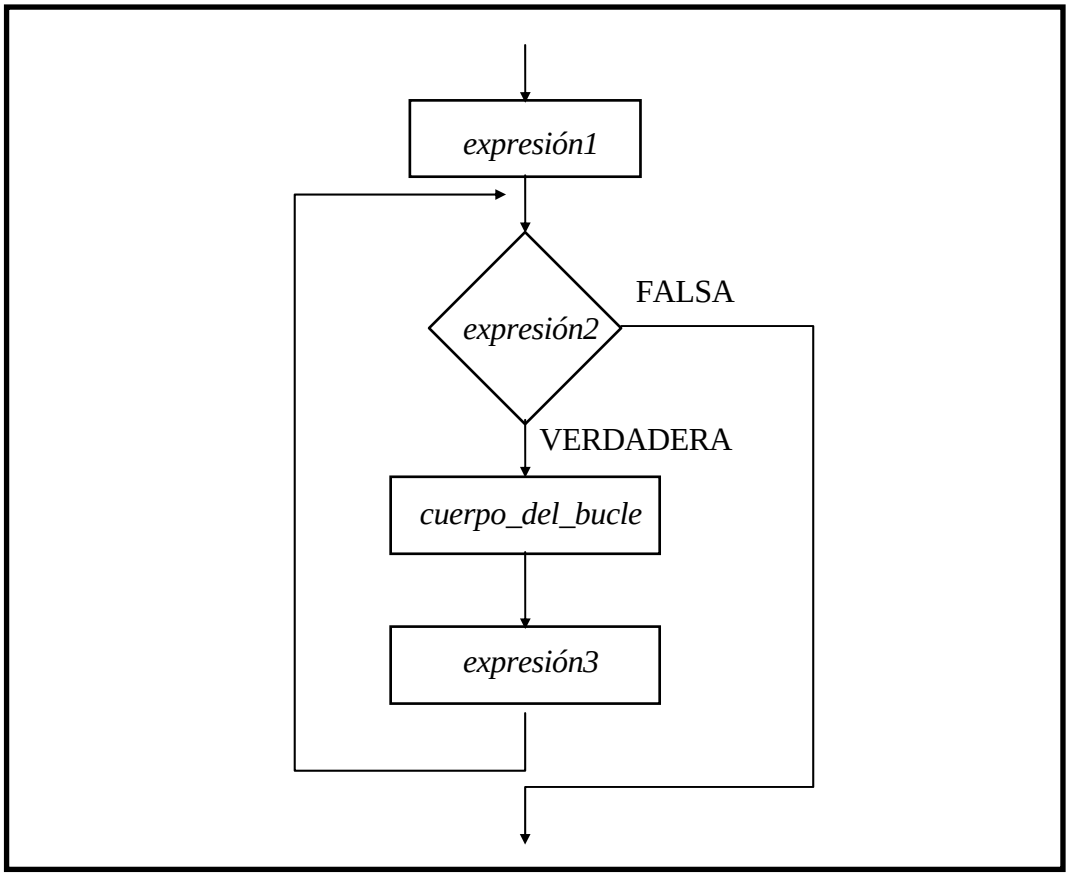
```
for ( ; ; ) {
    cuerpo_del_bucle;
}
```

que es un bucle sin fin. Estudiaremos más adelante, en este mismo capítulo, cómo abandonar un bucle infinito.

Cualquier expresión válida en C puede estar en cualquier sección de un bucle for: La forma del bucle **for** no tiene que ajustarse necesariamente a la mostrada en la página 67. En realidad la forma correcta es:

for (*expresión1***;** *expresión2***;** *expresión3***)** *cuerpo del bucle***;**

siendo *expresiónN* cualquier expresión válida C. Podemos decir que, en general, el flujo de sentencias de un bucle **for** es:



Aclararemos esto con el programa siguiente:

```
#include <stdio.h>
void main ()
{
    int t;
    for (mensaje (); t = lee_numero (); cuadrado (t));
}
mensaje ()
{
    printf ("\nTeclee un número (0 finaliza): ");
}
lee_numero ()
{
    int n;
    scanf ("%d", &n);
    return n;
}
cuadrado (int x)
{
    printf ("\nEl cuadrado es %d", x * x);
}
```

Vamos a fijarnos en el bucle **for** de la función **main()** y explicarlo mediante el diagrama de flujo de la página anterior.

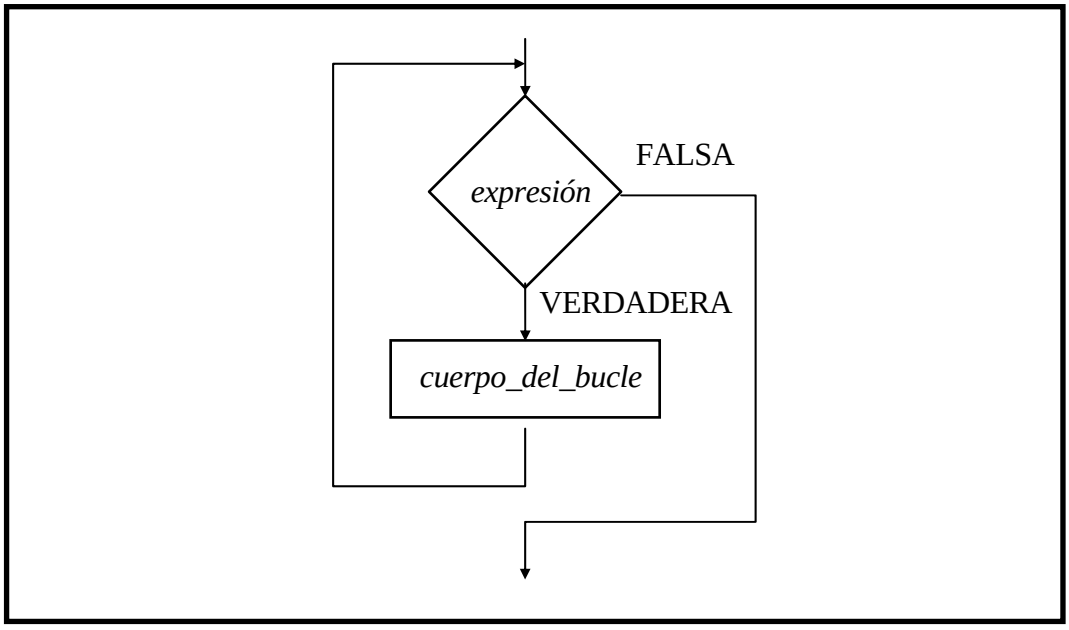
1. Se ejecuta la función **mensaje()** que muestra la cadena "Teclee un número (0 finaliza): ". (IMPORTANTE: Esta función sólo se ejecuta esta vez).
2. Se evalúa la expresión **t = lee_numero()**, es decir, **lee_numero()** captura un número entero del teclado y lo devuelve almacenándolo en **t**.
   2.1 Si la expresión **t = lee_numero ()** devuelve FALSO, es decir, si se ha tecleado 0, finaliza el bucle.
   2.2 En caso contrario continúa el bucle.
3. Se ejecuta el cuerpo del bucle. En este caso, dado que **for** finaliza con punto y coma, no hay cuerpo del bucle.
4. Se ejecuta la función **cuadrado(t)** que visualiza el cuadrado del número tecleado.
5. Se vuelve al paso 2.

El bucle while

Tiene la forma

while (*expresión***)** *cuerpo_del_bucle***;**

siendo *expresión* cualquier expresión C válida. El *cuerpo_del_bucle*, puede estar formado por una sentencia sencilla o por un bloque de sentencias, en cuyo caso, se encierran entre llaves {}. El flujo de sentencias es



Por lo tanto, en el bucle **while** el *cuerpo_del_bucle* se repite mientras *expresión* se evalúe como cierta. Veamos algunos ejemplos.

```
char c;
...
...
while (c != 's' && c != 'n') c = getche ();
```

En esta sentencia se solicita un carácter del teclado mientras no se teclee el carácter **n** ni el carácter **s**. Cuando se teclea alguno de estos caracteres, se almacena en **c** y se abandona el bucle.

El siguiente ejemplo es un caso de bucle **while** sin cuerpo.

```
while (getch () != 13);
```

El programa está detenido en esta sentencia hasta que se teclee ↵ (código ASCII 13).

El siguiente programa utiliza un bucle **while** para solicitar del usuario que adivine un número.

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

void main ()
{
    int num;
    int n = 0;

    randomize ();                    // Las funciones randomize() y random()
permiten
    num = random (20) + 1;           // generar números aleatorios

    while (n != num) {
        printf ("\nTeclee un número entre 1 y 20: ");
        scanf ("%d", &n);

        if (n == num) puts ("ACERTASTE);
        else if (n < num) puts ("TU NÚMERO ES MENOR");
        else puts ("TU NÚMERO ES MAYOR");
    }
}
```

Respecto del bucle **while** es conveniente tener presente lo siguiente:

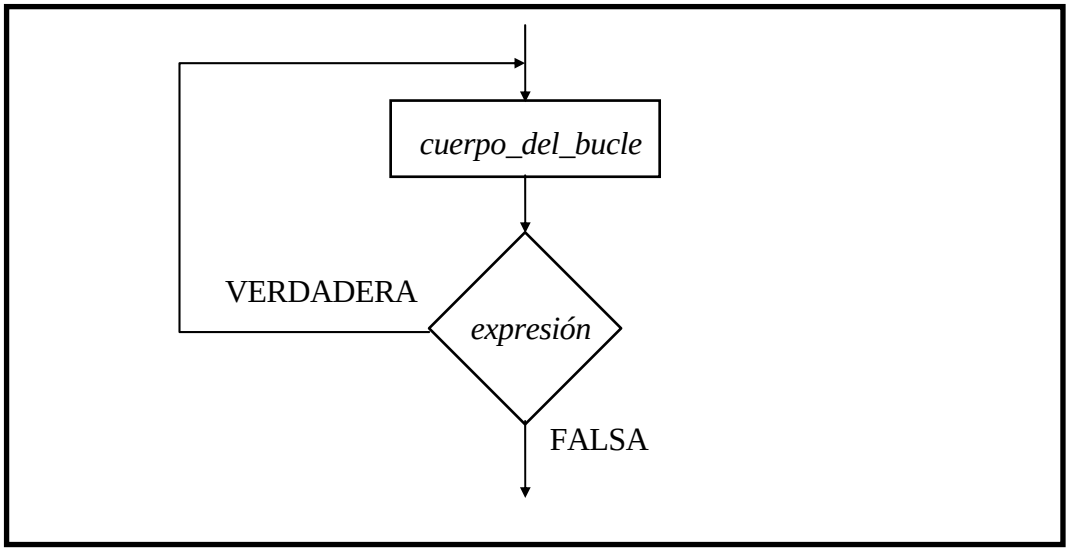
- El cuerpo del bucle no se ejecutará NUNCA si la primera vez no se cumple la condición.
- El bucle puede ser INFINITO si no se modifican adecuadamente las variables de la condición dentro del bucle.

El bucle do/while

Tiene la forma

do
    cuerpo_del_bucle;
while (*expresión***);**

siendo *sentencia* una sentencia simple o un grupo de sentencias encerradas entre llaves {}, y *expresión* cualquier expresión válida C. El flujo de ejecución es el siguiente:



Por lo tanto, en un bucle **do/while** el *cuerpo_del_bucle* se ejecuta al menos una vez, incluso aunque *expresión* se evalúe como falsa, puesto que la evaluación se hace al final, justo lo contrario del bucle **while**, en el que la evaluación de *expresión* se hace al principio.

En el siguiente ejemplo se solicita un carácter del teclado hasta que se pulse cualquiera de los caracteres 'S' o 'N'.

```
#include <stdio.h>

void main ()
{
    char tecla;

    do {
        printf ("\nPulse S o N: ");
        tecla = getch ();
    } while (tecla != 'S' && tecla != 'N');
}
```

Sentencia break

Es la misma sentencia que hemos visto para finalizar los **case** de la sentencia **switch**. Pero además permite forzar la salida inmediata de un bucle (**for**, **while** o **do/while**) en cualquier momento, ignorando el resto de sentencias. Veamos un ejemplo:

```
#include <stdio.h>

void main ()
{
    int n;

    for ( ; ; ) {
        printf ("\nTeclee un número: ");
        scanf ("%d", &n);
        if (!n) break;
        printf ("\nEl cuadrado es %d", n * n);
    }
}
```

En este ejemplo, el bucle **for** se ejecutaría sin fin a no ser por la sentencia

```
if (!n) break;          //Es lo mismo que if (n == 0) break;
```

Se van solicitando números por teclado y visualizando sus cuadrados hasta que se teclee un 0 (**!n** se evaluaría como cierto), en cuyo caso se ejecuta la sentencia **break**, que provoca la salida inmediata del bucle sin que se ejecute la sentencia **printf** del final.

Sentencia continue

Esta sentencia se utiliza en los bucles **for**, **while** y **do/while**. Cuando se ejecuta fuerza un nuevo ciclo del bucle, saltándose cualquier sentencia posterior. Por ejemplo, el bucle

```
int i, n;
...
...
for (i = 1; i <= 100; i++) {
    n = i / 2;
    if (i == 2 * n) continue;
    printf ("\n%d", i);
}
```

muestra en pantalla sólo los números impares, puesto que para los números pares la expresión **i == 2 * n** se evalúa como cierta, ejecutándose la sentencia **continue** que fuerza de inmediato un nuevo ciclo del bucle.

El siguiente programa muestra como actúa la sentencia **continue** en un bucle **do/while**.

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

void main ()
{
    int n;
    int positivos = 0;

    clrscr ();
    do {
        printf ("\nTeclea un número (-99 finaliza): ");
        scanf ("%d", &n);
        if (n <= 0) continue;
        positivos++;
    } while (n != -99);

    printf ("\nHas tecleado %d números positivos", positivos);
}
```

La sentencia **positivos++** sólo se ejecuta cuando **n** es un número positivo. Si **n** es negativo o vale 0, se ejecuta **continue** que fuerza una nueva evaluación de la condición de salida del bucle.

Etiquetas y sentencia goto

En C existe la sentencia de salto incondicional **goto** que fuerza un salto del programa a una línea identificada por una etiqueta. La etiqueta se define con un identificador válido C, seguido por dos puntos (:).

```
                goto etiqueta;
                ...
                ...
etiqueta:       sentencia;
                ...
                ...
```

La *etiqueta* puede estar antes o después del **goto**, pero siempre en la misma función.

Realmente, en lenguajes con suficientes estructuras de control (como C) no suelen presentarse situaciones que hagan necesaria la sentencia **goto**. Sin embargo, en alguna ocasión puede ser conveniente, bien porque la velocidad de proceso es importante (un salto con **goto** es más rápido que otro tipo de controles de bifurcación), o bien porque su uso clarifica el código. El caso más habitual es la salida de varios niveles de anidamiento.

```
        for (...) {
            while (...) {
                for (...) {
                    ...
                    ...
                    if (...) goto salir;
                    ...
                    ...
                }
            }
        }
salir:  ...
        ...
```

En este ejemplo, la única alternativa a **goto** sería la realización de varias comprobaciones en cada bucle que forzase sentencias **break**, lo cual haría más ilegible el código.

Función exit()

Esta función permite la finalización del programa en cualquier punto del mismo. Devuelve el control al sistema operativo o a otro proceso padre, enviando un valor de retorno. Necesita la inclusión del archivo de cabecera **process.h**, por medio de una sentencia **#include**.

Si en un programa escribimos

```
if (condición) exit (0);
```

se produce el final del programa cuando **condición** es cierta, en cuyo caso se devuelve el valor 0 al proceso padre.

El programa que se muestra a continuación (SINO.C) será ejecutado desde un archivo BAT (proceso padre). El programa SINO pasará un valor de retorno al proceso padre por medio de la función **exit()**. El proceso padre inspecciona este valor de retorno mediante ERRORLEVEL.

```
/* Programa SINO.C: Admite por teclado el carácter 's' o el carácter 'n'. En el primer caso ('s')
entrega un valor de retorno 1
En el segundo caso ('n') entrega un valor de retorno 0 */

#include <stdio.h>
#include <conio.h>
#include <process.h>

void main ()
{
    char letra;

    do
        letra = getch ();
    while (letra != 's' && letra != 'n');

    if (letra == 's') exit (1);
    exit (0);

    printf ("\nFin del programa");          //Esta sentencia no se ejecuta nunca
}
```

Compilamos y enlazamos este programa como SINO.EXE y lo incluimos en un archivo por lotes como el siguiente:

```
@ECHO OFF
ECHO Pulsa S ó N
SINO
IF ERRORLEVEL == 1 GOTO SI
GOTO NO
:SI
ECHO Pulsaste SÍ
GOTO FIN
:NO
ECHO Pulsaste NO
:FIN
@ECHO ON
```

Este archivo visualiza el mensaje "Pulsaste SÍ" cuando en SINO se teclea el carácter **s**, y visualiza el mensaje "Pulsaste NO" si en SINO se teclea el carácter **n**. En ningún caso se visualiza el mensaje "Fin del programa" de la última línea de SINO.C

Ejercicios

1. Escribe un programa que asigne una calificación literal a un estudiante, basada en la siguiente tabla de puntuación:

8.5 a 10	Sobresaliente
7 a 8.5	Notable
6 a 7	Bien
5 a 6	Suficiente
3.5 a 5	Insuficiente
0 a 3.5	Muy deficiente

El programa capturará un valor numérico del teclado y visualizará la calificación correspondiente. Los suspensos se mostrarán en amarillo parpadeando sobre fondo rojo, el sobresaliente en amarillo sobre fondo azul, y el resto en negro sobre fondo verde.

2. Escribe un programa para determinar si un atleta es seleccionado para correr una maratón. Para seleccionar a un corredor, debe haber terminado una maratón anterior en un determinado tiempo. Los tiempos de calificación son 150 minutos para hombres menores de 40 años, 175 minutos para hombres mayores de 40 años, y 180 minutos para mujeres. Los datos a introducir son: sexo (H/M), edad y tiempo efectuado en su anterior maratón. El programa visualizará el mensaje "Seleccionado" o "No seleccionado".

3. Los empleados de una fábrica trabajan en dos turnos: diurno y nocturno. Se desea calcular el jornal diario de acuerdo al siguiente baremo:

a) Las horas diurnas se pagan a 1000 pesetas.
b) Las horas nocturnas se pagan a 1600 pesetas.
c) Caso de ser domingo, la tarifa se incrementará en 400 pesetas el turno diurno y en 600 el nocturno.

4. Escribe un programa que calcule e imprima la suma de los pares y de los impares comprendidos entre dos valores **A** y **B** que se introducen por teclado (**A < B**).

5. Escribe un programa que calcule $\mathbf{x^n}$, siendo **x** y **n** dos números enteros que se introducen por teclado.

6. Escribe un programa que calcule el factorial de un número entero positivo que se introduce por teclado.

7. Escribe un programa que encuentre el primer valor **N** para el que la suma

$$1 + 2 + 3 + ... + N$$

excede a un valor **M** que se introduce por teclado.

8. Escribe un programa que calcule el primer elemento de la serie de Fibonacci que sea mayor o igual que un valor introducido por teclado. La serie de Fibonacci se define mediante:

$$a_0 = a_1 = 1 \qquad\qquad a_n = a_{n-1} + a_{n-2}$$

9. El valor de π puede calcularse mediante la serie

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 ...)$$

Escribe un programa que calcule el valor de π. Para elegir el número de términos de la serie adopta el criterio de que la diferencia absoluta entre el valor real de π (3.141592) y el valor calculado sea menor que 10^{-3}. Crea una función que devuelva el valor absoluto de una expresión.

6
Funciones

Introducción

Una de las formas más adecuadas de resolver un problema de programación consiste en descomponerlo en subproblemas. A cada uno de ellos se le asocia una función que lo resuelve, de tal modo que la solución del problema se obtiene por medio de llamadas a funciones. A su vez, cada función puede descomponerse en subfunciones que realicen tareas más elementales, intentando conseguir que cada función realice una y sólo una tarea.

En Lenguaje C una función se define de la siguiente forma:

```
tipo NombreFunción (parámetros formales)
{
    ...
    cuerpo de la función
    ...
}
```

El **tipo** es el tipo de dato que devuelve la función por medio de la sentencia **return** cuyo estudio se adelantó en la página 8. Cuando no se especifica un tipo, se asume que el tipo devuelto es **int**. El **NombreFunción** es un identificador válido en C. Es el nombre mediante el cual se invocará a la función desde **main()** o desde otras funciones. Los **parámetros formales** son las variables locales mediante las cuales la función recibe datos cuando se le invoca. Deben ir encerrados entre paréntesis. Incluso si no hay parámetros formales los paréntesis deben aparecer. La siguiente función devuelve el mayor de dos números enteros:

```
int mayor (int x, int y)
{
    int max;

    if (x > y) max = x;
    else max = y;

    return max;
}
```

Al manejar funciones en C debemos tener en cuenta que a una función sólo se puede acceder por medio de una llamada. Nunca se puede saltar de una función a otra mediante una sentencia **goto**. Tampoco está permitido declarar una función dentro de otra.

En C, a diferencia de otros lenguajes, como Pascal, no existe distinción entre funciones y procedimientos. Por ello, una función puede estar o no dentro de una expresión. Si la función devuelve un valor, son válidas sentencias del tipo:

```
a = b * funcion (c, d);
```

Pero también podemos encontrar funciones solas en una línea de programa,

```
funcion (a, b, c);
```

incluso aunque la función devuelva algún valor. En ese caso la función realizaría la tarea encomendada y el valor devuelto se perdería. Esto no provoca ningún error.

Por último, hemos de tener en cuenta que, aunque una función puede formar parte de una expresión compleja, nunca se le puede asignar un valor. Por tanto, es incorrecto escribir sentencias como

```
funcion () = variable;
```

que sería tan improcedente como escribir

```
5 = variable;
```

El único método para que una función reciba valores es por medio de los parámetros formales.

Argumentos de funciones

Los parámetros formales son variables locales que se crean al comenzar la función y se destruyen al salir de ella. Al hacer una llamada a la función los parámetros formales deben coincidir en tipo y en número con las variables utilizadas en la llamada, a las que denominaremos **argumentos de la función**. Si no coinciden, puede no detectarse el error. Esto se evita usando los llamados *prototipos de funciones* que estudiaremos más adelante, en este capítulo.

Los argumentos de una función pueden ser:

- valores (llamadas por valor)
- direcciones (llamadas por referencia)

Llamadas por valor

En las llamadas por valor se hace una copia del valor del argumento en el parámetro formal. La función opera internamente con estos últimos. Como las variables locales a una función (y los parámetros formales lo son) se crean al entrar a la función y se destruyen al salir de ella, cualquier cambio realizado por la función en los parámetros formales no tiene ningún efecto sobre los argumentos. Aclaremos esto con un ejemplo.

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambia (a, b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambia (int x, int y)
{
    int aux;

    aux = x;
    x = y;
    y = aux;
}
```

La salida de este programa es:

Valor de a: 3 - Valor de b: 4

es decir, NO intercambia los valores de las variables **a** y **b**. Cuando se hace la llamada

```
intercambia (a, b);
```

se copia el valor de **a** en **x**, y el de **b** en **y**. La función trabaja con **x** e **y**, que se destruyen al finalizar, sin que se produzca ningún proceso de copia a la inversa, es decir, de **x** e **y** hacia **a** y **b**.

Llamadas por referencia

En este tipo de llamadas los argumentos contienen direcciones de variables. Dentro de la función la dirección se utiliza para acceder al argumento real. En las llamadas por referencia cualquier cambio en la función tiene efecto sobre la variable cuya dirección se pasó en el argumento. Esto es así porque se trabaja con la propia dirección de memoria, que es única, y no hay un proceso de creación/destrucción de esa dirección.

Aunque en C todas las llamadas a funciones se hacen por valor, pueden simularse llamadas por referencia utilizando los operadores **&** (dirección) y ***** (en la dirección). Mediante **&** podemos pasar direcciones de variables en lugar de valores, y trabajar internamente en la función con los contenidos, mediante el operador *****.

El siguiente programa muestra cómo se puede conseguir el intercambio de contenido en dos variables, mediante una función.

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambio (&a, &b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambio (int *x, int *y)
{
    int aux;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

Este programa SÍ intercambia los valores de **a** y **b**, y muestra el mensaje

Valor de a: 4 - Valor de b: 3

Además, a diferencia del de la página 83, trabaja con direcciones. Como veremos en el siguiente capítulo, en la declaración

```
void intercambio (int *x, int *y);
```

los parámetros **x** e **y** son unas variables especiales, denominadas **punteros**, que almacenan direcciones de memoria. En este caso, almacenan direcciones de enteros.

Valores de retorno de una función

Como ya vimos, de una función se puede salir de dos formas distintas: bien porque se "encuentra" la llave **}** que cierra la función, bien porque se ejecuta una sentencia **return**. En este último caso la forma de la sentencia puede ser cualquiera de las siguientes:

return *constante***;**
return *variable***;**
return *expresión***;**

donde *expresión* puede, por claridad, ir entre paréntesis. El valor devuelto por **return** debe ser compatible con el tipo declarado para la función.

Como vimos al principio del capítulo, si a una función no se le asigna ningún tipo devuelve un valor de tipo **int**, por lo que debe tener una sentencia **return** de salida que devuelva un entero. Sin embargo, la ausencia de esta sentencia sólo provoca un **Warning** en la compilación. De cualquier manera, si una función no devuelve ningún valor (no hay sentencia **return**) lo correcto es declararla del tipo **void**.

Tampoco es necesario declarar funciones de tipo **char**, pues C hace una conversión limpia entre los tipos **char** e **int**.

Prototipos de funciones

Cuando una función devuelve un tipo no entero, antes de utilizarla, hay que "hacérselo saber" al resto del programa usando los **prototipos de funciones**. Un prototipo de función es algo que le indica al compilador que existe una función y cuál es la forma correcta de llamarla. Es una especie de "plantilla" de la función, con dos cometidos:

- identificar el tipo devuelto por la función.
- identificar el tipo y número de argumentos que utiliza la función.

La forma de definir un prototipo de función es:

tipo NombreFunción (lista de tipos);

Entre paréntesis se ponen los tipos de los argumentos separados por comas (no es necesario poner el nombre) en el mismo orden en el que deben estar en la llamada. Así, un prototipo como el siguiente

```
float Ejemplo (int, char);
```

indica que el programa va a utilizar una función llamada **Ejemplo** que devuelve, mediante una sentencia **return**, un valor de tipo **float**, y recibe dos valores en sus argumentos: el primero es un valor entero y el segundo un carácter. También es un prototipo válido

```
float Ejemplo (int x, char y);
```

En un programa, los prototipos de funciones se sitúan antes de la función **main()**. A continuación se muestra un programa que define un prototipo de función.

```
#include <stdio.h>

float multiplica (float, float);          /* Prototipo de la función multiplica() */

main ()
{
    float a, b, total;

    printf ("\nTeclee dos números: ");
    scanf ("%f %f", &a, &b);
    total = multiplica (a, b);
    printf ("\nEl producto de ambos es %f", total);
}

float multiplica (float m, float n)
{
    return m * n;
}
```

En este programa si en lugar de la línea

```
total = multiplica (a, b);
```

escribiésemos

```
total =  multiplica (a);
```

el compilador, alertado por el prototipo, informaría del error.

Tanto en los prototipos como en las declaraciones de funciones, las referencias a cadenas de caracteres se realizan mediante la expresión **char ***. El significado de esta expresión se entenderá más claramente en el próximo capítulo. Así, el prototipo de una función que devuelve una cadena de caracteres y tiene como único argumento una cadena de caracteres es

```
char *Funcion (char *);
```

Cuando una función tiene un número variable de argumentos, se especifica por medio de 3 puntos suspensivos. Este es el caso de la función **printf()** cuyo prototipo es:

```
int printf (const char *, ...);
```

La palabra reservada **const** aquí significa que la cadena referida con **char *** puede ser, bien una variable, como en

```
printf (cadena);
```

o bien una constante, como en

```
printf ("Cadena constante");
```

Para indicar wque una función no tiene argumentos se pone la palabra reservada **void** entre los paréntesis. Del mismo modo, se indica con **void** que una función no devuelve ningún valor (no hay sentencia return). Por ejemplo, la función de prototipo

```
void funcion (void);
```

no devuelve nada ni recibe valores.

Cuando deseamos que una función devuelva una cadena de caracteres, el prototipo se escribe

```
char *funcion (lista de parámetros);
```

En el próximo capítulo estudiaremos más profundamente el significado de la expresión **char ***.

Todos los programas que utilicen funciones de la biblioteca estándar deben incluir sus prototipos. Estos prototipos y otras definiciones usadas por las funciones están en los archivos de cabecera como **stdio.h**, por lo que el programa deberá tener las sentencias **#include** necesarias.

Recursividad

La recursividad es un concepto de lógica matemática que consiste en definir una función en términos de sí misma. Por ejemplo, la definición del factorial de un número es una definición recursiva:

$$n! = n \cdot (n - 1)!$$
$$0! = 1$$

En programación una función es recursiva si puede llamarse a sí misma. No todos los lenguajes permiten la recursividad. Una definición recursiva de una función requiere dos partes:

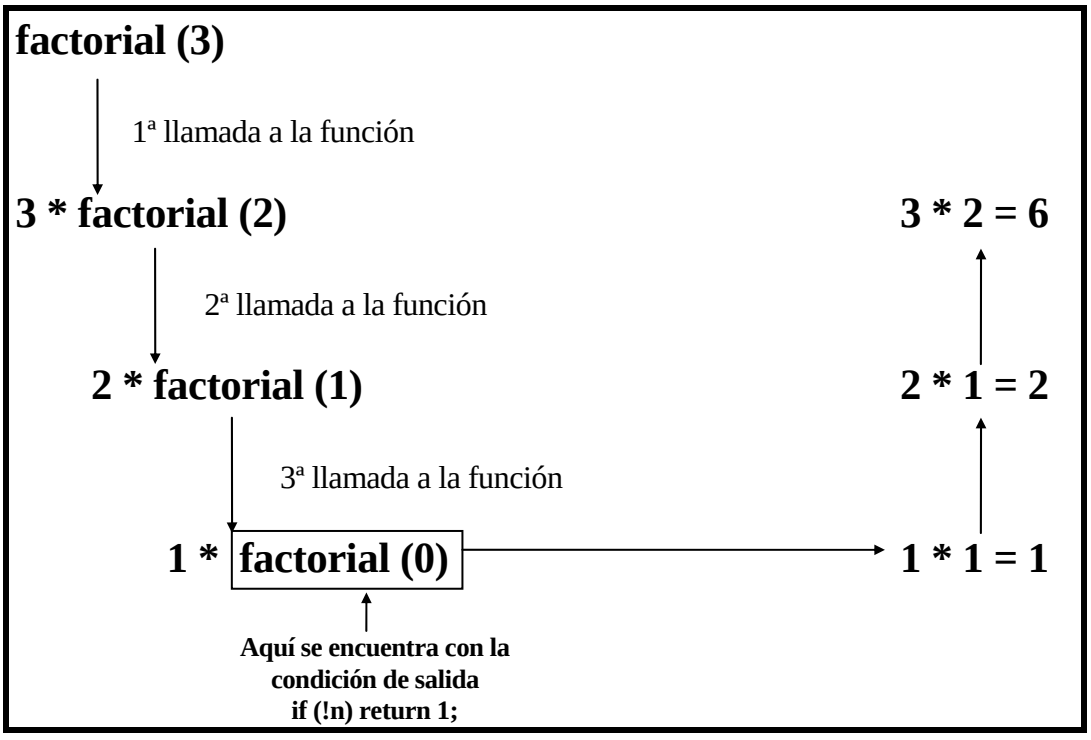
- Una **condición de salida** o **cláusula de escape**.
- Un paso en el que los valores restantes de la función se definen en base a valores definidos anteriormente.

La recursividad es una alternativa a la iteración. Sin embargo, en general, una solución recursiva es menos eficiente, en términos de tiempo, que una solución iterativa. Además, las soluciones recursivas utilizan mucha memoria de pila, pues cada llamada a la función hace que se copie en la pila (cada vez) todo el juego de

argumentos. Esto puede crear problemas cuando se llega a niveles profundos de recursión. Cómo decidir cuándo un problema se resuelve recursivamente o mediante iteraciones depende del problema en sí. La solución recursiva suele ser la más sencilla, y debe elegirse si no hay grandes requerimientos de velocidad del proceso ni problemas con el tamaño de la memoria. En otro caso debemos elegir la solución iterativa, mucho más rápida, aunque a menudo más compleja. La siguiente función resuelve recursivamente el cálculo del factorial

```
long factorial (int n)
{
    if (!n) return 1;
    return n * factorial (n - 1);
}
```

Representaremos ahora en un esquema el proceso de recursión al llamar, por ejemplo, a **factorial(3)**.



Se puede comparar este proceso con el que genera la solución iterativa:

```
long factorial (int n)
{
    register int i;
    long total = 1;

    for (i = 1; i <= n; i++) total *= i;
    return total;
}
```

En el primer caso se hacen varias llamadas a la función, situando en la pila los valores correspondientes y "dejando pendientes" los cálculos intermedios, hasta que

se llega a la condición de salida. En ese momento comienza el proceso inverso, sacando de la pila los datos previamente almacenados y solucionando los cálculos que "quedaron sin resolver". Obviamente esta solución es más lenta que la iterativa en la que el resultado se obtiene con una sola llamada a la función y con menos requerimientos de memoria.

Sin embargo, mientras que en el caso del cálculo del factorial de un número parece más aconsejable la solución iterativa, hay otros ejemplos en lo que ocurre lo contrario. Por ejemplo, la función de Ackerman se define del siguiente modo:

$$Ack(s, t) = \begin{cases} t + 1 \\ Ack(s - 1, 1) \\ Ack(s - 1, Ack(s, t - 1)) \end{cases} \text{ si } \begin{cases} s = 0 \\ s \neq 0 \text{ y } t = 0 \\ s \neq 0 \text{ y } t \neq 0 \end{cases}$$

El diseño recursivo de esta función consiste, simplemente, en aplicar la definición

```
int Ack (int s, int t)
{
    if (!s) return t + 1;
    else if (!t) return Ack (s - 1, 1);
    else return Ack (s - 1, Ack (s, t - 1));
}
```

La solución no recursiva de la función de Ackerman es mucho más complicada.

La biblioteca de funciones

Hay una serie de funciones y macros definidas por el estándar ANSI que realizan tareas que facilitan enormemente la labor del programador. Además, cada fabricante de software incluye sus propias funciones y macros, generalmente relacionadas con el mejor aprovechamiento de los ordenadores personales y del MS-DOS. Respecto a esto, debe tenerse en cuenta que la compatibilidad del código sólo está asegurado si se utilizan exclusivamente funciones pertenecientes al estándar ANSI. Como ya quedó dicho, los prototipos de estas funciones, así como declaraciones de variables, macros y tipos de datos utilizados por ellas, están definidos en los archivos de cabecera ***.h**. Por ello, es necesario incluirlos mediante sentencias **#include** cuando el programa hace uso de ellas.

La tabla de la página siguiente muestra los archivos de cabecera estándar usados por Turbo C++. En el Capítulo 13 se muestran algunas de las funciones de la biblioteca estándar.

LENGUAJE	***ARCHIVO DE***	***DESCRIPCIÓN***

	CABECERA	
	ALLOC.H	*Define funciones de asignación dinámica de memoria*
ANSI C	*ASSERT.H*	*Declara la macro de depuración* ***assert***
C++	*BCD.H*	*Define la clase* ***bcd***
	BIOS.H	*Define funciones utilizadas en rutinas de ROM-BIOS*
C++	*COMPLEX.H*	*Define las funciones matemáticas complejas*
C++	*CONIO.H*	*Define varias funciones utilizadas en las llamadas a rutinas de E/S por consola en DOS*
ANSI C	*CTYPE.H*	*Contiene información utilizada por las macros de conversión y clasificación de caracteres*
	DIR.H	*Contiene definiciones para trabajar con directorios.*
	DOS.H	*Declara constantes y da las declaraciones necesarias para llamadas específicas del 8086 y del DOS*
ANSI C	*ERRNO.H*	*Declara mnemónicos constantes para códigos de error*
	FCNTL.H	*Declara constantes simbólicas utilizadas en conexiones con la biblioteca de rutinas* ***open()***
ANSI.C	*FLOAT.H*	*Contiene parámetros para rutinas de coma flotante*
C++	*FSTREAM.H*	*Define los flujos de C++ que soportan E/S de archivos*
C++	*GENERIC.H*	*Contiene macros para declaraciones de clase genéricas*
C++	*GRAPHICS.H*	*Define prototipos para las funciones gráficas*
	IO.H	*Declaraciones de rutinas de E/S tipo UNIX*
C++	*IOMANIP.H*	*Define los gestores de flujos de E/S de C++ y contiene macros para creación de gestores de parámetros*
C++	*IOSTREAM.H*	*Define rutinas básicas de flujo de E/S de C++ (v2.0)*
ANSI C	*LIMITS.H*	*Parámetros y constantes sobre la capacidad del sistema*
ANSI C	*LOCALE.H*	*Define funciones sobre el país e idioma*
ANSI C	*MATH.H*	*Define prototipos para las funciones matemáticas*
	MEM.H	*Define las funciones de gestión de memoria*
	PROCESS.H	*Contiene estructuras y declaraciones para las funciones* ***spawn()***, ***exec()***
ANSI C	*SETJMP.H*	*Declaraciones para dar soporte a saltos no locales*
	SHARE.H	*Parámetros utilizados en funciones que utilizan arhivos-compartidos*
ANSI C	*SIGNAL.H*	*Declara constantes y declaraciones para utilizarlos en funciones* ***signal()*** *y* ***raise()***
ANSI C	*STDARG.H*	*Soporte para aceptar un número variable de argumentos*
ANSI C	*STDDEF.H*	*Declara varios tipos de datos y macros de uso común*
ANSI C	*STDIO.H*	*Declara tipos y macros para E/S estándar*
C++	*STDIOSTR.H*	*Declara las clases de flujo para utilizar con estructuras del archivo* ***stdio.h***
ANSI C	*STDLIB.H*	*Define algunas de las rutinas comúnmente utilizadas*
C++	*STREAM.H*	*Define las clases de flujo de C++ para utilizarlas con arrays de bytes en memoria*
ANSI C	*STRING.H*	*Define varias rutinas de manipulación de cadenas y de memoria*
	SYS\STAT.H	*Declara constantes simbólicas utilizadas para abrir y crear archivos*
	SYS\TIMEB.H	*Define la función* ***ftime()*** *y la estructura* ***timeb***
C++	*SYS\TYPES.H*	*Define el tipo* ***time_t***
ANSI C	*TIME.H*	*Estructuras y prototipos para funciones de tiempo*

	VALUES.H	*Declara constantes dependientes de la máquina*

Ejercicios

1. El seno de un ángulo puede calcularse mediante la serie

$$\text{sen}\,(x) = x - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \cdots$$

donde **x** se expresa en radianes (π radianes = 180 grados). Escribe un programa que calcule y muestre en pantalla el seno de un ángulo mediante la serie anterior. Finaliza el cálculo en el término de la serie cuyo valor sea menor o igual a10^{-3}. Para el cálculo de $\mathbf{x^n}$ crea una función no recursiva de prototipo

double potencia (duble x, int n);

De igual modo, para el cálculo de **n!** crea una función no recursiva de prototipo

double factorial (int n);

El ángulo debe ser un valor comprendido entre 0 y 360. En caso se enviará un mensaje a pantalla.

2. Escribe un programa que calcule $\mathbf{x^n}$, siendo **x** y **n** dos enteros positivos que se introducen por teclado. Para el cálculo crea una función recursiva de prototipo

long potencia (int x, int n);

que solucione el cálculo.

3. Escribe un programa que muestre los **N** primeros términos de la sucesión de Fibonacci, utilizando una función recursiva

int Fibonacci (int N);

que devuelva el elemento **N**. El valor **N** se leerá del teclado.

4. Escribe un programa que muestre en pantalla la información del equipo proporcionada por las funciones de biblioteca **biosequip()** y **biosmemory()**.

5. Por medio de la función **bioskey()** construye un programa que muestre en pantalla el estado de pulsación del teclado con el siguiente formato:

Mayúsculas derecha: SI/NO
Mayúsculas izquierda: SI/NO
Tecla Control: SI/NO
Tecla Alt: SI/NO
Tecla Bloq Despl: SI/NO
Tecla Bloq Num: SI/NO
Tecla Bloq Mayús: SI/NO
Tecla Ins: SI/NO

NOTA: Para evitar el efecto que produce el cursor en la presentación de la pantalla (hay que estar continuamente imprimiendo la información en la pantalla) elimínalo usando la función **_setcursortype()** de la biblioteca estándar.

6. Escribe una función de prototipo

```
char *Intro (int f, int c, int tam, char *cad);
```

que utilice la función **cgets()** para capturar en la fila **f**, columna **c**, una cadena de caracteres de longitud máxima **tam**, y la almacene en **cad**. Devuelve **cad**.

7. Escribe una función de prototipo

```
int strdigit (char *cad);
```

que inspeccione la cadena **cad** y devuelva **1** si **cad** está compuesta por dígitos numéricos, y **0** si algún carácter de **cad** no es numérico. Utiliza la función **isdigit()**.

8. Escribe una función de prototipo

```
char *Format_fecha (char *fecha, int tipo, char *format);
```

que recibe en la cadena **fecha** una fecha con formato DDMMAAAA y pone en la cadena **format** esta fecha en un formato indicado por **tipo**. Los valores permitidos por **tipo** y sus formatos correspondientes se muestran en la siguiente tabla:

tipo	***format***
0	*DD/MM/AA*
1	*DD/MM/AAAA*
2	*DD de MMMMMMMMMM de AAAA*
3	*diasemana, DD de MMMMMMMMMM de AAAA*

Si **fecha** no es una fecha válida o **tipo** está fuera de rango, **format** será la cadena nula. Devuelve **format**. Utiliza la función **sprintf()**.

9. Escribe una función de prototipo

```
int Valida_fecha (char *fecha);
```

que recibe en la cadena **fecha** una fecha en formato DDMMAAAA y devuelve un valor de **0** a **6** si la fecha es correcta, indicando con los valores 0 a 6 el día de la semana a que corresponde **fecha** (0=domingo, 1=lunes, ...). Si la fecha no es correcta, devolverá **-1**. Utiliza la función **intdos** para hacer las llamadas a las funciones 2Ah y 2Bh de la INT 21h.

10. Escribe una función de prototipo

```
int Tecla (int *scan);
```

que captura una pulsación de tecla y devuelve en **scan** el código de exploración, y mediante **return**, su código ASCII. Utiliza la función **bioskey()**.

11. Escribe una función de prototipo

```
int Mensaje (int st, int fil, int col, char *cad);
```

que presenta en pantalla la cadena **cad** en la fila **fil**, columna **col**. Si **st** vale **1**, espera a que se pulse una tecla, devolviendo mediante **return** su código ASCII si se pulsó una tecla normal, y 1000+código de exploración si se pulsó una tecla especial. Si **st** vale **0**, no espera a que se pulse ninguna tecla.

7
Matrices y punteros

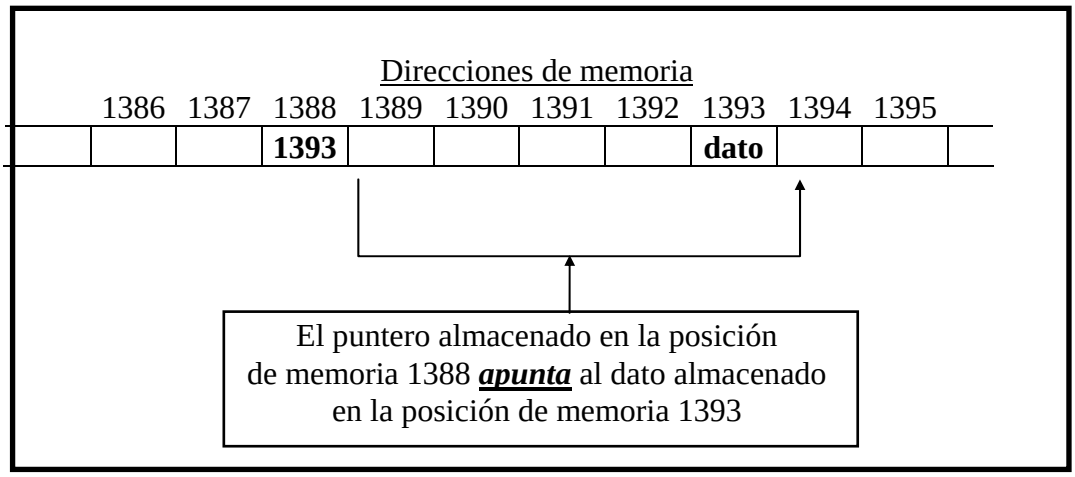
¿Qué es una matriz?

Una matriz es una estructura de datos interna que almacena un conjunto de datos del mismo tipo bajo un nombre de variable común. La posición de un elemento dentro de la matriz viene identificada por uno o varios índices, de tal modo que a cada elemento se accede mediante el nombre de la matriz y sus índices.

La **dimensión** de una matriz es el número de índices necesario para identificar un elemento.

¿Qué son los punteros?

Un puntero es una variable que contiene una dirección de memoria. Por ejemplo, la dirección de otra variable.



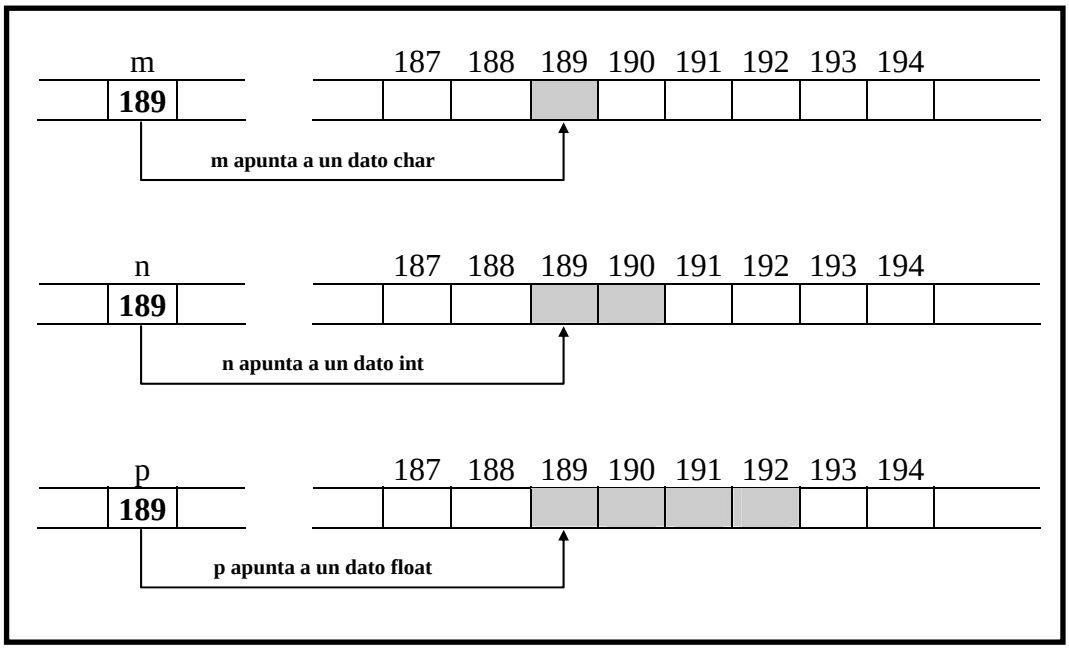
Las variables puntero se declaran de la siguiente forma:

tipo *nombre;

siendo **nombre** el identificador de la variable puntero, y **tipo** el tipo de variable a la que apunta. Por ejemplo,

```
char *m;
int *n;
float *p;
```

En estas declaraciones, las variables **m**, **n** y **p** son punteros que *apuntan*, respectivamente, a datos de tipo **char**, **int** y **float**. Es importante darse cuenta de que ni **m** es una variable de tipo **char**, ni **n** de tipo **int**, ni **p** de tipo **float**. Los tipos definen el tipo de dato al que apunta el puntero



Los operadores de punteros son los que estudiamos en el Capítulo 2, y se definieron en la página 22, es decir,

- **&** dirección de
- ***** en la dirección de

El operador ***** sólo se puede aplicar a punteros. Después de

```
float m;
float *p;
...
...
m = *p;
```

la variable **m** almacena el contenido de la dirección apuntada por **p**. Del mismo modo después de

```
*p = m;
```

el valor de **m** se almacena en la dirección apuntada por **p**. La siguiente asignación

```
int valor = 100, q;
...
...
q = valor;
```

puede conseguirse también mediante

```
int valor = 100, q, *m;
...
...
m = &valor;
q = *m;
```

es decir, se almacena en la variable **q** el valor **100**.

El código de formato para visualizar variables puntero mediante funciones tipo **printf()** es **%p**. Así,

```
int *m;
...
...
printf ("%p", m);
```

muestra en pantalla la dirección almacenada por **m**. La visualización se hace en hexadecimal. Otro código de formato relacionado con punteros es **%n**. Este código da lugar a que el número de caracteres que se han escrito en el momento en que se encuentra **%n**, se asocien a una variable cuya dirección se especifica en la lista de argumentos. El siguiente programa muestra en pantalla la frase

Se han escrito 11 caracteres

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

void main (void)
{
  int p;

  clrscr ();
  printf ("Hola mundo %n", &p);
  printf ("\nSe han escrito %d caracteres", p);
}
```

Las operaciones permitidas con punteros son:

- Asignación
- Incremento / Decremento
- Suma / Resta
- Comparación

Vamos a describir cada una de ellas.

Asignación

Dadas las declaraciones

```
float x;
float *p, *q;
```

la forma de asignar a **p** y **q** la dirección de **x** es:

```
p = &x;
q = &x;
```

Ahora **p** y **q** almacenan la misma dirección de memoria: la de la variable **x**. El mismo efecto se consigue con la asignación directa entre punteros:

```
p = &x;
q = p;
```

No es correcta una sentencia como

```
p = x;
```

puesto que **x** es una variable de tipo **float** (almacena un dato **float)**, mientras que **p** almacena *la dirección* de un dato de tipo **float**.

Por medio de punteros pueden hacerse asignaciones de contenidos. Por ejemplo:

```
float a, b;
float *p, *q;
...
...
a = 1.5;
p = &b;
q = &a;
*p = *q;
```

En esta secuencia, **p** y **q** almacenan valores diferentes (la dirección de **b** y la dirección de **a**, respectivamente). La última sentencia asigna contenidos, es decir, almacena en el lugar apuntado por **p** (la variable **b**) lo que hay en el lugar apuntado por **q** (la variable **a**). Es, por tanto, equivalente a

```
b = a;
```

Incremento / Decremento

Para comprender estas operaciones, debemos tener en cuenta que hacen referencia a elementos de memoria y no a direcciones. Esto quiere decir que los operadores **++** y **--** actúan de modo diferente según el tipo apuntado por el puntero. Si **p** es un puntero a caracteres (**char *p**) la operación **p++** incrementa el valor de **p** en **1**. Si embargo, si **p** es un puntero a enteros (**int *p**), la misma

operación **p++** incrementa el valor de **p** en **2** para que apunte al siguiente elemento, pues el tipo **int** ocupa dos bytes. Del mismo modo, para el tipo **float** la operación **p++** incrementa el valor de **p** en **4**.

Lo dicho para el operador **++** se cumple exactamente igual, pero decrementando, para el operador **--**.

Suma / Resta

Ocurre exactamente lo mismo que con las operaciones de incremento y decremento. Si **p** es un puntero, la operación

```
p = p + 5;
```

hace que **p** apunte 5 elementos más allá del actual. Si **p** estaba definido como un puntero a caracteres, se incrementará su valor en 5, pero si estaba definido como un puntero a enteros, se incrementará en 10.

Comparación

Pueden compararse punteros del mismo modo que cualquier otra variable, teniendo siempre presente que se comparan direcciones y no contenidos.

```
int *p, *q;
...
...
if (p == q) puts ("p y q apuntan a la misma posición de memoria");
...
...
if (*p == *q) puts ("Las posiciones apuntadas por p y q almacenan el mismo valor");
```

En el segundo caso puede aparecer el mensaje aunque **p** y **q** apunten a direcciones diferentes.

Matrices unidimensionales

Son aquellas que sólo precisan de un índice para acceder a cada elemento. También se les llama **vectores** o **listas**.

Todos los elementos de un vector se almacenan en posiciones de memoria contiguas, almacenándose el primer elemento en la dirección más baja.

Un vector se declara de la siguiente forma:

tipo nombre[num_elem];

donde **tipo** es el tipo de dato de todos los elementos del vector, **nombre** es cualquier identificador válido C, y **num_elem** es el número de elementos del vector. Por ejemplo,

```
char frase[20];
int numero[16];
float valor[12];
```

declaran, por este orden, un vector de 20 caracteres, otro de 16 elementos enteros y otro de 12 elementos en coma flotante. El número de bytes ocupado por una matriz se calcula multiplicando el número de elementos por el tamaño de cada uno de ellos. Así, los 3 vectores anteriores ocupan, respectivamente, 20, 32 y 48 bytes.

En las declaraciones de matrices, el nombre de la matriz sin índices es un puntero al primer elemento. Así, en las declaraciones anteriores **frase** es un puntero a **char** y almacena la dirección de **frase[0]**. Lo mismo ocurre para **numero** y **valor**.

En un vector de **N** elementos, el primero se referencia con el índice **0** y el último con el índice **N-1**. Así, en el vector

```
int numero[16];
```

el primer elemento es **numero[0]** y el último **numero[15]**.

Es importante tener en cuenta que C no hace comprobación de límites en el proceso de matrices. El control debe hacerlo el programador. Así, es posible escribir

```
numero[30] = 250;
```

manejando el elemento 30 del vector **numero** que se declaró para almacenar sólo 16 elementos. Las consecuencias para el programa suelen ser desastrosas.

En el siguiente programa se carga un vector de 10 caracteres desde el teclado, y se muestra después la dirección y contenido de cada elemento.

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

void main (void)
{
    register int i;
    char vector[10];

    for (i = 0; i <= 9; i++) vector[i] = getche ();

    for (i = 0; i <= 9; i++) printf ("\nLugar: %d - Dirección: %p - Valor: %c", i, vector + i, vector[i]);
}
```

Cadenas de caracteres

Un caso particular de vector es la cadena de caracteres. Una cadena de caracteres se declara mediante

char nombre[num_car];

y permite almacenar **num_car-1** caracteres y el carácter nulo **'\0'** de terminación. Por lo tanto, una declaración como

```
char frase[21];
```

es apta para almacenar 20 caracteres y el nulo.

C permite la inicialización de cadenas de caracteres en la declaración, mediante sentencias del tipo

```
char cadena[ ] = "Esto es una cadena de caracteres";
```

en la que no es necesario añadir el nulo final ni indicar el tamaño, pues lo hace automáticamente el compilador.

Las operaciones con cadenas de caracteres como copiar, comparar, concatenar, medir, etc., se hacen mediante funciones de la biblioteca estándar. Su utilización requiere incluir el archivo de cabecera **string.h**. Veamos alguna de ellas.

char *strcat (char *cad1, const char *cad2);

Concatena **cad2** a **cad1** devolviendo la dirección de **cad1**. Elimina el nulo de terminación de **cad1** inicial.

Ejemplo:

```
char cad1[80], cad2[80];
...
...
printf ("\nTeclee una frase: ");
gets (cad1);
printf ("\nTeclee otra frase: ");
gets (cad2);

strcat (cad1, cad2);
puts (cad1);
```

En este ejemplo, si se teclea **Primera frase** para **cad1** y **Segunda frase** para **cad2**, se muestra en pantalla

Primera fraseSegunda frase

Tendría el mismo efecto

```
puts (strcat (cad1, cad2));
```

Hay que asegurarse de que el tamaño de **cad1** es suficiente para almacenar el resultado.

char *strchr (const char *cad, int ch);

Devuelve la dirección de la primera aparición del carácter **ch** en la cadena **cad**. Si no se encuentra devuelve un puntero nulo.

Ejemplo:

```
char *p, caracter, cadena[80];
int lugar;
...
...
printf ("\nTeclee un carácter: ");
caracter = getche ();
printf ("\nTeclee una frase: ");
gets (cadena);

p = strchr (cadena, caracter);
if (!p) printf ("\nNo está el carácter %c en la frase", caracter);
else {
    lugar = p - cadena;
    printf ("\nEl carácter %c ocupa el lugar %d de la frase", caracter, lugar);
}
```

int strcmp (const char *cad1, const char *cad2);

Para la comparación de cadenas de caracteres no se permite la utilización de los operadores >, <, >=, **!=**, etc., sino que se debe utilizar la función **strcmp**. Esta función compara lexicográficamente **cad1** y **cad2** y devuelve un entero que se debe interpretar como sigue:

< 0	**cad1 < cad2**
0	**cad1 igual a cad2**
> 0	**cad1 > cad2**

Ejemplo:

```
char cad1[40], cad2[40];
...
...
printf ("\nTeclee una frase: ");
gets (cad1);
printf ("\nTeclee una frase: ");
gets (cad2);

n = strcmp (cad1, cad2);
```

```
if (!n) puts ("Son iguales");
else if (n < 0) puts ("La primera es menor que la segunda");
else puts ("La primera es mayor que la segunda");
```

Es necesario insistir en que la comparación no se hace en cuanto al tamaño de la cadena sino en cuanto al orden de los caracteres en el código ASCII. Esto quiere decir que si **cad1** es **ABCDE** y **cad2** es **xyz**, la función **strcmp** devuelve un valor negativo, pues se considera que **cad1** < **cad2** ya que el carácter **A** tiene un código ASCII menor que el carácter **x**.

char *strcpy (char *cad1, const char *cad2)

Copia la cadena **cad2** en **cad1**, sobreescribiéndola. Devuelve la dirección de **cad1**. Hay que asegurarse de que el tamaño de **cad1** es suficiente para albergar a **cad2**.

Ejemplo:

```
char cadena[40];
...
...
strcpy (cadena, "Buenos días");
```

int strlen (const char *cad);

Devuelve el número de caracteres que almacena **cad** (sin contar el nulo final).

Ejemplo:

```
char cad[30];
...
...
printf ("\nTeclee una frase: ");
gets (cad);
printf ("\nLa frase <<<%s>>> tiene %d caracteres", cad, strlen (cad));
```

char *strlwr (char *cad);

Convierte **cad** a minúsculas. La función no tiene efecto sobre los caracteres que no sean letras mayúsculas. Tampoco tiene efecto sobre el conjunto extendido de caracteres ASCII (código mayor que 127), por lo que no convierte las letras Ñ, Ç o vocales acentuadas. Devuelve la dirección de **cad**.

Ejemplo:

```
char cad[40];
...
...
printf ("\nTeclee una frase en mayúsculas: ");
```

```
gets (cad);
printf ("\nLa cadena en minúsculas es %s", strlwr (cad));
```

char *strrev (char *cad);

Invierte la cadena **cad** y devuelve su dirección.

Ejemplo:

```
char cad[80];
...
...
printf ("\nTeclee una frase: ");
gets (cad);
printf ("\nFrase invertida: %s", strrev (cad));
```

char *strset (char *cad, int ch);

Reemplaza cada uno de los caracteres de **cad** por el carácter **ch**. Devuelve la dirección de **cad**.

Ejemplo:

```
char cad[80];
...
...
strcpy (cad, "12345");
strset (cad, 'X');
puts ("Se visualiza una cadena con 5 X");
puts (cad);
```

char *strupr (char *cad);

Convierte **cad** a mayúsculas. La función no tiene efecto sobre los caracteres que no sean letras minúsculas ni sobre el conjunto extendido de caracteres ASCII (letras ñ, ç o vocales acentuadas). Devuelve la dirección de **cad**.

Ejemplo:

```
char cad[40];
...
...
printf ("\nTeclee una frase en minúsculas: ");
gets (cad);
printf ("\nLa frase en mayúsculas es %s", strupr (cad));
```

En el Capítulo 13 se muestran más funciones de cadenas de caracteres.

También existe un amplio conjunto de funciones que manejan caracteres individuales. La mayoría de estas funciones informan del tipo de carácter incluido en su argumento devolviendo un valor 1 ó 0. Estas funciones tienen su prototipo definido en **ctype.h** y generalmente no consideran el conjunto ASCII extendido. Alguna de estas funciones se explican a continuación:

int isalnum (int ch) Devuelve 1 si **ch** es alfanumérico (letra del alfabeto o dígito) y 0 en caso contrario.

int isalpha (int ch) Devuelve 1 si **ch** es una letra del alfabeto y 0 en caso contrario.

int isdigit (int ch) Devuelve 1 si **ch** es un dígito del 0 al 9, y 0 en caso contrario.

int islower (int ch) Devuelve 1 si **ch** es un letra minúscula y 0 en caso contrario.

int isupper (int ch) Devuelve 1 si **ch** es una letra mayúscula y 0 en caso contrario.

int tolower (int ch) Devuelve el carácter **ch** en minúscula. Si **ch** no es una letra mayúscula la función devuelve **ch** sin modificación.

int toupper (int ch) Devuelve el carácter **ch** en mayúscula. Si **ch** no es una letra minúscula la función devuelve **ch** sin modificación.

En el Capítulo 13 se muestran más funciones de caracteres.

El programa siguiente hace uso de alguna de las funciones anteriores para examinar una cadena de caracteres y convertir las minúsculas a mayúsculas y viceversa. Además cuenta cuántos caracteres son dígitos numéricos.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

void main (void)
{
    char cadena[100];
    int contador = 0;
    register int i;

    clrscr ();
    printf ("\nTeclee una cadena de caracteres: ");
    gets (cadena);

    for (i = 0; i <= strlen (cadena); i++) {
        if (isupper (cadena[i])) cadena[i] = tolower (cadena[i]);
        else if (islower (cadena[i])) cadena[i] = toupper (cadena[i]);
        else if (isdigit (cadena[i])) contador++;
    }
    printf ("\nLa cadena tiene %d dígitos numéricos\n", contador);
    puts (cadena);
```

```
}
```

Punteros y matrices

Puesto que una matriz se identifica mediante la dirección del primer elemento, la relación entre punteros y matrices es estrecha. Sea la declaración

```
int matriz[100];
```

Podemos identificar un elemento, digamos el 25, de cualquiera de las dos formas siguientes:

```
matriz[24];
*(matriz + 24);
```

puesto que **matriz** (sin índices) es la dirección del primer elemento.

El uso de punteros para manejar matrices es, en general, más eficiente y rápido que el uso de índices. La decisión de qué método utilizar (punteros o índices) depende del tipo de acceso. Si se va a acceder a los elementos de la matriz de forma aleatoria, es mejor utilizar índices. Sin embargo, si el acceso va a ser secuencial, es más adecuado usar punteros. Este segundo caso es el típico en las cadenas de caracteres. Por ejemplo, las sentencias siguientes muestran en pantalla, carácter a carácter una cadena de caracteres.

```
char *p, cadena[30];
register int i;
...
...
for (i = 0; cadena[i]; i++) putch (cadena[i]);
...
...
p = cadena;
for (; *p; p++) putch (*p);
...
...
p = cadena;
while (*p) putch (*p++);
```

Las dos últimas son las más adecuadas. Otro ejemplo lo constituyen las 3 funciones que se describen a continuación. Estas funciones comparan dos cadenas de caracteres de modo similar a como lo hace **strcmp()**.

```
int compara1 (char *cad1, char *cad2)
{
    register int i;

    for (i = 0; cad1[i]; i++) if (cad1[i] - cad2[i]) return (cad1[i] - cad2[i]);
    return 0;
```

```
}
```

```
int compara2 (char *cad1, char *cad2)
{
    char *p1, *p2;

    p1 = cad1;
    p2 = cad2;

    while (*p1) {
        if (*p1 - *p2) return (*p1 - *p2);
        else {
            p1++;
            p2++;
        }
    }
    return 0;
}
```

```
int compara3 (char *cad1, char *cad2)
{
    char *p1, *p2;

    p1 = cad1;
    p2 = cad2;

    for (; *p1; p1++, p2++) if (*p1 - *p2) return (*p1 - *p2);
    return 0;
}
```

De las 3 funciones anteriores son más eficientes las dos últimas.

Matrices bidimensionales

Una matriz bidimensional es aquella que necesita dos índices para identificar un elemento. Puede decirse que una matriz bidimensional es una estructura de datos organizados en filas y columnas. Se declaran de la siguiente forma:

tipo nombre[nº filas][nº columnas];

Por ejemplo, para declarar una matriz de números enteros organizada en 8 filas y 6 columnas se escribe

```
int total[8][6];
```

El primer elemento de la matriz es **total[0][0]** y se almacena en una dirección de memoria identificada por **total**. El último es **total[7][5]**.

Un caso particular de matrices bidimensionales lo constituyen las matrices de cadenas de caracteres. Por ejemplo, la sentencia

```
char cadenas[10][25];
```

declara una matriz de 10 cadenas de 24 caracteres más el nulo. Para acceder a una cadena en particular basta especificar el índice izquierdo (número de cadena). Así,

```
gets (cadena[6]);
```

lee del teclado una cadena de caracteres y la almacena en la séptima cadena (la primera es **cadena[0]**) de la matriz . Esta sentencia es equivalente a

```
gets (&cadena[6][0]);
```

Para acceder a un carácter concreto de una cadena hay que especificar ambos índices. Por ejemplo, la sentencia

```
cadena[3][9] = 'X';
```

almacena en el 10º carácter de la 4ª cadena el carácter **X**.

```
//Cambia en un grupo de cadenas las vocales por #
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
    char cadena[10][70];
    register int i, j;

    clrscr ();
    for (i = 0; i <= 9; i++) {
        printf ("\nCadena nº %d: ", i);
        gets (cadena[i]);
    }
    for (i = 0; i <= 9; i++) {
        for (j = 0; cadena[i][j]; j++) {
            if (strchr ("aeiouAEIOU", cadena[i][j])) cadena[i][j] = '#';
        }
        puts (cadena[i]);
    }
}
```

En el siguiente programa se carga del teclado una matriz de 10 cadenas de caracteres y se busca en ellas la primera aparición de la cadena **HOLA**, informando del número de cadena en que se encuentra y de la posición que ocupa en ella. Para ello se utiliza la función **strstr** explicada en el Capítulo 13 (página 212).

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

```
#include <conio.h>
#include <string.h>

void main (void)
{
    char cadena[10][50];
    register int i;
    char *p;

    clrscr ();
    for (i = 0; i <= 9; i++) {
        printf ("\nCadena nº %d: ", i);
        gets (cadena[i]);
        strupr (cadena[i]);
    }

    for (i = 0; i <= 9; i++) {
        p = strstr (cadena[i], "HOLA");
        if (p) printf ("\nCadena nº %d, posición %d", i, p - cadena[i]);
    }
}
```

Podemos acceder a los elementos de una matriz bidimensional mediante punteros. Se accede al elemento **matriz[i][j]** mediante la fórmula:

***(matriz + i * nº_de_filas + j)**

Matrices de más de 2 dimensiones

En C pueden manejarse matrices de más de 2 dimensiones. Para declarar estas matrices se hace

tipo nombre[tamaño 1][tamaño 2] ... [tamaño N];

La siguiente sentencia declara una matriz tridimensional de elementos de tipo **float**:

```
float matriz3D[5][3][8];
```

El problema de este tipo de matrices es la cantidad de memoria que pueden llegar a ocupar, ya que esta aumenta exponencialmente con el número de dimensiones. Es por ello que para estas estructuras suele utilizarse asignación dinámica de memoria, de modo que permite asignar o recortar memoria según se va necesitando. Estudiaremos esto en el Capítulo 9.

Las siguientes sentencias muestran como se puede cargar desde el teclado la matriz **matriz3D**.

```
for (i = 0; i  <= 4; i++) {
    for (j = 0; j <= 5; j++) {
        for (k = 0; k <= 7; k++) {
            printf ("\nElemento %d-%d-%d: ", i, j, k);
            scanf ("%f", &matriz3D[i][j][k]);
        }
    }
}
```

Cómo inicializar matrices

Cualquier matriz puede ser inicializada en el momento de la declaración. Para ello se encierran entre llaves, separados por comas, los datos de la inicialización. Veamos algunos ejemplos:

```
float vector[5] = {1.23, 16.9, -1.2, 2.06, 31.15};
int tabla[2][3] = {5, 62, 34, 21, 43, 90};
```

Para más claridad, en las matrices de dos dimensiones suele hacerse:

```
int tabla[2][3] = {    5,   62, 34,
                       21, 43, 90};
```

En los dos últimos casos, la matriz **tabla** queda inicializada de la siguiente manera:

```
tabla[0][0] = 5     tabla[0][1] = 62        tabla[0][2] = 34
tabla[1][0] = 21    tabla[1][1] = 43        tabla[1][2] = 90
```

El mismo efecto se produce usando anidamiento de llaves:

```
int tabla[2[3] = {    {   5,   62, 34   },
                      { 21,   43, 90   } };
```

Las matrices de cadenas de caracteres pueden inicializarse de la forma

```
char frases[3][30] = {    "Primera cadena", "Segunda cadena", "Tercera" };
```

En todas las matrices multidimensionales puede omitirse el primer índice cuando se inicializan en la declaración. Por ello, las declaraciones siguientes:

```
int tabla[3][4] = { 6, 12, 25, 4, 5, 13, 7, 2, 2, 4, 9, 6};
int tabla[ ][4] = {6, 12, 25, 4, 5, 13, 7, 2, 2, 4, 9, 6};
```

son idénticas. En el segundo caso, el compilador se encarga de calcular el tamaño correspondiente.

Matrices como argumentos de funciones

Cuando se pasa una matriz como argumento de una función, el compilador genera la llamada con la dirección del primer elemento. De esta forma se evita pasar la matriz completa, lo que consume mucha memoria y hace el proceso más lento. Sea, por ejemplo, la matriz unidimensional

```
int vector[30];
```

que se quiere pasar como argumento a una función **mayor()** que devuelve como valor de retorno el elemento más grande de la matriz. La llamada se hace de la forma

```
a = mayor (vector);
```

y la función **mayor()** puede declararse de cualquiera de las formas siguientes:

```
int mayor (int x[30])
{
...
...
}
```

```
int mayor (int x[ ])
{
...
...
}
```

En cualquiera de los 2 casos el resultado es idéntico, pues ambas declaraciones le indican al compilador que se va a recibir en el argumento la dirección de un entero (la del primer elemento de la matriz). En el primer caso, el valor 30 no se tiene en cuenta.

El siguiente programa carga desde el teclado una matriz de enteros y muestra el mayor de ellos, calculándolo mediante una función.

```
#include <stdio.h>

void main (void)
{
    register int i;
    int vector[30];

    for (i = 0; i <= 29; i++) {
        printf ("\nElemento %d: ", i);
        scanf ("%d", &vector[i]);
    }
    printf ("\nEl mayor es %d", mayor (vector, 30));
}

int mayor (int tabla[ ], int num_element)
{
    register int i;
    int max;

    max = tabla[0];
```

```
    for (i = 0; i < num_element; i++) if (max < tabla[i]) max = tabla[i];

    return max;
}
```

Nótese que es necesario pasar el tamaño de la matriz como argumento, pues de lo contrario la función **mayor()** no tendría ninguna información sobre el número de elementos de la matriz.

Para las matrices bidimensionales, aunque se pasa sólo la dirección del primer elemento, es necesario especificar el número de columnas (segundo índice) para que el compilador sepa la longitud de cada fila. Por ejemplo, una función que reciba como argumento una matriz declarada como

```
int matriz2D[10][20];
```

se declara de la forma siguiente:

```
funcion (int x[ ][20])
{
    ...
}
```

y la llamada se hace con una sentencia como

```
funcion (matriz2D);
```

En general, para matrices multidimensionales, sólo se puede omitir el primer índice en la declaración. Por ejemplo, una matriz declarada mediante

```
int matriz3D[5][10][20];
```

se pasa como argumento de una función mediante una llamada de la forma

```
funcion (matriz3D);
```

y la declaración de la función tiene el siguiente aspecto:

```
funcion (int x[ ][10][20])
{
    ...
}
```

Argumentos de la función main()

Cuando, por ejemplo, escribimos desde el indicador del DOS una orden como

C:\>XCOPY *.DAT B: /S

estamos ejecutando un programa llamado **XCOPY.EXE** con 3 parámetros: ***.DAT**, **B:** y **/S**. Esos parámetros, de alguna forma, son leídos por el programa **XCOPY**.

Para hacer esto en cualquier programa C, es necesario modificar la forma en que se llama a la función **main()**, incluyendo en ella argumentos que permitan la obtención de los parámetros introducidos en la línea de órdenes. La forma de hacer esto es la siguiente:

main (int argc, char *argv[ ])

Veamos el significado de las variables **argc** y **argv**.

argc: Entero que indica el número de parámetros tecleados (incluye el nombre del programa).
argv[ ]: Matriz de cadenas de caracteres. Cada uno de los elementos **argv[i]** es una cadena que almacena un argumento.

La variable **argc** vale 1 como mínimo, puesto que se cuenta el nombre del programa. Los parámetros se identifican mediante **argv** de la siguiente manera:

- **argv[0]** cadena que almacena el nombre del programa.
- **argv[1]** cadena que almacena el primer parámetro.
- **argv[2]** cadena que almacena el segundo parámetro.
  ...
  ...
- **argv[argc]** vale cero (En realidad es un puntero nulo).

Para que los argumentos sean tratados como diferentes tienen que ir separados por uno o varios espacios blancos. Así, en el ejemplo anterior la variable **argc** vale 4 (nombre del programa y 3 parámetros). Si escribimos, por ejemplo

C:\>PROG PAR1,PAR2

la variable **argc** valdría 2, puesto que la cadena **PAR1,PAR2** queda identificada como un sólo parámetro (almacenado en la cadena **argv[1]**) ya que la coma no actúa como separador. Para que **PAR1** y **PAR2** sean tratados como dos parámetros diferentes, debe ejecutarse **PROG** mediante

C:\>PROG PAR1 PAR2

El siguiente programa lista los parámetros, si los hay, de la línea de órdenes.

```
#include <stdio.h>

void main (int argc, char *argv[ ])
{
    register int i;

    printf ("\nNombre del programa: %s", argv[0]);
    if (argc == 1) printf ("\nNo se han introducido parámetros");
    else {
        printf ("\nParámetros en la línea de órdenes: ");
        for (i = 1; i < argc; i++) printf ("\n%d: %s", i, argv[i]);
    }
}
```

La función **main()** soporta una variable más, llamada **env**:

env[ ]: Matriz de cadenas de caracteres. Cada uno de los elementos de la matriz es una cadena que almacena el nombre y contenido de una variable del entorno.

```
#include <stdio.h>

void main (int argc, char *argv[ ], char *env[ ])
{
    register int i;

    printf ("\nNombre del programa: %s", argv[0]);
    if (argc == 1) printf ("\nNo se han introducido parámetros");
    else {
        printf ("\nParámetros en la línea de órdenes: ");
        for (i = 1; i < argc; i++) printf ("\n%d: %s", i, argv[i]);
    }

    printf ("\nVariables de entorno: \n");
    for (i = 0; env[i]; i++) puts (env[i]);
}
```

Si este programa esta compilado y linkado como **PROG.EXE** y se ejecuta como

C:\>PROG PAR1 PAR2	**C:\> PROG**
proporciona	**la salida**
Nombre del programa: PROG **Parámetros en la línea de órdenes:** **1: PAR1** **2: PAR2** **Variables de entorno:** **COMSPEC=C:\DOS\COMMAND.COM** **PROMPT=PG** **PATH=C:\;C:\DOS;C:\WINDOWS**	**Nombre del programa: PROG** **No se han introducido parámetros** **Variables de entorno:** **COMSPEC=C:\DOS\COMMAND.COM** **PROMPT=PG** **PATH=C:\;C:\DOS;C:\WINDOWS**

Matrices de punteros

Pueden definirse matrices de punteros, es decir, matrices cuyos elementos son direcciones. Por ejemplo,

```
int *pint[20];
char *pchar[40];
```

declaran una matriz **pint** de 20 punteros a enteros y otra matriz **pchar** de 40 punteros a caracteres.

Para asignar las direcciones se hace igual que con cualquier otro puntero. Mediante las siguientes sentencias

```
int *pint[20], a;
...
...
a = 20;
pint[3] = &a;
printf ("%d", *pint[3]);
```

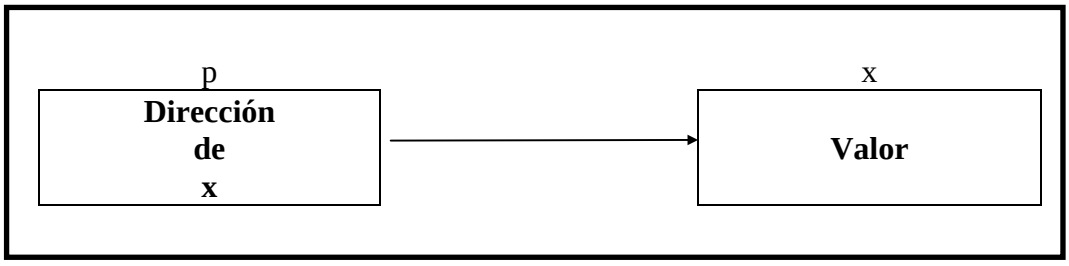
se muestra en pantalla el número 20.

Un ejemplo típico de matriz de punteros es la matriz de cadenas de caracteres. Las declaraciones siguientes son equivalentes

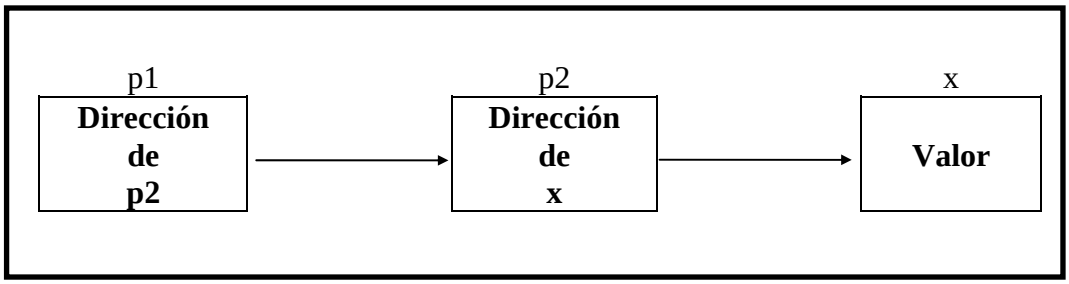
```
char dias[ ][10] = {  "Domingo", "Lunes", "Martes", "Miércoles",
                      "Jueves", "Viernes", "Sábado" };

char *dias[ ] = {  "Domingo", "Lunes", "Martes", "Miércoles",
                   "Jueves", "Viernes", "Sábado" };
```

Punteros a punteros

En la mayoría de los casos un puntero apunta a un dato.



y ***p** proporciona el valor de **x**. Pero también se pueden definir punteros a punteros.



En este caso ***p2** proporciona el valor de **x**. Pero **p1** proporciona la dirección de **p2**. Para acceder a **x** mediante **p1** hay que hacer ****p1**.

```
#include <stdio.h>

void main (void)
{
    int x, *p, **q;

    x = 10;
    p = &x;
    q = &p;

    printf ("%d", **q);
}
```

Este programa muestra en pantalla el número 10 que almacena **x**.

Punteros a funciones

Puesto que una función ocupa una posición en la memoria, pueden definirse punteros a funciones, y hacer llamadas a la función por medio de la dirección. Esto se aplica generalmente para trabajar con matrices de funciones. Veámoslo con un ejemplo. El siguiente programa realiza las operaciones básicas suma, resta y producto, por medio de llamadas a funciones que forman parte de una matriz.

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

int menu (void);
int suma (int, int);
int resta (int, int);
int producto (int, int);

int (*calculo[3]) (int, int) = { suma, resta, producto };

void main (void)
{
  int n, x, y, resultado;

  while (( n = menu()) != 3) {
    printf ("\nTeclea dos números: ");
    scanf ("%d %d", &x, &y);

    resultado = (*calculo[n]) (x, y);
    printf (" = %d", resultado);
    getch ();
  }
}

int menu (void)
{
```

```
  int opcion;

  clrscr ();
  puts ("0. Suma");
  puts ("1. Resta");
  puts ("2. Producto");
  puts ("3. Salir");

  do {
    opcion = getch ();
  } while (opcion < '0' || opcion > '3');

  return opcion - 48;
}

int suma (int a, int b)
{
  printf ("\n%d + %d", a, b);
  return a + b;
}

int resta (int a, int b)
{
  printf ("\n%d - %d", a, b);
  return a - b;
}

int producto (int a, int b)
{
  printf ("\n%d * %d", a, b);
  return a * b;
}
```

En general, un puntero a una función se declara de acuerdo a

tipo (*pfunc) ();

siendo **pfunc** el nombre del puntero, y **tipo** el tipo del valor devuelto por la función. Los paréntesis son necesarios, pues

tipo *pfunc ();

declara una función que devuelve un puntero. Para llamar a la función debe hacerse

(*pfunc) ();

Ejercicios

1. Carga mediante el teclado una matriz entera de 4 filas y 3 columnas. Calcula y muestra en pantalla la suma de cada fila y de cada columna por medio de dos funciones de prototipos

```
int suma_fila (int fila);
int suma_columna (int columna);
```

2. Construye un programa que cree y muestre en pantalla un cuadrado mágico de orden 3. Un cuadrado mágico es aquél en el que todas las filas y columnas suman lo mismo. Para ello, se coloca el valor 1 en el medio de la 1ª fila. Los siguientes valores (2, 3, 4, ...) se sitúan en la fila de arriba, columna de la derecha, salvo que esté ocupada, en cuyo caso se coloca inmediatamente debajo. Se supone que las filas y columnas de los extremos son adyacentes.

	1	
3		
		2

	1	**6**
3	**5**	
4		2

8	1	6
3	5	**7**
4	**9**	2

3. Construye un programa que cree y muestre en pantalla un cuadrado mágico de orden **N** impar ($3 < N \leq 19$). El valor de **N** se leerá de la línea de órdenes (declarando adecuadamente la función **main**). Envía a pantalla un mensaje de error en cada uno de los siguientes casos:

- El valor de **N** está fuera de rango.
- No se ha dado valor a **N** en la línea de órdenes.
- Hay demasiados parámetros en la línea de órdenes.

En cualquiera de los casos finalizarás el programa después de enviado el mensaje. (NOTA: Utiliza la función **atoi()** de la biblioteca estándar).

4. Construye un programa que cargue del teclado dos matrices enteras **A** y **B** de orden 3x3, y que visualice la matriz producto **P = A · B**.

$$p_{ij} = \sum_{k=0}^{3} a_{ik} \cdot b_{kj}$$

La carga de matrices debes realizarla mediante una función a la que llamarás dos veces: una para cargar **A** y otra para cargar **B**. También calcularás los elementos $\mathbf{p_{ij}}$ mediante una función.

5. Construye dos funciones similares a **strlwr()** y **strupr()** que actúen también sobre los caracteres ñ, ç, letras acentuadas, etc. Se llamarán **strminus()** y **strmayus()**.

6. Desarrolla las siguientes funciones:

- Una función llamada **burbuja()** que ordene ascendentemente un vector de 100 elementos enteros por el método de la burbuja.
- Una función llamada **seleccion()** que ordene ascendentemente un vector de 100 elementos enteros por el método de selección.
- Una función llamada **insercion()** que ordene ascendentemente un vector de 100 elementos enteros por el método de inserción.
- Una función llamada **shell()** que ordene ascendentemente un vector de 100 elementos enteros por el método shell.
- Una función llamada **quick()** que ordene ascendentemente un vector de 100 elementos enteros por el método de ordenación rápida o Quick Sort.

Construye un programa que cargue un vector ordenado descendentemente

a[0] = 99 a[1] = 98 a[2] = 97 ... a[99] = 0

y que determine, mediante llamadas a las funciones anteriores, qué método de ordenación es más rápido. Para medir el tiempo utiliza la función de la biblioteca estándar **biostime()** cuyo prototipo está en **bios.h**.

7. Construye dos funciones de prototipos

```
char *Copia (char *cad1, const char *cad2);
char *Concatena (char *cad1, const char *cad2);
```

que realicen las mismas operaciones, respectivamente, que las funciones de biblioteca estándar **strcpy()** y **strcat()**. Haz dos versiones de cada función: una con índices y otra con punteros.

8. Construye una función de prototipo

```
char *Insertar (char *cad, int car, int pos);
```

que inserte el carácter *car* en la posición *pos* de la cadena *cad*. La función debe devolver la dirección de *cad*. Haz dos versiones de la función: una con índices y otra con punteros.

9. Construye una función de prototipo

```
char *Elimina (char *cad, int pos);
```

que elimine de *cad* el carácter situado en la posición *pos*. La función debe devolver la dirección de *cad*. Haz dos versiones de la función: una con índices y otra con punteros.

10. Construye una función de prototipo

char *Substr (char *orig, int desde, int n, char *dest);

que almacene en la cadena *dest* la subcadena de *cad* que comienza en la posición *desde* y tiene *n* caracteres. La función debe devolver la dirección de *dest*. Haz dos versiones de la función: una con índices y otra con punteros.

11. Crea un vector de 1000 elementos enteros ordenados descendentemente:

x[0]	x[1]	x[2]	...	x[998]	x[999]
999	998	997	...	1	0

y ordénalo ascendentemente por el método de la burbuja. Haz dos versiones del programa: una con índices y otra con punteros. Mide la eficiencia de cada método utilizando la función **clock()** de la biblioteca estándar.

12. Utilizando punteros carga un vector de 50 elementos enteros (Genera los valores aleatoriamente mediante las funciones de la biblioteca estándar **random()** y **randomize()**). Posteriormente muestra cuál es el valor mayor, su posición en el vector y la posición de memoria que ocupa.

13. Haz un programa que cree un vector preparado para almacenar 50 datos enteros que funcione con estructura de pila. El programa ejecutará las siguientes acciones:

1. Mostrar el contenido de la pila.
2. Introducir un dato en la pila.
3. Sacar un dato de la pila.
4. Fin del programa.

El programa presentará este menú y realizará las acciones deseadas mediante funciones agrupadas en una matriz.

8
Otros tipos de datos

Introducción

El Lenguaje C permite al usuario crear nuevos tipos de datos mediante 5 herramientas:

- La sentencia **typedef**, que permite dar nuevos nombres a tipos de datos que ya existen.
- Las **estructuras**, agrupaciones de variables de distinto tipo bajo un nombre común.
- Los **campos de bits**, que son una variante de las estructuras, y que permiten el acceso individual a los bits de una palabra.
- Las **uniones**, que permiten asignar la misma zona de memoria para variables diferentes.
- Las **enumeraciones**, que son listas de símbolos.

Estudiaremos cada una de ellas a continuación.

Tipos definidos por el usuario

Mediante la palabra reservada **typedef** podemos dar nuevos nombres a tipos de datos que ya existen. La sintaxis es la siguiente:

typedef *tipo nombre*;

donde *tipo* es un tipo de dato y *nombre* es el nuevo nombre para el tipo de dato anterior. Por ejemplo:

```
typedef register int CONTADOR;
```

hace que C reconozca **CONTADOR** como un tipo de dato idéntico a **register int**. Así, después de la definición anterior podemos declarar variables del nuevo tipo mediante sentencias como

```
CONTADOR i, j;
```

que declara dos variables **i**, **j** del tipo **CONTADOR**, es decir, **register int**. Además, este tipo de definiciones no anulan el tipo anterior, por lo que podemos seguir declarando variables del tipo **register int** que convivan con declaraciones del tipo **CONTADOR**.

Mediante esta sentencia podemos usar nombres más cómodos para los tipos de datos, y hacer que el código fuente sea más claro.

Estructuras

Una estructura es un conjunto de variables de igual o distinto tipo, agrupadas bajo un nombre común. A esas variables se les denomina *campos de la estructura*. Las estructuras se declaran mediante la palabra reservada **struct**. Hay varias formas de declarar una estructura, pero la más general es la siguiente:

```
struct nombre_estructura {
    tipo   variable1;
    tipo   variable2;
    ...
    ...
    tipo   variableN;
} ;
```

Veamos un ejemplo. La siguiente declaración

```
struct FICHA {
    char nombre[40];
    int edad;
    float altura;
};
```

define una estructura llamada **FICHA** con tres campos de distinto tipo: **nombre**, **edad** y **altura**.

Hemos de tener en cuenta que la declaración anterior sólo define la forma de la estructura, pero no declara ninguna variable con dicha forma. Por así, decirlo, hemos definido una plantilla de 3 campos, pero no hemos aplicado dicha plantilla a ninguna variable. Para hacerlo hay que escribir sentencias como

```
struct FICHA registro;
```

que declara una variable llamada **registro** con la estructura **FICHA**, es decir, **registro** es una variable compuestas por 3 campos: **nombre**, **edad** y **altura**. Para

acceder a cada uno de los campos se utiliza el operador punto (**.**) de la siguiente forma:

```
strcpy (registro.nombre, "José García");
registro.edad = 38;
registro.altura = 1.82;
```

Por supuesto, pueden declararse varias variables con la misma estructura:

```
struct FICHA var1, var2;
```

que son dos variables con la estructura **FICHA**. Los campos de cada una de ellas, como por ejemplo **var1.edad** y **var2.edad** son dos variables distintas que ocupan posiciones de memoria diferentes.

También pueden declararse las variables a la vez que se define la estructura:

```
struct FICHA {
    char nombre[40];
    int edad;
    float altura;
} var1, var2;
```

que declara dos variables **var1** y **var2** con la estructura **FICHA**. Este tipo de declaraciones no impide que puedan declararse más variables con esa estructura:

```
#include <...

struct FICHA {
    char nombre[40];
    int edad;
    float altura;
} var1, var2;

void Funcion (void);

void main (void)
{
    struct FICHA var3;
    ...
    ...
}

void Funcion (void)
{
    struct FICHA var4;
    ...
    ...
}
```

En las sentencias anteriores se declaran 4 variables con la estructura **FICHA**: **var1** y **var2**, globales; **var3**, local a **main()**; y **var4**, local a **Funcion()**.

Pueden hacerse declaraciones de una estructura sin darle nombre. Por ejemplo:

```
struct {
    char nombre[40];
    int edad;
    float altura;
} var1, var2;
```

declara las variables **var1** y **var2** con la estructura indicada. El problema que plantea este tipo de declaraciones es que, al no tener nombre la estructura, no es posible declarar otras variables con esa estructura.

También es habitual definir las estructuras con un nuevo tipo. Por ejemplo:

```
typedef struct {
    char nombre[40];
    int edad;
    float altura;
} MIESTR;
```

Ahora podemos hacer uso del nuevo tipo para declarar variables mediante sentencias como:

```
MIESTR var1, var2;
```

Fijémonos que, debido a la sentencia **typedef**, **MIESTR** no es una variable, sino un tipo de dato.

Inicialización de estructuras

Es posible inicializar variables con estructura en el momento en que son declaradas. Por ejemplo:

```
#include <...

struct FICHA {
    char nombre[40];
    int edad;
    float altura;
};

void main (void)
{
    struct FICHA registro = { "José García", 38, 1.82 };
    ...
```

que produce el mismo efecto que las sentencias

```
strcpy (registro.nombre, "José García");
registro.edad = 38;
registro.altura = 1.82;
```

Anidamiento de estructuras

Es posible el anidamiento de estructuras, en el sentido de que un campo de una estructura puede ser, a su vez, una estructura. Veámoslo con un ejemplo:

```
struct FECHA {
    int dia;
    int mes;
    int anyo;
};

struct CUMPLE {
    char nombre[40];
    struct FECHA nacim;
};

struct CUMPLE aniversario;
```

La variable **aniversario** así definida tiene los siguientes campos:

aniversario.nombre
aniversario.nacim.dia
aniversario.nacim.mes
aniversario.nacim.anyo

El anidamiento de estructuras puede hacerse a más profundidad, en cuyo caso el acceso a los campos de la variable se hace utilizando sucesivamente operadores punto.

Matrices de estructuras

Uno de los usos más habituales de las estructuras son las matrices. Para declarar una matriz de una estructura se hace como para cualquier otra variable. Por ejemplo,

```
struct FICHA {
    char nombre[40];
    int edad;
    float altura;
};
...
...
struct FICHA vector[100];
```

declara un vector de 100 elementos llamado **vector**. Cada uno de los elementos **vector[i]**, está formado por los 3 campos definidos en la estructura. Los campos del elemento **i** de **vector** son:

vector[i].nombre
vector[i].edad
vector[i].altura

Si alguno de los campos de la estructura es una matriz (como es el caso del campo **nombre**) puede accederse a cada elemento de la forma siguiente:

vector[10].nombre[3]

que se refiere al carácter 3 del campo **nombre** del elemento **vector[10]**.

Paso de estructuras a funciones

Un campo de una estructura puede pasarse como argumento de una función del mismo modo que cualquier variable simple. El siguiente pograma muestra en pantalla, por medio de una función, un campo de una estructura.

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

struct FICHA {
   char nombre[40];
   int edad;
   float altura;
};

void Funcion (int);

void main (void)
{
   struct FICHA registro = { "José García", 37, 1.82 };

   clrscr ();
   Funcion (registro.edad);
}

void Funcion (int n)
{
   printf ("\nEdad: %d", n);
}
```

También se puede pasar la dirección de un campo de una estructura. Un programa similar al anterior, pero manejando la dirección del campo, es el siguiente:

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

struct FICHA {
   char nombre[40];
   int edad;
   float altura;
};
void Funcion (int *);

void main (void)
{
```

```
  struct FICHA registro = { "José García", 37, 1.82 };

  clrscr ();
  Funcion (&registro.edad);
}

void Funcion (int *n)
{
  printf ("\nEdad: %d", *n);
}
```

Por último, también podemos pasar la estructura completa como argumento de una función. El siguiente ejemplo muestra cómo hacerlo:

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

struct FICHA {
  char nombre[40];
  int edad;
  float altura;
};

void Funcion (struct FICHA);

void main (void)
{
  struct FICHA registro = { "José García", 37, 1.82 };

  clrscr ();
  Funcion (registro);
}

void Funcion (struct FICHA n)
{
  printf ("\nFuncion3");
  printf ("\nNombre: %s", n.nombre);
  printf ("\nEdad: %d", n.edad);
  printf ("\nAltura: %f", n.altura);
}
```

Punteros a estructuras

Cuando las estructuras son complejas el paso de la estructura completa a una función puede ralentizar los programas debido a la necesidad de introducir y sacar cada vez todos los elementos de la estructura en la pila. Es por ello por lo que es conveniente recurrir a pasar sólo la dirección de la estructura, utilizando punteros.

Un puntero a una estructura se declara del mismo modo que un puntero a cualquier otro tipo de variable, mediante el operador *****. Por ejemplo, la sentencia

```
struct MIESTR *p, var;
```

declara un puntero **p** a una estructura **MIESTR** y una variable **var** con esa estructura. Después de una declaración como la anterior, puede hacerse una asignación como la siguiente:

```
p = &var;
```

que asigna a **p** la dirección de la variable **var**.

Para acceder a los campos de la estructura mediante el puntero se hace

(*p).campo

Sin embargo esta sintaxis es antigua y está fuera de uso. En su lugar se utiliza el operador flecha (->), formado por el guión (-) y el símbolo mayor que (>).

p -> campo

El siguiente programa muestra en pantalla los campos de una estructura a través de un puntero que contiene su dirección.

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

struct FICHA {
  char nombre[40];
  int edad;
  float altura;
};

void main (void)
{
  struct FICHA *p, registro = { "José García", 37, 1.82 };

  clrscr ();
  p = &registro;
  printf ("\nNombre: %s", p -> nombre);
  printf ("\nEdad: %d", p -> edad);
  printf ("\nAltura: %f", p -> altura);
}
```

Campos de bits

Los campos de bits son un caso particular de estructuras que permiten acceder a los bits individuales de una palabra. La sintaxis que define un campo de bits es la siguiente:

struct *nombre_estructura* **{**
    *tipo variable1***:** *ancho1***;**
    *tipo variable2***:** *ancho2***;**
    ...

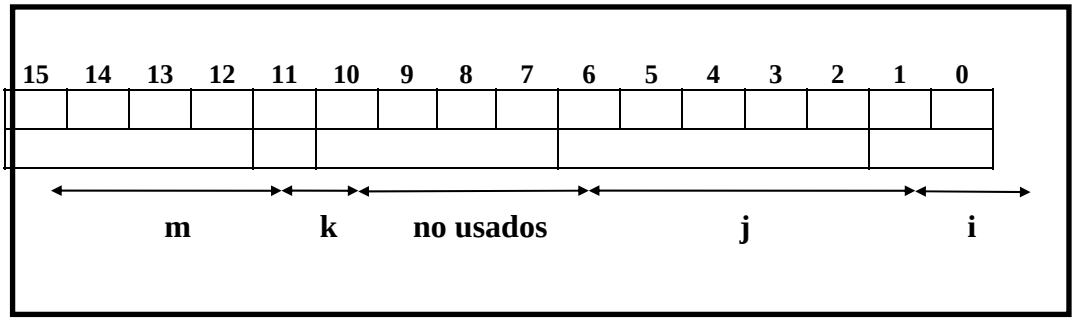
```
    ...
    tipo variableN:anchoN;
};
```

donde *tipo* es uno de los tipos **char**, **unsigned char**, **int** o **unsigned int**, y *anchoN* es un valor de 0 a 16, que representa el ancho en bits del campo de bits *variableN*. Si no se pone nombre al campo de bits, los bits especificados en *ancho* se reservan, pero no son accesibles.

El siguiente ejemplo muestra una declaración de campo de bits:

```
struct MIESTR {
    int i: 2;
    unsigned j: 5;
    int  : 4;
    int k: 1;
    unsigned m: 4;
};

struct MIESTR var;
```

Esta declaración se corresponde con el siguiente diseño de la palabra:



Los campos de bits son útiles para almacenar variables lógicas en un byte, para codificar los bits de un dispositivo, etc. El acceso a los campos de bits se realiza del mismo modo que a los campos de cualquier estructura. Tan sólo hay que tener en cuenta un par de restricciones con los campos de bits: no se permiten matrices y no se puede trabajar con direcciones.

Los campos de bits pueden presentar algún problema respecto de la portabilidad de los programas de una máquina a otra. Aquí se ha supuesto que los campos funcionan de izquierda a derecha, pero eso puede cambiar en otras máquinas, con lo que la interprestación de los diferente campos cambia.

Uniones

Una unión es una agrupación de variables que ocupan la misma posición de memoria. Se declaran de modo similar a las estructuras:

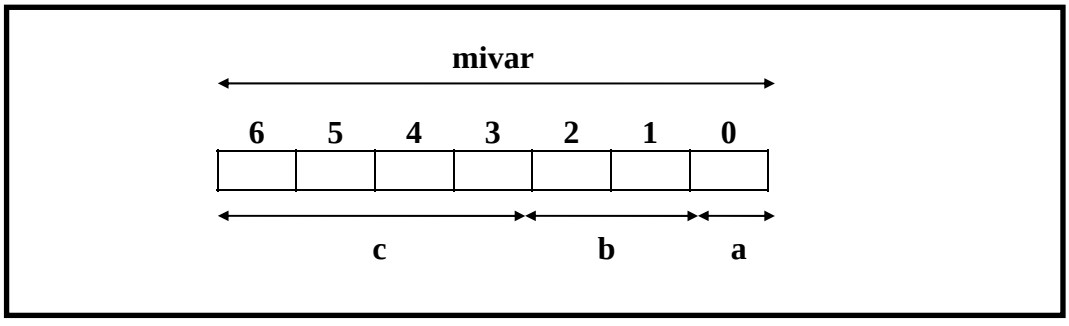
```
union nombre_unión {
    tipo variable_1;
    tipo variable_2;
    ...
    ...
    tipo variable_N;
};
```

Cuando se declara una unión, las variables *variable_1*, *variable_2*, ..., *variable_N* ocupan la misma posición de memoria. El compilador reserva el tamaño suficiente para almacenar la variable más grande.

Es importante advertir una diferencia notable entre las estructuras (**struct**) y las uniones (**union**). En las primeras cada campo ocupa una posición de memoria propia, y cuando se almacena un valor en uno de los campos, no influyen en el resto. Por el contrario, en las uniones todas las variables de la unión tienen asignada la misma posición de memoria, lo cual implica que cuando se almacena un dato en una de ellas, influye en todas las demás. Aclaremos esto con un ejemplo sencillo. Sea la estructura

```
struct MIESTR {
    char a;
    int b;
    float c;
};

struct MIESTR mivar;
```

Esta declaración asigna 7 bytes de memoria para la variable **mivar**: uno para el campo **a**, dos más para el campo **b**, y otros cuatro para el campo **c**.



Si hacemos una asignación como

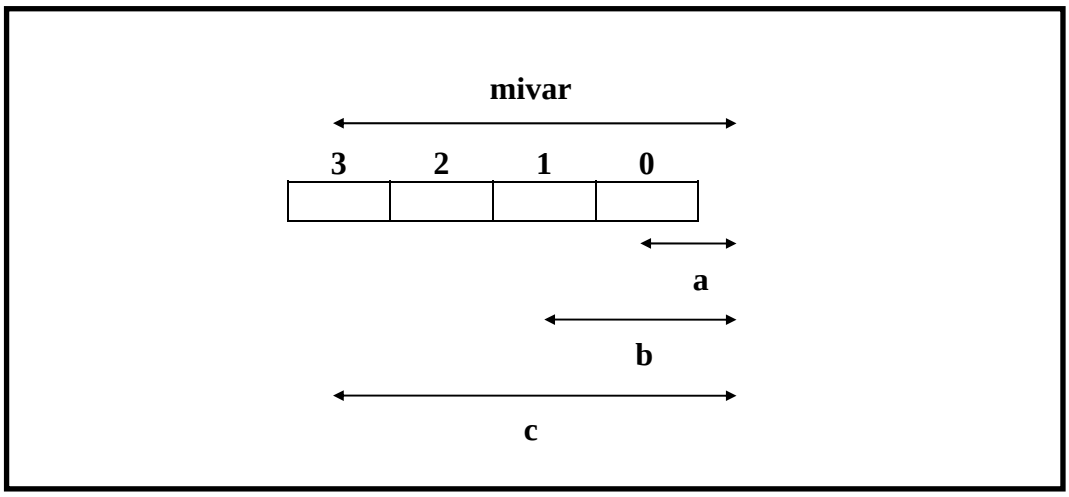
```
mivar.b = 120;
```

no afecta a **mivar.a** ni a **mivar.c**, pues cada uno de ellos tiene asignadas direcciones distintas de memoria.

Sin embargo, sea la union

```
union MIUNION {
    char a;
    int  b;
    float c;
};

union MIUNION mivar;
```

Esta declaración asigna 4 bytes de memoria para **mivar** (el campo más grande de la unión es **c**, de 4 bytes).



Si ahora hacemos una asignación como

```
mivar.b = 120;
```

afecta tanto a **mivar.a** como a **mivar.c**.

Veamos un ejemplo. Las siguientes declaraciones utilizan las estructuras y uniones para almacenar datos de 100 alumnos y profesores de un Centro de Enseñanza. Se almacena el tipo (A: alumno, P: profesor), nombre, edad, dirección y teléfono. Si es alumno se almacena, además, el grupo al que pertenece, el número de asignaturas que cursa y si es o no repetidor. Si es profesor se almacena el número de registro y el cargo que desempeña.

```
struct ALUMNO {
    char grupo[15];
    int asignat;
    char repite;
};

struct PROFESOR {
    char nrp[16];
    char cargo[21];
};

union AL_PR {
    struct ALUMNO al;
    struct PROFESOR pr;
};

struct DATOS {
    char tipo;
    char nombre[40];
    int edad;
    char direccion[40];
    char telefono[8];
    union AL_PR ambos;
} personal[100];
```

El siguiente segmento de progama muestra los datos de la matriz **personal**.

```
for (i = 0; i < 100; i++) {
    printf ("\nNombre: %s", personal[i].nombre);
    printf ("\nEdad: %d", personal[i].edad);
    printf ("\nDirección: %s", personal[i].direccion);
    printf ("\nTeléfono: %s", personal[i].telefono);
    if (personal[i].tipo == 'A') {
        printf ("\nALUMNO");
        printf ("\nGrupo: %s", personal[i].ambos.al.grupo);
        printf ("\nNº de Asignaturas: %d", personal[i].ambos.al.asignat);
        printf ("\nRepite: %d", personal[i].ambos.al.repite); }
    else {
        printf ("\nPROFESOR");
        printf ("\nN.R.P.: %s", personal[i].ambos.pr.nrp);
        printf ("\nCargo: %s", personal[i].ambos.pr.cargo);
    }
}
```

Combinando estructuras, uniones y campos de bits podemos definir variables de 16 bits con acceso por bit, byte o palabra.

```
struct BITS {
unsigned bit0: 1;
unsigned bit1: 1;
unsigned bit2: 1;
unsigned bit3: 1;
unsigned bit4: 1;
unsigned bit5: 1;
unsigned bit6: 1;
```

```
unsigned bit7: 1;
unsigned bit8: 1;
unsigned bit9: 1;
unsigned bit10: 1;
unsigned bit11: 1;
unsigned bit12: 1;
unsigned bit13: 1;
unsigned bit14: 1;
unsigned bit15: 1;
};

struct BYTES {
unsigned byte0: 8;
unsigned byte1: 8;
};

union PALABRA {
int x;
struct BYTES byte;
struct BITS bit;
} dato;
```

Ahora, mediante **dato** podemos acceder a la palabra completa (**dato.x**), a uno de los dos bytes (**dato.byte.byteN**), o a bits individuales (**dato.bit.bitN**).

Vamos a utilizar la función de la biblioteca estándar **biosequip()** en un programa quehace uso de esta capacidad. La funcón **biosequip()** devuelve una palabra con la siguiente información:

bits 14-15 Número de impresoras paralelo
bit 13 Impresoria serie instalada
bit 12 Joystick instalado
bits 9-11 Número de puertos COM
bit 8 Chip DMA instalado (0: SÍ, 1: NO)
bits 6-7 Nº de disqueteras (si **bit0 = 1**)
00: 1 disquetera
01: 2 disqueteras
10: 3 disqueteras
11: 4 disqueteras
bits 4-5 Modo inicial de vídeo
00: No usado
01: 40x25 BN, adaptador color
10: 80x25 BN, adaptador color
11: 80x25 BN, adaptador monográfico
bits 2-3 RAM en placa base
00: 16k
01: 32k
10: 48k
11: 64k
bit 1 Coprocesador matemático 80x87 instalado
bit 0 Arranque desde disquete

El siguiente programa presenta en pantalla la información proporcionada por **biosequip()**.

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

struct BIOSEQUIP {
  unsigned arranque: 1;
  unsigned cop80x87: 1;
  unsigned RAM_base: 2;
  unsigned video: 2;
  unsigned discos: 2;
  unsigned DMA: 1;
  unsigned puertos: 3;
  unsigned juegos: 1;
  unsigned serie: 1;
  unsigned paralelo: 2;
};

union EQUIPO {
  int palabra;
  struct BIOSEQUIP bits;
};

void Presentar (void);

void main (void)
{
  union EQUIPO bios;

  bios.palabra = biosequip ();

  clrscr ();
  Presentar ();

  gotoxy (31, 1);
  if (bios.bits.cop80x87) puts ("SI");
  else puts ("NO");

  gotoxy (31, 2);
  printf ("%d Kb", (bios.bits.RAM_base + 1) * 16);

  gotoxy (31, 3);
  switch (bios.bits.video) {
    case 1:  puts ("40x25 Blanco y Negro, Adaptador Color");
             break;
    case 2:  puts ("80x25 Blanco y Negro, Adaptador Color");
             break;
    case 3:  puts ("80x25 Blanco y Negro, Adaptador Monocromo");
  }

  gotoxy (31, 4);
  if (bios.bits.arranque) printf ("%d", bios.bits.discos + 1);
  else puts ("0");

  gotoxy (31, 5);
  if (bios.bits.DMA) puts ("NO");
  else puts ("SI");
```

```
  gotoxy (31, 6);
  printf ("%d", bios.bits.puertos);

  gotoxy (31, 7);
  if (bios.bits.juegos) puts ("SI");
  else puts ("NO");

  gotoxy (31, 8);
  printf ("%d", bios.bits.serie);

  gotoxy (31, 9);
  printf ("%d\n", bios.bits.paralelo);
}

void Presentar (void)
{
  puts ("Coprocesador Matemático ..... ");
  puts ("RAM en placa base ..... ");
  puts ("Modo inicial de vídeo ..... ");
  puts ("Nº unidades de disquete ..... ");
  puts ("Chip DMA ..... ");
  puts ("Nº de puertos COM ..... ");
  puts ("Joystick ..... ");
  puts ("Impresora serie ..... ");
  puts ("Impresoras paralelo ..... ");
}
```

Enumeraciones

Son grupos de constantes agrupadas de modo similar a las estructuras y que permiten controlar el rango de una variable. La sintaxis que permite definir una enumeración es:

enum *nombre_enumeración* **{**
*constante1***;**
*constante2***;**
...
...
*constanteN***;**
} *variable***;**

siendo *nombre_enumeración* el identificador que da nombre a la enumeración, y *variable* la variable que puede tomar los valores definidos para *constante1*, *constante2*, ..., *constanteN*. Estas constantes asumen los valores 0, 1, 2, ..., N. Por ejemplo:

```
enum COLORES {
NEGRO;
AZUL;
```

```
VERDE;
CYAN;
ROJO;
MORADO;
AMARILLO;
BLANCO;
} color;
```

Para una enumeración como la anterior, la sentencia

```
printf ("%d %d", AZUL, MORADO);
```

mostraría en pantalla los valores 1 y 5. También son posibles sentencias del tipo

```
switch (color) {
case NEGRO:  ...
             break;
case AZUL:   ...
...
...
default: ...
}
```

Es importante no olvidar que los valores definidos con los identificadores **NEGRO**, **AZUL**, etcétera, no son variables sino constantes.

Es posible modificar el rango de valores de una enumeración, inicializando las constantes en el momento de la declaración. Así, la enumeración

```
enum VALORES {
CTE1;
CTE2;
CTE3 = 25;
CTE4;
CTE5 = 31;
};
```

define las constantes de la enumeración con los siguientes valores

```
CTE1: 0
CTE2: 1
CTE3: 25
CTE4: 26
CTE5: 31
```

Ejercicios

1. Deseamos hacer un listado de precios de los artículos de la empresa QUIEBRA,S.A. Los datos a procesar para cada artículo son los siguientes:

- **Código del artículo**: Cadena de 3 caracteres.
- **Descripción**: Cadena de 40 caracteres.
- **Componentes**: Matriz de 5 elementos. Cada elemento de esta matriz contendrá los siguientes campos:
    1. **Código del componente**: Cadena de 8 caracteres.
    2. **Cantidad**: Entero entre 1 y 100.
    3. **Precio unitario**: Entero entre 500 y 5000.

Construye un programa que realice las siguientes tareas:

a) Capturar por teclado los datos de los artículos, almacenándolos en una matriz (máximo, 25 artículos).
b) Imprimir los datos de la matriz con el formato siguiente:

QUIEBRA, S.A. Listado de precios **Página:XXX**

		COMPONENTES			
Cod	**Descripción**	**Código**	**Cant**	**Precio**	**Importe**
xxx	**xxxxxxxxxxxxxxxxxxxxxxxxxx**	**xxxxxxxx**	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
				Total:	**xxxxxx**
xxx	**xxxxxxxxxxxxxxxxxxxxxxxxxx**	**xxxxxxxx**	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**
		xxxxxxxx	**xxxxx**	**xxxxxx**	**xxxxxx**

				Total:	**xxxxxxx**
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...

Imprime 5 artículos en cada página.

2. La función #2 de la INT 17h devuelve el estado de un puerto paralelo especificado con el registro DX. Los requerimientos previos de la función son:

AH = 2
DX = Nº del puerto paralelo
0 - LPT1:
1 - LPT2:
...
...

La función devuelve en AH el estado del puerto. Los bits de AH deben interpretarse como sigue:

bit 0 La impresora no respondió en el tiempo previsto
bits 1-2 No usados
bit3 Error de E/S
bit4 Impresora seleccionada
bit5 Sin papel
bit6 Reconocimiento de impresora (Acuse de recibo ACK)
bit7 Impresora libre

Construye un programa que muestre esta información para el primer puerto paralelo (LPT1:).

9
Asignación dinámica de memoria

Almacenamiento estático y dinámico

La forma convencional de almacenar las variables en memoria se denomina **almacenamiento estático**: al principio del programa se declaran las variables y se reserva la cantidad de memoria necesaria. Este es el sistema que hemos utilizado hasta el momento en todos los ejercicios y ejemplos. Con este método cuando un programa utiliza una matriz, previamente decidimos el tamaño máximo que necesita y declaramos la matriz de acuerdo a ese tamaño máximo. Por ejemplo, si el programa usa una matriz bidimensional de enteros y suponemos que el máximo de filas y de columnas que precisará es de 100x100, haremos una declaración como

```
int matriz[100][100];
```

con lo que estaremos reservando 100 x 100 x 2 = 20000 bytes de memoria al inicio del programa (en el supuesto de que estén disponibles) y permanecerán reservados durante todo el programa, independientemente de que se usen o no para almacenar valores.

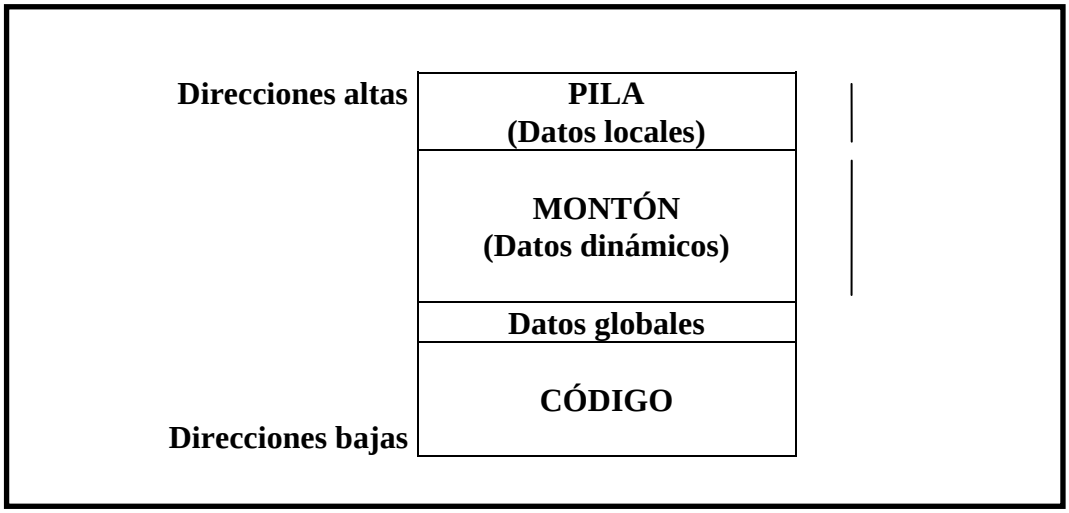
Este tipo de almacenamiento presenta algún inconveniente. Fijémonos en el Ejercicio 3 del Capítulo 7 (página 118) en el que se construye un cuadrado mágico. En ese programa se trabaja con una matriz cuya dimensión puede ir desde 3x3 hasta 19x19. Para el cuadrado de dimensión mínima se necesitan 18 bytes de memoria; cuando se usa la máxima dimensión son necesarios 722 bytes. Pero, independientemente del cuadrado mágico que se desee construir, hemos de declarar la matriz del tamaño máximo, y cuando se construya el cuadrado de ordenes inferiores a 19 estaremos malgastando cierta cantidad de memoria. En este programa esto no es demasiado importante, pero sí lo es para otros que usen matrices mayores. Si el ejercicio referido tratase con cuadrados de órdenes muy grandes, la cantidad de memoria malgastada podría ser muy importante. Basta fijarse en que una matriz de enteros de orden 500x500 necesita, aproximadamente, 0.5 Mb de almacenamiento estático.

La alternativa ideal sería que el programa pudiese solicitar al sistema la cantidad de memoria precisa en cada caso. Para el cuadrado mágico se trataría de

pedir al sistema 18 bytes para el orden 3x3 y 722 bytes cuando se construya el cuadrado de orden 19x19. Esto se consigue con la **asignación dinámica de memoria**, mediante la cual el programa va requiriendo al sistema la memoria que necesita y devolviendo la que ya no es necesaria. Para ello se utilizan las funciones **malloc()** y **free()**, que estudiaremos en el próximo apartado.

La memoria asignada dinámicamente se toma de una zona denominada **montón** (heap) y su tamaño es variable, dependiendo del modelo de memoria[1] usado para compilar el programa.

Básicamente, la forma de situar en la memoria datos y código de un programa atiende al siguiente esquema:



El tamaño para las zonas de código y de datos de vida global permanece constante durante todo el programa. La pila crece hacia abajo y en ella se almacenan las variables de ámbito local utilizadas por las funciones. El montón, también denominado **zona de almacenamiento libre**, crece hacia arriba, a medida que se asigna memoria dinámicamente a petición del programa. En algunos casos la pila puede solapar parte del montón. También puede ocurrir que el montón no pueda satisfacer todas las demandas de memoria del programa. Estos problemas pueden controlarse con las funciones de asignación dinámica de C.

Las funciones malloc() y free()

[1] Los *modelos de memoria* se estudian en el Capítulo 12.

Son las dos funciones básicas para la asignación dinámica de memoria. Con la función **malloc()** se solicita memoria del montón. Con la función **free()** se devuelve memoria al montón.

El prototipo para la función **malloc()** es

void *malloc (size_t *tama***);**

y está definido en **stdlib.h** y **alloc.h**. Esta función asigna un bloque de *tama* bytes del montón. Si la cantidad de memoria solicitada está disponible, **malloc()** devuelve un puntero a la zona de memoria asignada. En caso contrario, devuelve un puntero nulo. El tipo **void *** indicado en el prototipo quiere decir que el puntero devuelto por **malloc()** puede (y debe) transformarse a cualquier tipo. El siguiente ejemplo solicita memoria para 100 datos enteros:

```
int *bloque;
...
...
bloque = malloc (200);
if (!bloque) {
puts ("Memoria insuficiente");
Funcion_error ();
}
```

Si la función **malloc()** tiene éxito, el puntero *bloque* tendrá la dirección de una zona de 200 bytes del montón. En caso contrario debemos finalizar el programa o ejecutar una rutina de error. Hay que poner especial cuidado en no trabajar con punteros nulos, puesto que ello provoca, generalmente, la caida del sistema.

Aunque el ejemplo anterior es correcto, se recomienda hacer la llamada a **malloc()** de la forma siguiente:

```
int *bloque;
...
...
bloque = (int *) malloc (100 * sizeof (int));
if (!bloque) {
puts ("Memoria insuficiente");
Funcion_error ();
}
```

pues asegura la transportabilidad del código.

La función **free()** tiene como prototipo

void free (void * *bloque* **);**

y precisa la inclusión del archivo de cabecera **alloc.h**. Esta función devuelve al montón el bloque de memoria apuntado por *bloque*, siendo *bloque* un puntero proporcionado por una llamada previa a **malloc()**. La memoria devuelta por **free()** puede volver a ser asignada por otra llamada a **malloc()**.

El programa de la página siguiente ilustra el uso de ambas funciones. En él se solicita memoria para almacenar una cadena de caracteres de longitud variable. Sólo se usará la memoria necesaria dada por la longitud de la cadena.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <process.h>

void main (void)
{
    char *cadena;
    int N;

    clrscr ();
    puts ("EJEMPLO DE ASIGNACIÓN DINÁMICA DE MEMORIA");
    printf ("Longitud de la cadena: ");
    scanf ("%d", &N);
    getchar ();                          //Elimina el Intro de la entrada

    cadena = (char *) malloc (N + 1);    //N + 1 para incluir el nulo final
    if (!cadena) {
        puts ("\nMEMORIA INSUFICIENTE");
        exit (1); }
    else printf ("\nDirección asignada: %p", cadena);

    printf ("\nTeclea %d caracteres: ", N);
    gets (cadena);
    printf ("\nHas tecleado: %s", cadena);

    free (cadena);
}
```

Otras funciones de asignación dinámica de Turbo C son: **allocmem()**, **calloc()**, **coreleft()**, **farcalloc()**, **farmalloc()** y **realloc()**.

Matrices asignadas dinámicamente

El programa mencionado anteriormente del cuadrado mágico ilustra bastante bien una de las situaciones en que es conveniente usar programación dinámica: el programa maneja una matriz cuyo tamaño no puede determinarse a priori.

El siguiente programa resuelve el ejercicio del cuadrado mágico utilizando asignación dinámica de memoria. Aunque en este caso no es necesario, se mantiene la limitación del orden máximo a 19x19, por razones de presentación en

pantalla. Además se supone que las funciones **Intro()** y **Strdigit()** se encuentran en módulos objeto separados que se enlazarán posteriormente[2].

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <alloc.h>
#include <process.h>

void main (void)
{
    int *magico, *p;
    int i, j, ant_i, ant_j, orden, tope;
    register int valor;
    char N[3];

    clrscr ();
    puts ("CUADRADO MÁGICO DE ORDEN N IMPAR (3 a 19)");
    do {
        do {
            printf ("\nValor de N: ");
            Intro (wherey (), wherex (), 2, N);
        } while (!Strdigit (N));
        orden = atoi (N);
    } while (orden < 3 || orden > 19 || !(orden % 2));
    tope = orden * orden;

    magico = (int *) malloc (tope * sizeof (int));
    if (!magico) {
        puts ("\nMemoria insuficiente");
        exit (1);
    }

    for (i = 0; i < orden; i++) {
        for (j = 0; j < orden; j++) {
            p = magico + i * orden + j;          //Dirección de magico[i][j]
            *p = 0;
        }
    }

    i = 0;
    j = orden / 2;
    for (valor = 1; valor <= tope; valor++) {
        p = magico + i * orden + j;              //Dirección de magico[i][j]
        if (*p) {
            i = ant_i + 1;
            j = ant_j;
            if (i > orden - 1) i = 0;
        }
        p = magico + i * orden + j;              //Dirección de magico[i][j]
        *p = valor;
```

[2] La compilación y enlazado independientes del entorno integrado de programación de Turbo C se estudian en el Capítulo 12.

```
        ant_i = i--;
        ant_j = j++;
        if (i < 0) i = orden - 1;
        if (j > orden - 1) j = 0;
    }

    clrscr ();
    for (i = 0; i < orden; i++) {
        for (j = 0; j < orden; j++) {
            p = magico + i * orden + j;          //Dirección de magico[i][j]
            printf ("%3d ", *p);
        }
        printf ("\n");
    }
    free (magico);
}
```

Colas dinámicas

Una cola es una estructura FIFO (**F**irst **I**n, **F**irst **O**ut), es decir, el primer elemento que entra en la estructura es el primero en salir. Las colas se ajustan muy bien al tipo de estructuras susceptibles de programarse dinámicamente, pues son estructuras que crecen y se contraen a medida que la información entra y sale de ellas.

Para programar dinámicamente este tipo de estructuras (como las pilas, los árboles binarios, etc.) es necesario recurrir a las **estructuras autoreferenciadas**. Una estructura autoreferenciada se define, básicamente, de la siguiente forma:

struct *nombre_estructura* **{**
*campo_de_nformación1***;**
*campo_de_información2***;**
...
...
*campo_de_informaciónN***;**
struct *nombre_estructura enlace1***;**
struct *nombre_estructura enlace2***;**
...
...
struct *nombre_estructura enlaceN***;**
};

es decir, hay uno o varios campos en los que se almacena la información que se desee, y uno o varios campos de enlace con otros elementos. Para el caso de una cola dinámica de números enteros cada elemento debe enlazar con el siguiente mediante una estructura autoreferenciada como la siguiente:

```
struct COLA {
```

```
int dato;
struct COLA *enlace;
};
```

siendo **dato** el número entero que se almacena en la cola, y **enlace** un puntero a la estructura del siguiente elemento de la cola.

El siguiente programa gestiona una cola dinámica de números enteros positivos. El funcionamiento del programa se basa en un menú de opciones y en dos funciones, **Introducir()** y **Sacar()**, que controlan la entrada/salida de los datos en la cola.

La función **Introducir()**realiza las siguientes operaciones:

- Llama a **malloc()** para solicitar memoria para el nuevo elemento. En caso de que no haya memoria disponible, la función devuelve un valor 0.
- Almacena el dato en la dirección asignada mediante **malloc()**.
- Recorre toda la cola desde el principio para localizar el último elemento, a partir del cual debe incluirse el nuevo.
- Enlaza el elemento que antes era el último con el nuevo elemento introducido en la cola.
- Devuelve un valor 1.

La función **Sacar()** efectúa las siguientes operaciones:

- Comprueba si la cola está vacía, en cuyo caso devuelve un valor -1.
- Obtiene la información del primer elemento de la cola.
- Hace que el segundo elemento de la cola sea el primero, apuntándolo con el puntero del elemento que se saca.
- Devuelve al montón mediante **free()** la memoria ocupada por el elemento que se saca.
- Devuelve el valor extraído.

El programa completo se muestra a continuación:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>

struct COLA {
    int dato;
    struct COLA *enlace;
};

int Menu (void);
int Introducir (int n);
int Sacar (void);

struct COLA *primero;
```

```
void main (void)
{
    int valor, opcion, st;
    primero = 0;
    while ((opcion = Menu ()) != 3) {
        switch (opcion) {
            case 1:  printf ("\nValor a introducir en la cola: ");
                     scanf ("%d", &valor);
                     if (!Introducir (valor)) {
                         puts ("\nMemoria agotada. Pulse una tecla ...");
                         while (!getch ());
                     }
                     break;
            case 2:  valor = Sacar ();
                     if (valor < 0) puts ("\nCola vacía");
                     else printf ("\nValor obtenido: %d", valor);
                     printf ("\nPulse una tecla ...");
                     while (!getch ());
        }
    }
}

int Menu (void)
{
    int n;

    clrscr ();
    puts ("COLA DINÁMICA\n");
    puts ("1. Introducir");
    puts ("2. Sacar");
    puts ("3. Salir\n");

    do {
        n = getch ();
    } while (n < '1' || n > '3');

    return n - 48;
}

int Introducir (int n)
{
    struct COLA *nuevo, *actual, *anterior;

    nuevo = (struct COLA *) malloc (sizeof (struct COLA));
    if (!nuevo) return 0;

    nuevo -> dato = n;
    nuevo -> enlace = 0;

    if (!primero) primero = nuevo;
    else {
        actual = primero;
        while (actual) {
            anterior = actual;
            actual = anterior -> enlace;
        }
        anterior -> enlace = nuevo;
    }
```

```
    return 1;
}
int Sacar (void)
{
    struct COLA *antprimero;
    int n;

    if (!primero) return -1;
    else {
        antprimero = primero;
        n = primero -> dato;
        primero = primero -> enlace;
        free (antprimero);
        return n;
    }
}
```

Ejercicios

1. Construye un programa que gestione una pila de enteros con asignación dinámica de memoria. El programa presentará un menú con las opciones

1. Introducir
2. Sacar
3. Salir

El programa dispondrá de funciones **Introducir()** y **Sacar()** similares a las utilizadas en el apartado "Colas dinámicas" del capítulo.

10
Ficheros

Canales y ficheros

Ya quedó dicho que toda la E/S en C se hace mediante funciones de biblioteca. Esto, que ya se estudió para la E/S por consola, se estudiará en este capítulo para la E/S sobre cualquier dispositivo, en particular sobre archivos de disco.

El sistema de E/S del ANSI C proporciona un intermediario entre el programador y el dispositivo al que se accede. Este intermediario se llama **canal** o **flujo** y es un buffer independiente del dispositivo al que se conecte. Al dispositivo real se le llama **archivo** o **fichero**. Por tanto, programa y archivo están conectados por medio de un canal y la misma función permite escribir en pantalla, impresora, archivo o cualquier puerto. Existen dos tipos de canales:

- **Canales de texto**: Son secuencias de caracteres. Dependiendo del entorno puede haber conversiones de caracteres (LF ⇔ CR + LF). Esto hace que el número de caracteres escritos/leídos en un canal pueda no coincidir con el número de caracteres escritos/leídos en el dispositivo.
- **Canales binarios**: Son secuencias de bytes. A diferencia de los canales de texto, en los canales binarios la correspondencia de caracteres en el canal y en el dispositivo es 1 a 1, es decir, no hay conversiones.

Un archivo es, por tanto, un concepto lógico que se puede asociar a cualquier cosa susceptible de realizar con ella operaciones de E/S. Para acceder a un archivo hay que relacionarlo con un canal por medio de una operación de apertura que se realiza mediante una función de biblioteca. Posteriormente pueden realizarse operaciones de lectura/escritura que utilizan búfers de memoria. Para estas operaciones hay disponible un amplio conjunto de funciones. Para desasociar un canal de un archivo es necesario realizar una operación de cierre. Hay 5 canales que se abren siempre que comienza un programa C. Son:

- **stdin** Canal estándar de entrada. Por defecto el teclado. (ANSI)
- **stdout** Canal estándar de salida. Por defecto la pantalla. (ANSI)
- **stderr** Canal estándar de salida de errores. Por defecto la pantalla. (ANSI)
- **stdaux** Canal auxiliar (canal serie COM1). (A partir de Turbo C v2.0)
- **stdprn** Canal para la impresora. (A partir de Turbo C v2.0)

Abrir y cerrar ficheros

Para poder manejar un fichero es necesario asociarle un canal. Esto se consigue mediante una operación de apertura que se realiza mediante la función de biblioteca **fopen()**, que devuelve un puntero al canal que se asocia. El prototipo de esta función está definido en **stdio.h** y es el siguiente:

FILE *fopen (const char *_nombre_, const char *_modo_);

Esta función abre el fichero *nombre* y le asocia un canal mediante el puntero a **FILE** que devuelve. El tipo de dato **FILE** está definido también en **stdio.h** y es una estructura que contiene información sobre el fichero. Consta de los siguientes campos:

```
typedef struct {
short level;            //nivel de ocupación del buffer
unsigned flags;         //indicadores de control
char fd;                //descriptor del fichero (nº que lo identifica)
char hold;              //carácter de ungetc()[1]
short bsize;            //tamaño del buffer
unsigned char *buffer;  //puntero al buffer
unsigned char *curp;    //posición en curso
short token;            //se usa para control
} FILE;
```

No debe tocarse ninguno de los campos de esta estructura a menos que se sea un programador muy experto. Cualquier cambio no controlado en los valores de esas variables, posiblemente dañaría el archivo. Si se produce cualquier error en la apertura del fichero, **fopen()** devuelve un puntero nulo y el fichero queda sin canal asociado.

Los valores posibles del parámetro *modo* se muestran en la tabla siguiente:

MODO	***DESCRIPCIÓN***
r	*Abre un fichero sólo para lectura. Si el fichero no existe **fopen()** devuelve un puntero nulo y se genera un error.*
w	*Crea un nuevo fichero para escritura. Si ya existe un fichero con este nombre, se sobreescribe, perdiéndose el contenido anterior.*
a	*Abre o crea un fichero para añadir. Si el fichero existe, se abre apuntando al final del mismo. Si no existe se crea uno nuevo.*
r+	*Abre un fichero para leer y escribir. Si el fichero no existe **fopen()** devuelve un puntero nulo y se genera un error. Si existe, pueden realizarse sobre él operaciones de lectura y de escritura.*
w+	*Crea un nuevo fichero para leer y escribir. Si ya existe un fichero con este nombre, se sobreescribe, perdiéndose el contenido anterior. Sobre el archivo pueden realizarse operaciones de lectura y de escritura.*
a+	*Abre o crea un fichero para leer y añadir. Si el fichero ya existe se abre apuntando al final del mismo. Si no existe se crea un fichero nuevo.*

[1] Esta función tiene como prototipo **int ungetc (int** *carácter*, **FILE *** *canal*); y devuelve *carácter* al canal de entrada *canal*, lo que permite que pueda volver a ser leido.

Para indicar si el canal asociado al fichero es de texto o binario, se añade al modo la letra **t** o **b**, respectivamente. Así, si *modo* es **rt**, se está abriendo el fichero en modo texto sólo para lectura, mientras que si *modo* es **w+b** se abrirá o creará un fichero en modo binario para lectura y para escritura[2].

Para cerrar un fichero y liberar el canal previamente asociado con **fopen()**, se debe usar la función **fclose()** definida en **stdio.h** y cuyo prototipo es

int fclose (FILE ****canal*);**

Esta función devuelve **0** si la operación de cierre ha tenido éxito, y **EOF** en caso de error. **EOF** es una macro definida en **stdio.h** de la siguiente forma:

```
#define   EOF   (-1)
```

El siguiente programa abre sólo para lectura un fichero de nombre DATOS.DAT y, posteriormente lo cierra.

```
#include <stdio.h>
#include <process.h>            //Para exit()

void main (void)
{
    FILE *f;
    int st;

    f = fopen ("DATOS.DAT", "rt");
    if (!f) {
        puts ("Error al abrir el fichero");
        exit (1);
    }

    //Aquí vendrían las operaciones sobre el fichero
    st = fclose (f);
    if (st) puts ("Error al cerrar el fichero");
}
```

Es habitual escribir la sentencia de apertura del fichero como sigue:

```
if (!(f = fopen ("DATOS.DAT", "rt"))) {
puts ("Error al abrir el fichero");
exit (1);
}
```

[2] Cuando no se indica ninguno de los modos **b** o **t** el modo de apertura está gobernado por la variable global **_fmode**. Si está inicializada con el valor **O_BINARY**, los ficheros se abren en modo binario. Si está inicializada como **O_TEXT**, se abren en modo texto. Estas constantes están definidas en **fcntl.h**.

Control de errores y de fin de fichero

Cada vez que se realiza una operación de lectura o de escritura sobre un fichero debemos comprobar si se ha producido algún error. Para ello disponemos de la función **ferror()** cuyo prototipo, definido en **stdio.h**, es

int ferror (FILE ****canal*);**

Esta función devuelve **0** si la última operación sobre el fichero se ha realizado con éxito. Hay que tener en cuenta que todas las operaciones sobre el fichero afectan a la condición de error, por lo que debe hacerse el control de error inmediatamente después de cada operación o, de lo contrario, la condición de error puede perderse. La forma de invocar esta función puede ser

```
if (ferror (f)) puts ("Error de acceso al fichero");
```

siendo **f** un puntero a **FILE**.

Se puede conseguir un mensaje asociado al último error producido mediante la función **perror()** cuyo prototipo, definido en **stdio.h**, es

void perror (const char ****cadena*);**

Esta función envía a **stderr** (generalmente la pantalla) la cadena indicada en el argumento, dos puntos y, a continuación, un mensaje del sistema asociado al último error producido. La forma en que se relaciona el error con el mensaje es mediante una variable global predefinida llamada **errno** (definida en **errno.h**) que se activa cuando se producen errores. Por ejemplo, dado el segmento de programa

```
FILE *f;

if (!(f = fopen ("DATOS.DAT", "rb"))) {
printf ("\nError %d. ", errno);
perror ("DATOS.DAT");
exit (1);
}
```

si no existe el fichero DATOS.DAT se envía a pantalla el mensaje

Error 2. DATOS.DAT: No such file or directory

Cada vez que se realiza una operación de lectura sobre un fichero, el indicador de posición del fichero se actualiza. Es necesario, pues, controlar la condición de fin de fichero. Por ello, debemos saber que cuando se intentan realizar lecturas más allá del fin de fichero, el carácter leído es siempre **EOF**. Sin embargo en los canales binarios un dato puede tener el valor **EOF** sin ser la marca de fin de fichero. Es aconsejable, por ello, examinar la condición de fin de fichero mediante la función **feof()**, cuyo prototipo, definido en **stdio.h**, es

int feof (FILE ****canal*);**

Esta función devuelve un valor diferente de cero cuando se detecta el fin de fichero.

Un algoritmo que lea todo un archivo puede ser el siguiente:

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

void main (void)
{
    FILE *f;

    if (!(f = fopen ("DATOS.DAT", "rt"))) {
        perror ("\nDATOS.DAT");
        exit (1);
    }

    //operación de lectura sobre DATOS.DAT
    while (!feof (f)) {
        //tratamiento
        //operación de lectura sobre DATOS.DAT
    }

    fclose (f);
    if (ferror (f)) puts ("Error al cerrar el archivo DATOS.DAT");
}
```

E/S de caracteres

Para leer caracteres individuales de un fichero se utiliza la función

int fgetc (FILE ****canal*);**

o su macro equivalente

int getc (FILE ****canal*);**

Ambas están definidas en **stdio.h** y son completamente idénticas. Devuelven el carácter leído e incrementan el contador de posición del fichero en 1 byte. Si se detecta la condición de fin de fichero, devuelven **EOF**, pero para canales binarios es mejor examinar dicha condición mediante **feof()**.

Para escribir caracteres individuales en un fichero se utiliza la función

int fputc (int *carácter***, FILE ****canal***);**

o su macro equivalente

int putc (int *carácter***, FILE ****canal***);**

Ambas tienen el prototipo definido en **stdio.h** y son completamente idénticas. Escriben el carácter indicado en el argumento (que puede ser también una variable **char**) en el fichero asociado a *canal*. Si no hay error, devuelven el carácter escrito; en caso contrario devuelven **EOF**.

El siguiente programa copia carácter a carácter el fichero DATOS.DAT en COPIA.DAT.

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

void main (void)
{
    FILE *fent, *fsal;
    char caracter;

    if (!(fent = fopen ("DATOS.DAT", "rb"))) {
        perror ("DATOS.DAT");
        exit (1);
    }
    if (!(fsal = fopen ("COPIA.DAT", "wb"))) {
        perror ("COPIA.DAT");
        exit (1);
    }

    caracter = getc (fent);
    while (!feof (fent)) {
        putc (caracter, fsal);
        if (ferror (fsal)) puts ("No se ha escrito el carácter");
        caracter = getc (fent);
    }

    fclose (fent);
    fclose (fsal);
}
```

La misma operación puede realizarse en una sola línea mediante

```
while (!feof (fent)) putc (getc (fent), fsal);
```

Existen dos funciones análogas a **getc()** y **putc()** pero para leer y escribir enteros en lugar de caracteres:

int getw (FILE ****canal*);**

int putw (int *entero***, FILE ****canal***);**

Ambas están definidas en **stdio.h** y la única diferencia con **getc()** y **putc()** está en que se procesan dos bytes en cada operación. Estos dos bytes deben interpretarse

como un número entero. La función **getw()** no debe usarse con ficheros abiertos en modo texto.

E/S de cadenas de caracteres

Para leer cadenas de caracteres se utiliza la función

char *fgets (char **cadena*, int *n*, FILE **canal*);

cuyo prototipo está definido en **stdio.h**. Esta función lee caracteres del fichero asociado a *canal* y los almacena en *cadena*. En cada operación de lectura se leen *n - 1* caracteres, a menos que se encuentre primero un carácter nueva línea (que también se almacena en *cadena*). Si no se produce error, la función devuelve un puntero a *cadena*; en caso contrario, devuelve un puntero nulo.

El siguiente programa muestra el contenido del fichero AUTOEXEC.BAT, numerando las líneas. Se supone que ninguna línea tiene más de 80 caracteres.

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

void main (void)
{
    FILE *f;
    register int i = 1;
    char cadena[81];

    if (!(f = fopen ("AUTOEXEC.BAT", "rt"))) {
        perror ("AUTOEXEC.BAT");
        exit (1);
    }
    fgets (cadena, 80, f);
    while (!feof (f)) {
        printf ("%d: %s", i, cadena);
        i++;
        fgets (cadena, 80, f);
    }
    fclose (f);
}
```

Para escribir cadenas de caracteres se utiliza la función

int fputs (const char **cadena*, FILE **canal*);

cuyo prototipo está definido en **stdio.h**, y que escribe la cadena indicada en el argumento en el fichero asociado a *canal*. Hay que tener en cuenta que **fputs()** no

copia el carácter nulo ni añade un carácter nueva línea al final. Si no se produce error, **fputs()** devuelve el último carácter escrito; en caso contrario devuelve **EOF**.

El siguiente programa escribe en la impresora las cadenas de caracteres que se van tecleando, hasta que se pulsa ↵.

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

void main (void)
{
    char *cadena[85];

    printf ("\nTeclee cadena: ");
    gets (cadena);
    while (cadena[0]) {
        strcat (cadena, "\n\r");
        fputs (cadena, stdprn);
        gets (cadena);
    }
}
```

E/S de bloques de datos

La biblioteca estándar de C dispone de funciones que permiten leer y escribir bloques de datos de cualquier tipo. Las funciones que realizan estas operaciones son, respectivamente, **fread()** y **fwrite()**, cuyos prototipos, definidos en **stdio.h**, son los siguientes:

int fread (void ****buffer*, int** *nbytes***, int** *contador***, FILE ****canal***);**

int fwrite (void ****buffer*, int** *nbytes***, int** *contador***, FILE ****canal***);**

La función **fread()** lee *contador* bloques de datos del fichero asociado a *canal* y los sitúa en *buffer*. Cada bloque contiene *nbytes* bytes. La función **fwrite()** vuelca en el fichero asociado a *canal* los *contador* bloques de *nbytes* bytes que se encuentran almacenados a partir de *buffer*. Si la operación tiene éxito, **fread()** devuelve el número de bloques (no de bytes) realmente leídos. Análogamente, **fwrite()** devuelve el número de bloques realmente escritos. La declaración

**void ****buffer*

indica que ***buffer*** es un puntero a cualquier tipo de variables.

En el programa siguiente se almacenan en el fichero MATRIZ.FLO un conjunto de 10 números de tipo **float**. La escritura se hace con una sentencia **fwrite()**

para cada dato. En la operación de lectura se leen los 10 datos de una sola vez con una sentencia **fread()** y, a continuación, se muestran en pantalla.

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

void main (void)
{
    register int i;
    FILE *f;
    float elem, matriz[10];

    if (!(f = fopen ("MATRIZ.FLO", "w+b"))) {
        perror ("MATRIZ.FLO");
        exit (1);
    }

    for (i = 0; i <= 9; i++) {
        printf ("\nTeclee número: ");
        scanf ("%f", &elem);
        fwrite (&elem, sizeof (elem), 1, f);
    }
    rewind (f);

    fread (matriz, sizeof (matriz), 1, f);

    clrscr ();
    for (i = 0; i <= 9; i++) printf ("\n%d: %f", i, matriz[i]);

    fclose (f);
}
```

Fijémonos en algunos detalles del programa anterior. La sentencia

```
fwrite (&elem, sizeof (elem), 1, f);
```

vuelca en el fichero, cada vez, los 4 bytes -**sizeof (elem)**- de **elem**. Una vez finalizado el primero de los bucles **for** se han ejecutado 10 operaciones **fwrite()** y, por tanto, el indicador de posición del fichero apunta al final del mismo. Para poder realizar la lectura de los datos hemos de recolocar este indicador al principio del fichero. Esto puede hacerse de dos formas:

- Cerrando el fichero y volviéndolo a abrir para lectura.
- Mediante la función **rewind()**.

En el ejemplo se utiliza el segundo método. La función **rewind()** reposiciona el indicador de posición del fichero al principio del mismo. Su prototipo, definido en **stdio.h**, es el siguiente:

void rewind (FILE ****canal*);**

Siguiendo con el programa anterior, la sentencia

```
fread (matriz, sizeof (matriz), 1, f);
```

lee del fichero 40 bytes -**sizeof (matriz)**- y los sitúa en **matriz**. Fijémonos que ahora no es necesario escribir **&matriz** en el primer parámetro, pues **matriz** ya es un puntero.

Habitualmente **fread()** y **fwrite()** se utilizan para leer o escribir estructuras. Veámoslo con un programa que crea un archivo de listín telefónico. El registro de ese archivo constará de los siguientes campos:

Nombre	**40 caracteres**
Domicilio	**40 caracteres**
Población	**25 caracteres**
Provincia	**15 caracteres**
Teléfono	**10 caracteres**

El siguiente programa crea ese fichero, llamado LISTIN.TEL.

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

typedef struct {
    char nom[41];
    char dom[41];
    char pob[26];
    char pro[16];
    char tel[11];
} REG;

void main (void)
{
    FILE *f;
    REG var;

    if (!(f = fopen ("LISTIN.TEL", "wb"))) {
        perror ("LISTIN.TEL");
        exit (1);
    }

    clrscr ();
    printf ("Nombre: ");
    gets (var.nom);
    while (var.nom[0]) {
        printf ("\nDomicilio: ");
        gets (var.dom);
        printf ("\nPoblación: ");
        gets (var.pob);
        printf ("\nProvincia: ");
        gets (var.pro);
        printf ("\nTeléfono: ");
        gets (var.tel);
```

```
        fwrite (&var, sizeof (var), 1, f);
        if (ferror (f)) {
            puts ("No se ha almacenado la información");
            getch ();
        }

        clrscr ();
        printf ("Nombre: ");
        gets (var.nom);
    }

    fclose (f);
}
```

El siguiente programa ilustra como se pueden leer bloques de registros de un fichero. Concretamente lee los registros del fichero LISTIN.TEL en grupos de 4, mostrando en pantalla los campos **Nombre** y **Teléfono**.

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

typedef struct {
    char nom[41];
    char dom[41];
    char pob[26];
    char pro[16];
    char tel[11];
} REG;

void main (void)
{
    FILE *f;
    REG var[4];
    int i, n;

    if (!(f = fopen ("LISTIN.TEL", "rb"))) {
        perror ("LISTIN.TEL");
        exit (1);
    }

    do {
        clrscr ();
        n = fread (var, sizeof (REG), 4, f);
        for (i = 0; i < n; i++) printf ("\n%-41s %s", var[i].nom, var[i].tel);
        puts ("\nPulse una tecla ...");
        getch ();
    } while (!feof (f));

    fclose (f);
}
```

Si nos fijamos en la sentencia **fread()** de este programa, se leen en cada operación 4 bloques de **sizeof (REG)** bytes (135, tamaño de cada registro). La misma cantidad de bytes se leería mediante la sentencia

```
fread (var, sizeof (var), 1, f);
```

sin embargo, así no tendríamos un control adecuado sobre la variable **n**.

E/S con formato

La función que permite escribir con formato en un fichero es

int fprintf (FILE ****canal*, const char****formato***,** *lista de argumentos***);**

que escribe los argumentos de la lista, con el formato indicado, en el fichero asociado a *canal*. Esta función es idéntica a **printf()** salvo en que permite escribir en cualquier dispositivo y no sólo en **stdout**. Un uso de **fprintf()** se estudió en el Capítulo 4 para salida por impresora.

Para leer con formato existe la función

int fscanf (FILE ****canal*, const char****formato***,** *lista de argumentos***);**

que es idéntica a **scanf()** salvo en que puede leer de cualquier dispositivo, no necesariamente de **stdin**.

Aunque estas funciones pueden ser de gran utilidad en ciertas aplicaciones, en general es más recomendable usar **fread()** y **fwrite()**.

Acceso directo

El acceso directo (tanto en lectura como en escritura) a un archivo, se realiza con la ayuda de la función **fseek()** que permite situar el indicador de posición del archivo en cualquier lugar del mismo. El prototipo de esta función es:

int fseek (FILE ****canal*, long** *nbytes***, int** *origen***);**

Esta función sitúa el indicador de posición del fichero *nbytes* contados a partir de *origen*. Los valores posibles del parámetro *origen* y sus macros asociadas se muestran en la siguiente tabla:

ORIGEN	***VALOR***	***MACRO***
Principio del fichero	*0*	*SEEK_SET*
Posición actual	*1*	*SEEK_CUR*

Fin del fichero	*2*	*SEEK_END*

La función devuelve 0 cuando ha tenido éxito. En caso contrario devuelve un valor diferente de 0.

Esta función simplemente maneja el indicador de posición del fichero, pero no realiza ninguna operación de lectura o escritura. Por ello, después de usar **fseek()** debe ejecutarse una función de lectura o escritura.

El siguiente programa crea un archivo llamado FRASE.TXT con una cadena de caracteres. Posteriormente lee un carácter de la cadena cuya posición se teclea.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <process.h>

void main (void)
{
    FILE *f;
    int nbyte, st;
    char frase[80], caracter;

    if (!(f = fopen ("FRASE.TXT", "w+t"))) {
        perror ("FRASE.TXT");
        exit (1);
    }

    clrscr ();
    printf ("Teclee frase: ");
    gets (frase);
    fwrite (frase, strlen (frase) + 1, 1, f);

    printf ("\nLeer carácter nº: ");
    scanf ("%d", &nbyte);
    st = fseek (f, (long) nbyte, SEEK_SET);
    if (st) puts ("Error de posicionamiento");
    else {
        caracter = getc (f);
        if (caracter != EOF) printf ("\nEl carácter es: %c", caracter);
        else puts ("Se sobrepasó el fin de fichero");
    }

    fclose (f);
}
```

Puede usarse **fseek()** para acceder a registros. Para ello debe calcularse previamente en qué byte del fichero comienza el registro buscado. El siguiente programa escribe registros ayudándose de **fseek()**.

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

typedef struct {
    char  nombre[40];
    int   edad;
    float altura;
} REGISTRO;

void main (void)
{
    FILE *f1;
    REGISTRO mireg;
    int num;
    long int puntero;

    if (!(f1 = fopen ("REGISTRO.DAT", "r+b"))) {
        puts ("Error de apertura");
        exit (1);
    }

    clrscr ();
    printf ("Escribir registro nº: ");
    scanf ("%d", &num);
    while (num > 0) {
        getchar ();
        printf ("Nombre: ");
        gets (mireg.nombre);
        printf ("Edad: ");
        scanf ("%d", &mireg.edad);
        printf ("Altura: ");
        scanf ("%f", &mireg.altura);
        puntero = (num - 1) * sizeof (REGISTRO);
        if (fseek (f1, puntero, SEEK_SET)) puts ("Error de posicionamiento");
        else {
            fwrite (&mireg, sizeof (mireg), 1, f1);
            if (ferror (f1)) {
                puts ("ERROR de escritura");
                getch ();
            }
        }
        clrscr ();
        printf ("Escribir registro nº: ");
        scanf ("%d", &num);
    }

    fclose (f1);
}
```

Ejercicios

1. Escribe un programa que vaya almacenando en un archivo todos los caracteres que se tecleen hasta que se pulse CTRL-Z. El nombre del archivo se pasará como parámetro en la línea de órdenes.

2. Escribe un programa que lea un archivo de texto y cambie todas las apariciones de una palabra determinada, por otra. El programa se ejecutará con la orden

PROG fichero palabra1 palabra2

siendo **palabra1** la palabra sustituida por **palabra2**. El programa debe informar del número de sustituciones efectuadas.
Nota: Se supondrá que las líneas del fichero no tienen más de 80 caracteres.

3. En el club de baloncesto BAHEETO'S BASKET CLUB se está realizando una campaña de captación de jugadores altos. Se dispone de un fichero con datos de aspirantes, llamado ALTOS.DAT, que se describe a continuación

ALTOS.DAT	**Aspirantes a jugadores del club**	
Campo	**Descripción**	**Tipo**
nom	Nombre del aspirante	char(41)
alt	Altura del aspirante (en metros)	float
pro	Provincia de nacimiento	char(26)

- Los campos de tipo **char** incluyen el nulo final
- El fichero almacena un máximo de 500 registros.
- La provincia se almacena en mayúsculas.
- El fichero no está ordenado.

Construye un programa que realice las siguientes operaciones:

- Solicitar por teclado una provincia y almacenar en sendas matrices los nombres y las alturas de los aspirantes nacidos en la provincia indicada.
- Calcular la altura media de todos los aspirantes de dicha provincia.
- Emitir un informe impreso con los nombres y alturas de los aspirantes de la provincia cuya altura supere la media. El formato del listado debe ser el siguiente:

Provincia: xxxxxxxxxxxxxxxxxxxxxxxxx
Altura Media: x.xx

Nombre	**Altura**
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	**x.xx**

... 50 líneas de detalle por página

4. Se tienen dos archivos de datos **ARTIC.DAT** y **NOMPRO.DAT**, que se describen a continuación:

ARTIC.DAT Maestro de Artículos

Campo	Descripción	Tipo
cod	Código del artículo	char(7)
des	Descripción	char(31)
exi	Existencia	unsigned
pco	Precio de compra	unsigned
pvp	Precio de Venta	unsigned
pro	Código del proveedor	char(5)

- Los campos de tipo **char** incluyen el nulo final
- Los campos **cod** y **pro** almacenan sólo dígitos numéricos y están completados a la izquierda con ceros. Por ejemplo, el artículo de código 65 se almacena como 000065.
- El fichero está ordenado ascendentemente por el campo **cod**.
- No están todos los valores de **cod**.

NOMPRO.DAT Nombres de proveedores

Campo	Descripción	Tipo
nom	Nombre del proveedor	char(41)

- Cada nombre de proveedor está almacenado en el registro cuyo número coincide con su código. Así, el nombre del proveedor de código 000041 se almacena en el registro 41.

Escribe un programa que emita un informe impreso con el formato de la página siguiente.
Al inicio del programa se solicitarán por teclado los códigos de artículo inicial y final que limitarán los registros que se deben listar.

11
Ficheros indexados: la interfase Btrieve

Introducción

El Lenguaje C no incorpora instrucciones ni funciones de manejo de ficheros indexados. Para disponer de este tipo de acceso en un programa C debemos recurrir a una gestión propia, lo que resulta excesivamente complejo, o a algún programa comercial que proporcione una interfase adecuada. Uno de los programas más eficientes de gestión de archivos indexados es el programa Btrieve, desarrollado por Novell. En el presente capítulo se describirá una parte importante de Btrieve, pero no se realizará un estudio exahustivo, pues ello merecería un libro monográfico.

Las explicaciones del capítulo se refieren a la versión 5.0, aunque gran parte de lo que se diga es válido para otras versiones.

Descripción de Btrieve

Btrieve es un programa residente en memoria que gestiona, lee, inserta y actualiza registros en los ficheros de datos de las aplicaciones, utilizando para ello los recursos del sistema operativo. En principio, Btrieve fue diseñado para trabajar en entornos monousuario, pero las últimas versiones incorporan operaciones de bloqueo de registros que permiten trabajar en entornos de red. El estudio de este capítulo se circunscribe a entornos monousuario. Para otros trabajos debe consultarse el Manual de Operaciones Btrieve.

Los requerimientos mínimos de Btrieve son los siguientes:

- Ordenador personal IBM o compatible.
- 128 Kb de memoria.
- Una unidad de disco.
- Sistema operativo MS-DOS 2.0 o posterior.

La rutina que queda residente en memoria ocupa un mínimo de 35 kb.

Entre las características de Btrieve que cabe destacar están las siguientes:

- Soporta hasta 24 claves para cada fichero.
- Pueden añadirse o eliminarse claves después de creado el fichero.
- Admite 14 tipos de claves.
- Permite claves duplicadas (más de un registro con el mismo valor para la clave), modificables, segmentadas (formadas por dos segmentos disjuntos del registro), nulas y descendentes.
- Maneja ficheros de hasta 4 Mb con un número ilimitado de registros.
- Puede trabajar con ficheros almacenados en dos dispositivos diferentes.
- Proporciona 36 operaciones distintas que pueden ejecutarse desde el programa de aplicación.

El flujo de control entre el programa de aplicación y el gestor Brieve es, básicamente, el siguiente:

- El programa emite una llamada a Btrieve por medio de una función a la que se pasa un bloque de parámetros. La forma de estas llamadas cambia ligeramente de un lenguaje a otro. Más adelante estudiaremos cómo se hace para Turbo C.
- Una pequeña rutina de interfase incluida en el programa guarda el bloque de parámetros en memoria y llama a la parte residente de Btrieve.
- Btrieve recibe el bloque de parámetros, los valida, y ejecuta la operación que se haya indicado, por ejemplo la lectura o escritura de un registro en el fichero.
- Btrieve devuelve los datos apropiados, entre ellos un código que indica si la operación ha tenido éxito o no.
- Btrieve devuelve el control al programa de aplicación.

Gestión de ficheros Btrieve

Los ficheros Btrieve se crean mediante el programa BUTIL.EXE (se explica más adelante) y se gestionan mediante el Gestor de Datos BTRIEVE.EXE.

Un fichero Btrieve está formado por una serie de páginas. La página es la unidad de transferencia utilizada por Btrieve entre memoria y disco en una operación de E/S. El tamaño de la página se especifica cuando se crea el fichero, y ha de ser un múltiplo de 512 bytes, hasta un máximo de 4096. El tamaño óptimo de la página depende de las características del registro. Más adelante veremos cómo determinar ese valor.

Lás páginas de un fichero Btrieve son de tres tipos:

- **Página de cabecera**, también llamada **registro de control del fichero** (FCR). Contiene información sobre las características del fichero.
- **Páginas de datos**, en donde se almacenan los registros de datos.
- **Páginas de índices**, que contienen los valores de las claves.

Btrieve almacena en la página de datos tantos registros como le es posible. Es conveniente determinar el tamaño óptimo de la página de datos para aumentar la eficiencia de Btrieve. Para calcular este tamaño óptimo hay que tener en cuenta lo siguiente:

- Por cada clave duplicada Btrieve almacena 8 bytes extra de información en cada registro.
- Si el fichero admite registros de longitud variable Btrieve añade 4 bytes por registro.
- Si se admite truncamiento de blancos se añaden 6 bytes por registro.
- Cada página requiere 6 bytes para información de cabecera.

Con estos datos es posible determinar el tamaño de página que hace mejor utilización del disco. Veamos un ejemplo. Sea un fichero cuya longitud de registro lógico es 117 bytes, y que utiliza dos claves que admiten duplicados. La longitud del registro físico es:

$$117 + 2 * 8 = 133 \text{ bytes}$$

La siguiente tabla muestra el porcentaje de ocupación para cada página:

TAMAÑO DE PÁGINA	***ÚTIL***	***NÚMERO DE REGISTROS***	***OCUPADO***	***NO UTILIZADO***	***% DE PÁGINA OCUPADA***
512	*506*	*3*	*405*	*107*	*79.1*
1024	*1018*	*7*	*937*	*87*	*91.5*
1536	*1530*	*11*	*1469*	*67*	*95.6*
2048	*2042*	*15*	*2001*	*47*	*97.7*
2560	*2554*	*19*	*2533*	*27*	*98.9*
3072	*3066*	*23*	*3065*	*7*	*99.8*
3584	*3578*	*26*	*3464*	*120*	*96.7*
4096	*4090*	*30*	*3996*	*100*	*97.6*

En este ejemplo vemos que un tamaño de página de 512 bytes es inadecuado, pues por cada 512 bytes se malgastan 107. También vemos que en una página de 4096 bytes se almacenan 30 registros, pero se desperdician 100 bytes, lo que significa que si el número de registros del fichero es elevado, se pierde mucho espacio. El tamaño óptimo parece ser el de 3072 bytes, pues tan solo se pierden 7 bytes cada 23 registros.

Una característica importante de Btrieve es el control que realiza de la consistencia de los datos. Si se produce un fallo en el sistema mientras Btrieve hace actualizaciones en las páginas pueden producirse inconsistencias en el

fichero. Para protegerse contra ello Btrieve realiza un proceso llamado **preimagen**, mediante el cual se restaura automáticamente el fichero a la situación inmediatamente anterior a la operación no completada. Para ello Btrieve utiliza un fichero temporal, llamado **fichero preimagen**, en el que registra los cambios en el fichero mientras dura la operación. El fichero preimagen tiene el mismo nombre que el fichero de datos Btrieve, pero con extensión .PRE, por lo que debe evitarse esta extensión para los ficheros de datos.

Otro metodo proporcionado por Btrieve para proteger la consistencia de los ficheros es la definición de **transacciones**. Entre las operaciones aportadas por Btrieve hay dos, denominadas **Inicio** y **Fin de Transacción**. Las operaciones Btrieve realizadas entre el inicio y el final de una transacción sólo se actualizarán en el fichero si la transacción tiene éxito. Si el sistema falla antes de completar la transacción, las operaciones Btrieve solicitadas desde el inicio de la transacción no se actualizan en el fichero. Btrieve utiliza un **fichero de control de transacción** para seguir la pista de los ficheros involucrados.

Otra particularidad de Btrieve es que permite abrir ficheros en modo acelerado para mejorar el rendimiento en operaciones de inserción, actualización y eliminación. Cuando se abre un fichero en modo acelerado Btrieve no escribe páginas en disco hasta que la memoria intermedia está llena y el algoritmo LRU (**L**ast **R**eciently **U**sed) selecciona un área para ser rellenada.

Por último, Btrieve permite restringir el acceso a ficheros especificando un nombre de propietario o palabra de paso, evitando accesos no autorizados a los datos.

El Gestor de Datos Btrieve

Para que un programa pueda hacer llamadas a Btrieve ha de cargarse previamente en memoria el Gestor de Datos **BTRIEVE.EXE**, que es quién ejecuta todas las operaciones de gestión de registros y ficheros. Para ello, desde el indicador del DOS debe ejecutarse el programa BTRIEVE.EXE mediante

BTRIEVE [*opciones*]

Una vez cargado correctamente en memoria aparecerá un mensaje similar a

Btrieve Record Manager Version 5.xx
Copyright (c) 1982, 1988, novell, Inc. All Rights Reserved

Si en la carga se produce algún error, Btrieve envía un mensaje describiendo las características del problema. Si no hay error, Btrieve permanece residente en memoria hasta que se interrumpe la corriente, se recarga el DOS, se descarga mediante el utilitario **BUTIL** (hablaremos de él más adelante) o desde el programa de aplicación se emite una orden de parada.

La tabla siguiente describe algunas de las opciones de arranque del Gestor de Datos.

OPCIÓN	***DESCRIPCIÓN***
/M: *Tamaño de memoria*	*Es un valor comprendido entre 9 y 64 que determina el tamaño de memoria en Kb de las áreas de Btrieve. El valor por defecto es 32 Kb.*
/P: *Tamaño de página*	*Especifica el tamaño de la página. Ha de ser un múltiplo de 512 hasta un máximo de 4096. Si se especifica un tamaño de página de 4096, debe asignarse, al menos, 36 kb para la opción* ***/M****.*
/T: *Fichero de transacción*	*Especifica el fichero de control de transacciones.*
/I: *Dispositivo del fichero preimagen*	*Especifica el dispositivo en el que se ubicará el fichero preimagen.*
/F: *Ficheros abiertos*	*Especifica el máximo número de ficheros Btrieve que pueden permanecer abiertos simultáneamente. El valor por defecto es 20. El máximo es 255.*

El utilitario BUTIL

El utilitario **BUTIL.EXE** es un programa que realiza ciertas operaciones sobre ficheros Btrieve (crear, copiar, ...), permite descargar de memoria el Gestor de Datos, o informar sobre la versión de Btrieve que se está utilizando.

Antes de ejecutar BUTIL debe cargarse en memoria el Gestor de Datos. La sintaxis de BUTIL es la siguiente:

BUTIL -*mandato* [*parámetros*] [**-O** <*propietario*>]

A continuación se describen algunos de los mandatos de BUTIL.

CLONE

Crea un fichero Btrieve vacío con las mismas características que uno existente. La sintaxis para CLONE es:

butil -CLONE <*fichero existente*> <*fichero nuevo*> [**-O** <*propietario*>]

COPY

Copia el contenido de un fichero Btrieve en otro. Ambos deben existir. La sintaxis es:

butil -COPY <*fich. entrada*> <*fich. salida*> [**-O** <*prop. ent*> [**-O** <*prop. Sal*>]]

CREATE

Los ficheros Btrieve se crean con esta orden del utilitario BUTIL. Mediante CREATE se crea un fichero Btrieve vacío con ciertas propiedades. Éstas se especifican en un fichero de texto que contiene una lista de parámetros que definen las características del fichero. El fichero que contiene los parámetros se crea con cualquier editor de texto, y contiene sentencias del tipo

palabra_reservada=valor

en donde *palabra_reservada* se refiere a una característica del fichero o de sus claves, y *valor* es el valor asignado a dicha característica. La tabla siguiente muestra algunas de las palabras reservadas utilizadas para fijar características generales del fichero.

record=#	*Especifica la longitud del registro lógico. El valor # debe estar comprendido entre 4 y 4090*
variable=<***y***|***n***>	*Especifica si el fichero contiene o no registros de longitud variable.*
key=#	*Especifica el número de claves del fichero. Se admite cualquier valor entre 1 y 24.*
page=#	*Especifica el tamaño de página. Debe ser un múltiplo de 512 hasta un máximo de 4096.*
replace=<***y***|***n***>	*Se utiliza para indicar si se desea que Btrieve cree un nuevo fichero si ya existe uno con el mismo nombre, advirtiendo de su existencia.*
position=#	*Especifica la posición, dentro del registro lógico, en que comienza la clave que se describe.*
length=#	*Especifica la longitud de la clave.*
duplicates=<***y***|***n***>	*Determina si se permite o no duplicados en las claves.*
modifiable=<***y***|***n***>	*Indica si el programa puede modificar el valor de la clave.*
type=*tipo*	*Indica el tipo de clave. Algunos de los tipos de clave permitidos se muestran en una tabla posterior.*
descending=y	*Se especifica cuando se quiere mantener el fichero ordenado descendentemente respecto de la clave definida.*
alternate=<***y***|***n***>	*Se especifica **y** cuando se desea un criterio de clasificación diferente al ASCII estándar. En ese caso, el nuevo criterio se almacena en un fichero cuya estructura se explica más adelante.*
segment=<***y***|***n***>	*Indica si la clave tiene o no algún segmento más.*
name=*fichero*	*Especifica el fichero que contiene el nuevo criterio de clasificación si se ha indicado **alternate=y**.*

La siguiente tabla muestra alguno de los tipos de clave que se pueden especificar con el parámetro **type**

TIPO	***DESCRIPCIÓN***
string	*Cadenas de caracteres*
integer	*Enteros almacenados en binario (2 bytes)*
float	*Números en punto flotante (4-8 bytes)*
numeric	*Números almacenados como cadenas de caracteres*
lstring	*Como **string**, especificando la longitus en el byte 0*
zstring	*Cadenas tipo C: finalizadas con el ASCII nulo*

Si en alguna clave se ha especificado la sentencia **alternate=y**, la última sentencia del fichero debe ser del tipo **name=***fichero*, siendo *fichero* el archivo que almacena el nuevo criterio de clasificación. Este archivo debe tener la siguiente estructura:

- **byte 0** Es un byte de control que debe almacenar siempre el valor AC hex.
- **bytes 1-8** Almacenan un nombre que identifica la secuencia de intercalación alternativa ante Btrieve.
- **256 bytes** Almacenan la nueva clasificación.

Cuando se crea el fichero de parámetros deben tenerse en cuenta las siguientes normas:

- Todos los elementos deben estar en minúsculas.
- Deben presentarse en el mismo orden en que se han expuesto en las tablas anteriores.
- Las sentencias han de ser consistentes. Por ejemplo, si se ha indicado para alguna clave **alternate=y**, el último elemento del fichero de parámetros debe ser una sentencia **name**.

Veamos un ejemplo. Sea el siguiente fichero, de nombre DATOS.PAR:

```
record=113
variable=n
key=2
page=3072
replace=n
position=1 length=5 duplicates=n modifiable=n type=zstring alternate=n segment=n
position=6 length=30 duplicates=y modifiable=y type=zstring alternate=n segment=y
position=46 length=3 duplicates=y modifiable=y type=zstring alternate=n
segment=n
```

Mediante la orden

BUTIL -CREATE DATOS.BTR DATOS.PAR

se crea un fichero llamado DATOS.BTR con las siguientes especificaciones:

- Registros de longitud fija de 113 bytes
- Páginas de 3072 bytes
- Dos claves de acceso definidas de la siguiente manera
  1. Clave no segmentada de 5 bytes, contados desde el primer byte del registro, de tipo **zstring**, no duplicada y no modificable.
  2. Clave de tipo **zstring**, duplicada y modificable con dos segmentos: uno de 30 bytes contados a partir del 6º, y otro de 3 bytes a partir del 46º.

LOAD

Permite insertar registros de un fichero secuencial en un fichero Btrieve. La sintaxis es

butil -LOAD <*fichero secuencial*> <*fichero Btrieve*> [**-O** <*propietario*>]

La estructura de los registros del fichero secuencial debe ser

#,reg<CR+LF>

siendo **#** la longitud del registro, **reg** el registro a insertar en el fichero Btrieve, y **CR+LF** los caracteres retorno de carro y salto de línea. Por ejemplo, si con un editor de texto creamos un archivo DATOS.SEC que contenga

```
20,ABCDEFGHIJKLMNOPQRST
20,abcdefghijklmnopqrst
```

y tenemos un fichero Btrieve llamado DATOS.BTR, la orden

BUTIL -LOAD DATOS.SEC DATOS.BTR

inserta los 2 registros de DATOS.SEC en DATOS.BTR.

Es importante asegurarse de que cada línea escrita en el fichero secuencial finaliza con los caracteres Retorno de Carro y Salto de Línea, y de que la longitud de registro especificada es la misma que se indicó al crear el fichero Btrieve.

SAVE

Vuelca el contenido de un fichero Btrieve en un fichero secuencial. Es la operación inversa a LOAD. La sintaxis es

butil -SAVE <*fich.Btrieve*> <*fich.secuencial*> <*Nº clave*> [**-O** <*propietario*>]

Los registros creados con SAVE tienen la misma estructura explicada para el mandato LOAD. Si no se especifica lo contrario, el fichero se vuelca ordenado por la clave 0. Si se quiere otro orden se indica en el parámetro *Nº clave*.

STAT

Muestra en pantalla información con las características de un fichero Btrieve. La sintaxis es

butil -STAT <*fichero Btrieve*> [**-O** <*propietario*>]

STOP

Elimina el Gestor de Datos de memoria. La sintaxis es

butil -STOP

VER

Informa de la versión del Gestor de Datos con que se está trabajando. La sintaxis es:

butil -VER

Interfase de Btrieve con Turbo C

Las operaciones Btrieve en C no utilizan ninguna de las funciones estudiadas en el Capítulo 10. Todas las operaciones sobre ficheros Btrieve se hacen mediante llamadas a una función **BTRV**, de prototipo

int BTRV (int *op***, char *** *bloque***, void *** *datos***, int *** *lonreg***, char *** *keyval***, char *** *nkey***);**

A continuación se describe cada uno de los parámetros:

- *op* Es un valor entero que indica la operación a realizar sobre el fichero. Las más habituales se describen en el siguiente apartado del capítulo.
- *bloque* Es un área de 128 bytes, denominado **bloque de posición** del fichero, que Btrieve utiliza para un correcto manejo del mismo, inicializándolo en la apertura. Debe asignarse un bloque de posición diferente para cada fichero Btrieve que se abra.
- *datos* Es la dirección de una variable, generalmente de tipo **struct**, en

donde se almacena el registro que va a ser escrito en el fichero y donde se recibe el registro leído del fichero.

- *lonreg* Con este parámetro se pasa la dirección de una variable que contiene la longitud del registro.
- *keyval* Almacena el valor de la clave por la que se está accediendo.
- *nkey* Es el número de clave con que se accede al fichero.

La función BTRV devuelve un valor entero. Si la operación ha tenido éxito, devuelve 0. En caso contrario devuelve un código de error. Los códigos de error se muestran más adelante en el capítulo. Para poder utilizar la función BTRV debe incluirse el archivo **turcbtrv.c** mediante una sentencia **#include** como

```
#include <c:\btr\turcbtrv.c>
```

suponiendo que el fichero **turcbtrv.c** se encuentra en el directorio **c:\btr**. También puede compilarse este fichero y obtener un módulo objeto que se enlace más tarde con el programa (se estudia cómo hacerlo en el Capítulo 12). Veremos más adelante algunos ejemplos de programas que manejan ficheros Btrieve. Antes es necesario conocer las operaciones que se pueden realizar sobre ellos.

Operaciones Btrieve

Vamos a estudiar algunas de las operaciones Btrieve más comúnmente usadas. Se muestran en orden alfabético.

ABORTAR TRANSACCIÓN **21**

Deshace todas las operaciones realizadas desde el comienzo de una transacción activa y termina la transacción.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X					
Devuelve						

ABRIR **0**

Abre un fichero Btrieve cuyo nombre se especifica en **keyval**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X		X	X	X	X
Devuelve		X				

- Si hay palabra de paso se pone en **datos**, y en **longitud** se pone su longitud.
- En **nkey** se indica el modo de apertura. Los modos habituales son:
  a) **-1** Acelerado.
  b) **0** Apertura normal.

ABRIR TRANSACCIÓN **19**

Señala el comienzo de una transacción.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X					
Devuelve						

ACTUALIZAR 3

Actualiza un registro existente en un fichero Btrieve.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X	X	X		X
Devuelve		X			X	

Previamente debe haberse realizado con éxito una operación de lectura del registro que se desea actualizar.

ASIGNAR PALABRA DE PASO 29

Asigna una palabra de paso a un fichero Btrieve para prevenir el acceso al mismo a usuarios no autorizados.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X	X	X	X	X
Devuelve		X				

- El fichero debe estar abierto y no tener asignada una palabra de paso.
- La palabra de paso debe colocarse en el parámetro **datos** y en el parámetro **keyval**. Debe ser una cadena de hasta 8 caracteres acabada con un nulo.
- El parámetro **nkey** se inicializa a un valor que indica el tipo de restricción de acceso asociada al fichero. Los valores posibles son:
  a) **0** Palabra de paso requerida para cualquier acceso.
  No se criptografían datos.
  b) **1** Permiso de lectura sin palabra de paso.
  No se criptografían datos.
  c) **2** Palabra de paso requerida para cualquier acceso.
  Se criptografían datos.
  d) **3** Permiso de lectura sin palabra de paso.
  Se criptografían datos.

BORRAR 4

Actualiza un registro existente en un fichero Btrieve.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X		X		
Devuelve		X		X		

Previamente debe haberse realizado con éxito una operación de lectura del registro que se desea borrar.

CERRAR 1

Cierra un fichero Btrieve.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X				
Devuelve						

ELIMINAR PALABRA DE PASO **30**

Elimina la palabra de paso previamente asignada a un fichero Btrieve.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X				
Devuelve		X				

INSERTAR **2**

Inserta un nuevo registro en un fichero Btrieve.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X	X	X		X
Devuelve		X			X	

El parámetro **datos** debe almacenar el registro a insertar.

LEER ANTERIOR **7**

Accede al registro inmediatamente anterior al "registro actual".

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

Anteriormente debe haberse realizado con éxito una operación de lectura. El parámetro **keyval** debe pasarse tal como fue devuelto en dicha operación.

LEER IGUAL **5**

Accede al 1er registro cuyo valor de clave es el mismo que el que se indica en **keyval**.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X		X	X	X
Devuelve		X	X	X		

LEER INFERIOR **12**

Accede al registro correspondiente a la clave más baja de la vía **nkey**.

	op	bloque	datos	lonreg	keyval	nkey
A pasar	X	X		X		X
Devuelve		X	X	X	X	

LEER MAYOR **8**

Accede al 1er registro cuyo valor de clave es mayor que el especificado en **keyval**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

LEER MAYOR O IGUAL **9**

Accede al 1er registro con valor de clave mayor o igual que el indicado en **keyval**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

LEER MENOR **10**

Accede al 1er registro cuyo valor de clave es menor que el especificado en **keyval**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

LEER MENOR O IGUAL **11**

Accede al 1er registro con valor de clave menor o igual que el indicado en **keyval**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

LEER SIGUIENTE **6**

Accede al registro siguiente al "registro actual".

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X	X	X
Devuelve		X	X	X	X	

Anteriormente debe haberse realizado con éxito una operación de lectura. El parámetro **keyval** debe pasarse tal como fue devuelto en dicha operación.

LEER SUPERIOR **13**

Accede al registro correspondiente a la clave más alta para la vía **nkey**.

	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X	X		X		X
Devuelve		X	X	X	X	

PARAR						**25**
Descarga el Gestor de Datos de memoria y cierra todos los ficheros.						
	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X					
Devuelve						

TERMINAR TRANSACCIÓN						**20**
Completa una transacción y confirma las operaciones realizadas desde su inicio.						
	op	**bloque**	**datos**	**lonreg**	**keyval**	**nkey**
A pasar	X					
Devuelve						

Ejemplos

Veremos ahora algunos ejemplos de programas C que utilizan archivos Btrieve. El primero es un programa que realiza algunas operaciones básicas que se seleccionan desde un menú.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "c:\btr\turcbtrv.c"

#define  ABRIR          0
#define  CERRAR         1
#define  INSERTAR       2
#define  ACTUALIZAR     3
#define  BORRAR         4
#define  IGUAL          5
#define  SIGUIENTE      6
#define  ANTERIOR       7
#define  INFERIOR       12
#define  SUPERIOR       13

struct REG {
   char nombre[26];              // Campo Clave, no dup, no modificable
   char direcc[31];
   char telef[8];
};

void ErrorBtr (int, int);
void Pantalla (void);
int Opcion (void);
void Presentar (struct REG);
char *Intro (int, int, int, char *);
```

```
void main (void)
{
  int st;
  char bloque[128];
  struct REG datos;
  int lonreg = sizeof (datos);
  char keyval[26];
  char opc;

  st = BTRV (ABRIR, bloque, &datos, &lonreg, "DATOS.BTR", -1);
  if (st) {
    ErrorBtr (st, ABRIR);
    exit (1);
  }

  Pantalla ();
  while ((opc = Opcion ()) != 'X') {
    switch (opc) {
      case 'A':  Intro (15, 9, 25, datos.nombre);
                 strupr (datos.nombre);
           Intro (16, 9, 30, datos.direcc);
           Intro (17, 9, 7, datos.telef);
              st = BTRV (INSERTAR, bloque, &datos, &lonreg, keyval, 0);
              if (st) ErrorBtr (st, INSERTAR);
              else {
                 printf ("\nRegistro insertado. Pulse una tecla");
                 getch ();
                 delline ();
              }
              break;
      case 'B':  Intro (15, 9, 25, datos.nombre);
                 strcpy (keyval, strupr (datos.nombre));
                 st = BTRV (IGUAL, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, IGUAL);
                 else Presentar (datos);
                 break;
      case 'C':  st = BTRV (INFERIOR, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, INFERIOR);
                 else Presentar (datos);
                 break;
      case 'D':  st = BTRV (SUPERIOR, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, SUPERIOR);
                 else Presentar (datos);
                 break;
      case 'E':  st = BTRV (ANTERIOR, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, ANTERIOR);
                 else Presentar (datos);
                 break;
      case 'F':  st = BTRV (SIGUIENTE, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, SIGUIENTE);
                 else Presentar (datos);
                 break;
      case 'G':  st = BTRV (BORRAR, bloque, &datos, &lonreg, keyval, 0);
                 if (st) ErrorBtr (st, BORRAR);
                 else {
                    gotoxy (1, 20);
                    printf ("Registro borrado. Pulse una tecla ...");
                    getch ();
```

```
                    delline ();
                }
                break;
        case 'H':  Intro (16, 9, 30, datos.direcc);
                Intro (17, 9, 7, datos.telef);
                st = BTRV (ACTUALIZAR, bloque, &datos, &lonreg, keyval, 0);
                if (st) ErrorBtr (st, ACTUALIZAR);
                else {
                    gotoxy (1, 20);
                    printf ("Registro actualizado. Pulse una tecla ...");
                    getch ();
                    delline ();
                }
    }
  }

  clrscr ();
  st = BTRV (CERRAR, bloque, &datos, &lonreg, keyval, 0);
  if (st) ErrorBtr (st, CERRAR);
}

void ErrorBtr (int st, int op)
{
   gotoxy (1, 20);
   printf ("Error %d en operación %d", st, op);
   printf ("\nPulse una tecla ...");
   getch ();
   gotoxy (1, 20);
   delline ();
   delline ();
}

void Pantalla (void)
{
   clrscr ();
   printf ("A: Insertar\n");
   printf ("B: Leer registro\n");
   printf ("C: Leer inferior\n");
   printf ("D: Leer superior\n");
   printf ("E: Leer anterior\n");
   printf ("F: Leer siguiente\n");
   printf ("G: Borrar\n");
   printf ("H: Modificar\n");
   printf ("X: Salir");

   gotoxy (1, 15);
   printf ("Nombre: ");
   printf ("\nDirecc: ");
   printf ("\nTelef.: ");
}

int Opcion (void)
{
   int tecla;

   do {
      gotoxy (1, 11);
      printf ("==> Teclee opción:    ");
      gotoxy (20, 11);
```

```
    tecla = toupper (getche ());
  } while (!strchr ("ABCDEFGHX", tecla));

  return tecla;
}

void Presentar (struct REG x)
{
  gotoxy (9, 15);
  printf ("%-25s", x.nombre);
  gotoxy (9, 16);
  printf ("%-30s", x.direcc);
  gotoxy (9, 17);
  printf ("%-7s", x.telef);
}

char *Intro (int f, int c, int tam, char *cad)
{
  char aux[50];
  register int i;

  textattr (WHITE | BLUE * 16);
  gotoxy (c, f);
  for (i = 0; i < tam; i++) putch (' ');
  gotoxy (c, f);

  aux[0] = tam + 1;
  strcpy (cad, cgets (aux));

  textattr (LIGHTGRAY | BLACK * 16);
  return cad;
}
```

Puesto que en la mayoría de las llamadas a BTRV el único parámetro que cambia de una a otra es el código de operación, suele simplificarse la llamada mediante una función que invoque a BTRV de la forma siguiente:

```
int Btr (int codop)
{
    return BTRV (codop, ...  );
}
```

Esto implica declarar como globales las variables Btrieve que se pasan a BTRV. Para el programa anterior la función es

```
int Btr (int codop)
{
    return BTRV (codop, bloque, &datos, &lonreg, keyval, 0);
}
```

y, por ejemplo, la operación INSERTAR se ejecuta del modo

```
st = Btr (INSERTAR);
```

habiendo declarado como globales las variable **bloque**, **datos**, **lonreg** y **keyval**.

Esta nomenclatura simplifica la escritura de programas que usan ficheros Btrieve. Habitualmente se tendrán que manejar varios ficheros Btrieve en un mismo programa. Puede utilizarse una sola función **Btr** para hacer las llamadas Btrieve de todos los ficheros, sin más que hacer una cuidadosa declaración de variables. Veamos un ejemplo con un programa que usa dos ficheros Btrieve llamados ARTIC.BTR y PEDIDOS.BTR cuya descripción es la siguiente:

ARTIC.BTR Maestro de Artículos Indexado Btrieve			**PEDIDOS.BTR Pedidos a proveedor Indexado Btrieve**		
Campo	**Descripción**	**Tipo**	**Campo**	**Descripción**	**Tipo**
cod	Código del Artículo	char(7)	**pro**	Código Proveedor	char(7)
des	Descripción	char(41)	**art**	Código del Artículo	char(7)
exi	Existencia	int	**can**	Cantidad a pedir	int
min	Mínimo	int	**pre**	Precio de compra	int
opt	Óptimo	int			
pco	Precio de compra	int			
pvp	Precio de Venta	int			
pro	Código Proveedor	char(7)			
El campo **cod** es la única clave de acceso al fichero. No se admiten duplicados ni modificaciones de claves. Los campos de tipo *char* incluyen el nulo final.			La única clave está formada por los campos **pro** y **art**. No se admiten duplicados ni modificaciones de claves. Los campos de tipo *char* incluyen el nulo final		

El programa inserta un registro en PEDIDOS.BTR por cada registro de ARTIC.BTR cuya existencia está bajo mínimos. En ese caso genera un registro en el que la cantidad a pedir (campo **can** de PEDIDOS.BTR) se obtiene como la diferencia **opt - exi** de ARTIC.BTR.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "c:\btr\turcbtrv.c"

#define  ABRIR        0
#define  CERRAR       1
#define  INSERTAR     2
#define  SIGUIENTE    6
#define  INFERIOR     12

#define  AR           0
#define  PD           1

#define  NF           2
```

```
struct REGART {
  char cod[7];
  char des[41];
  unsigned exi;
  unsigned min;
  unsigned opt;
  unsigned pco;
  unsigned pvp;
  char pro[7];
};

struct REGPED {
  char pro[7];
  char art[7];
  unsigned can;
  unsigned pre;
};

union REGTOT {
    struct REGART ar;
    struct REGPED pd;
};

int Btr (int op, int fich);
void BtrError (int st, int op, int fich);
void Cerrar (int fich);

char fichero[NF][13] = { "ARTIC.BTR", "PEDIDOS.BTR" };
union REGTOT reg[NF];
char bloque[NF][128];
int longitud[NF] = { sizeof (struct REGART), sizeof (struct REGPED) };
char keyval[NF][14];

void main (void)
{
  int st, i;

  for (i = 0; i < NF; i++) {
    strcpy (keyval[i], fichero[i]);
    st = Btr (ABRIR, i);
    if (st) {
      BtrError (st, ABRIR, i);
      Cerrar (i);
      exit (1);
    }
  }

  st = Btr (INFERIOR, AR);
  while (!st) {
    if (reg[AR].ar.exi < reg[AR].ar.min) {
      strcpy (reg[PD].pd.pro, reg[AR].ar.pro);
      strcpy (reg[PD].pd.art, reg[AR].ar.cod);
      reg[PD].pd.can = reg[AR].ar.opt - reg[AR].ar.exi;
      reg[PD].pd.pre = reg[AR].ar.pco;

       st = Btr (INSERTAR, PD);
       if (st) BtrError (st, INSERTAR, PD);
    }
```

```
    st = Btr (SIGUIENTE, AR);
  }
  if (st != 9) BtrError (st, SIGUIENTE, AR);

  Cerrar (NF);
}

int Btr (int op, int fich)
{
  return BTRV (op, bloque[fich], &reg[fich], &longitud[fich], keyval[fich], 0);
}

void BtrError (int st, int op, int fich)
{
  printf ("\n%s: Error %d en operación %d", fichero[fich], st, op);
  printf ("\nPulse una tecla ...");
  while (!getch ()) getch ();
}

void Cerrar (int fich)
{
  register int i;
  int st;

  for (i = 0; i < fich; i++) {
    st = Btr (CERRAR, i);
    if (st) BtrError (st, CERRAR, i);
  }
}
```

Códigos de error Btrieve

La siguiente tabla muestra los valores de retorno posibles devueltos por BTRV si la operación no tiene éxito:

st	***DESCRIPCIÓN***	***st***	***DESCRIPCIÓN***
1	*Código de operación inválido*	*12*	*Fichero no encontrado*
2	*Error de E/S*	*13*	*Error de extensión*
3	*Fichero no abierto*	*14*	*Error de preapertura*
4	*Clave no encontrada*	*15*	*Error de preimagen*
5	*No se admiten claves duplicadas*	*16*	*Error de expansión*
6	*Número de clave incorrecto*	*17*	*Error de cierre*
7	*Número de clave cambiado*	*18*	*Disco lleno*
8	*Posicionamiento inválido*	*19*	*Error irrecuperable*
9	*Fin de fichero*	*20*	*No está cargado BTRIEVE*
10	*No se permite modificar la clave*	*21*	*Error en área de clave*
11	*Nombre de fichero inválido*	*22*	*Error en tamaño de registro*
st	***DESCRIPCIÓN***	***st***	***DESCRIPCIÓN***
23	*Bloque de posición < 128 bytes*	*48*	*Error en secuencia alternativa*

24	*Error de tamaño de página*	*49*	*Error en tipo de clave*
25	*Error de E/S en creación*	*50*	*Palabra de paso ya asignada*
26	*Número de claves incorrecto*	*51*	*Palabra de paso inválida*
27	*Posición de la clave incorrecta*	*52*	*Error al grabar desde memoria*
28	*Longitud del registro incorrecta*	*53*	*Interfase inválida (Versión)*
29	*Longitud de clave incorrecta*	*54*	*No se puede leer página variable*
30	*No es un fichero Btrieve*	*55*	*Error de autoincremento*
31	*Error en extensión*	*56*	*Índice incompleto*
32	*Error de E/S en extensión*	*57*	*Error de memoria expandida*
34	*Nombre inválido par extensión*	*58*	*Área de compresión pequeña*
35	*Error de directorio*	*59*	*Fichero existente*
36	*Error en transacción*	*80*	*Conflicto*
37	*Transacción ya activa*	*81*	*Bloqueo inválido*
38	*Error de E/S en fichero de trans.*	*82*	*Posicionamiento perdido*
39	*Error de fin/aborto de transac.*	*83*	*Lectura fuera de transacción*
40	*Demasiados ficheros en transac.*	*84*	*Registro bloqueado*
41	*Operación no permitida*	*85*	*Fichero bloqueado*
42	*Acceso acelerado incompleto*	*86*	*Tabla de ficheros llena*
43	*Dirección inválida (Operación 23)*	*87*	*Tabla de identificadores llena*
44	*Vía de acceso con clave nula*	*88*	*Error en modo de apertura*
45	*Error en indicadores de clave*	*93*	*Tipo de bloqueo incompatible*
46	*Acceso denegado*	*94*	*Error de alto nivel*
47	*Demasiados ficheros abiertos*		

Ejercicios

1. En un Centro de Enseñanza se dispone de una Base de Datos con información sobre las faltas de asistencia a clase de sus alumnos. Dos de los ficheros de dicha Base de Datos se describen a continuación:

ALUMNOS.BTR Datos de Alumnos
Indexado Btrieve

Campo	Descripción	Tipo	Clave
gru	Clave del grupo	char(6)	K0, K1
nom	Apellidos y nombre del alumno	char(31)	K1, K2
fal	Faltas mes/asignatura	int 12x8	

- La clave **K0** es el campo **gru**.Admite duplicados.
- La clave **K1** está formada por los campos **gru** y **nom**. No admite duplicados.
- La clave **K2** está formada por el campo **nom**. No admite duplicados.
- Las tres claves son de tipo *string*.
- El campo **gru** sólo puede almacenar los valores: 1FP2B, 2FP2A, 2FP2B, 2FP2C, 3FP2A y 3FP2B.
- El campo **fal** es una matriz entera de 12 filas por 8 columnas que debe interpretarse como sigue: El elemento **fal[i][j]** se refiere al número de faltas del mes **i** en la asignatura **j**.

GRUPO.BTR Datos del Grupo
Indexado Btrieve

Campo	Descripción	Tipo	Clave
gru	Clave del grupo	char(6)	K0
asg	Abreviatura de las asignaturas	char 8x6	

- La clave **K0** es el campo **gru**. No admite duplicados. Es de tipo *string*.
- El campo **gru** sólo puede almacenar los valores: 1FP2B, 2FP2A, 2FP2B, 2FP2C, 3FP2A y 3FP2B.
- El campo **asg** es una matriz de 8 cadenas de 6 caracteres que almacenan las abreviaturas de los nombres de las asignaturas del grupo. Por ejemplo, PRACT para Prácticas.

Escribe un programa que solicite por teclado un grupo y un mes, y realice un informe impreso de las faltas de todos los alumnos del grupo durante ese mes, por orden alfabético y totalizando por alumno y asignatura. El formato de dicho informe debe ser el siguiente:

GRUPO: XXXXX
FALTAS DEL MES DE: XXXXXXXXXX

Apellidos y Nombre	abr1	abr2	abr3	abr4	abr5	abr6	abr7	abr8	TOT
xxxxxxxxxxxxxxxxxxxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
xxxxxxxxxxxxxxxxxxxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
TOTALES:	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx

(Los títulos **abr1**, ..., **abr8** son los nombres abreviados de las asignaturas).

2. En la biblioteca de un Instituto se controla el préstamo de libros mediante una Base de Datos formada por los siguientes ficheros Btrieve:

PRESTAMO.BTR Libros prestados
Indexado Btrieve

Campo	Descripción	Tipo	Clave
fec	Fecha del préstamo	char(9)	K0
isbn	ISBN	char(11)	
exp	Nº de expediente del alumno	char(6)	
dev	Devuelto S/N	char	k1

- La clave **K0** es el campo **fec**. Admite duplicados. Tipo *string*.
- La clave **K1** es el campo **dev**. Admite duplicados. Tipo *string*.
- El campo **fec** almacena la fecha del préstamo en formato AAAAMMDD.
- El campo **exp** almacena sólo dígitos numéricos y está completado con ceros a la izquierda. Por ejemplo, el expediente 65 se almacena como 00065.
- El campo **dev** almacena el carácter **S** si el libro ha sido devuelto, y el carácter **N** en caso contrario.

LIBROS.BTR Maestro de libros
Indexado Btrieve

Campo	Descripción	Tipo	Clave
isbn	ISBN	char(11)	K0
tit	Título	char(51)	
aut	Autor	char(31)	
edi	Editorial	char(31)	

- La clave **K0** es el campo **isbn**. No admite duplicados. Es de tipo *string*.

ALUMNOS.BTR	Datos de alumnos Indexado Btrieve		
Campo	**Descripción**	**Tipo**	**Clave**
exp	Nº de expediente del alumno	char(6)	K0
nom	Nombre	char(31)	
gru	Grupo	char(11)	

- La clave **K0** es el campo **exp**. No admite duplicados. Es de tipo *string*.
- El campo **exp** almacena sólo dígitos numéricos, y está completado a la izquierda con ceros.

Teniendo en cuenta que los libros pueden estar prestados un máximo de 7 días, construye un programa que realice un listado de aquellos libros que no han sido devueltos en el plazo indicado. Para ello se asumirá como fecha del día la del sistema. El listado imprimirá 6 fichas por página con el siguiente formato para cada ficha:

Alumno: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	Grupo: xxxxxxxxxx
Título:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Autor:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Editorial:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Fecha del préstamo: xx de xxxxxxxxxx de xxxx	

3. Disponemos de un fichero secuencial llamado MOVIM.TMP en el que se registran todos los movimientos de artículos en los almacenes de una empresa. La descripción de este fichero es la siguiente:

MOVIM.TMP	E/S de artículos Secuencial		
Campo	**Descripción**	**Tipo**	**Clave**
alm	Código de almacén	char(3)	
art	Código de artículo	char(7)	
can	Cantidad	unsigned	
ope	Operación E/S/B	char	

- El campo **alm** contiene dos digitos numéricos. Así, el almacén 1 se codifica como "01".
- El campo **ope** almacena una **E** si es una entrada de material, una **S** si es una salida, y una **B** si el artículo debe darse de baja en su almacén.
- El archivo no está ordenado bajo ningún criterio.

A partir de este archivo se desea actualizar las existencias en el Maestro de Artículos, ARTIC.BTR, cuya descripción es la siguiente:

ARTIC.BTR	Maestro de Artículos Indexado Btrieve

Campo	Descripción	Tipo	Clave
alm	Código de almacén	char(3)	K0
art	Código de artículo	char(7)	K0, K1
des	Descripción del artículo	char(41)	
exi	Existencia	int	
min	Mínimo	unsigned	
opt	Óptimo	unsigned	
pvp	Precio de venta	unsigned	
pco	Precio de compra	unsigned	

- La clave **K0** está formada por los campos **alm** y **art**. No se admiten duplicados ni modificaciones. Es de tipo *string*.
- La clave **K1** coincide con el campo **art**. Admite duplicados. Es de tipo *string*.
- El campo **alm** contiene dos digitos numéricos. Así, el almacén 1 se codifica como "01".

Escribe un programa que efectúe dicha actualización de existencias. Para ello has de tener en cuenta:

Si el campo ope de MOVIM.TMP almacena...	La operación a realizar en ARTIC.BTR es:
E	exi = exi + can
S	exi = exi - can
B	Borrar el registro correspondiente

12
Compilación y enlazado

Introducción

Turbo C proporciona un entorno integrado de desarrollo de aplicaciones (IDE: **I**ntegrated **D**evelopment **E**nvironment) que permite, mediante un sencillo sistema de menús, editar, compilar, enlazar, ejecutar y depurar un programa sin abandonar dicho entorno. Pero, además, puede usarse cualquier editor de texto para crear nuestros programas, y realizar la compilación y enlazado desde la línea de órdenes del DOS. En este capítulo se muestra cómo hacerlo.

Modelos de memoria

Cuando un programa C se carga en memoria debemos distinguir en él cuatro partes:

- **Código del programa**.
- **Datos estáticos**: Datos con tiempo de vida global a los que se asigna memoria en los segmentos de datos del programa.
- **Datos dinámicos**: Datos a los que se les asigna memoria en tiempo de ejecución, a medida que se necesita. Esta memoria no se toma de los segmentos de datos, sino de una zona aparte denominada **montón** (heap).
- **Datos automáticos**: Datos locales. Se les asigna memoria en la pila cuando se ejecuta el bloque del programa en que están definidos.

En C hay 6 formas posibles de organizar estas partes del programa en la memoria. Estos 6 modelos de memoria se denominan:

- Diminuto (Tiny)
- Pequeño (Small)
- Mediano (Medium)
- Compacto (Compact)
- Grande (Large)
- Enorme (Huge)

Modelo Diminuto (Tiny)

Tanto el código como los datos ocupan el mismo segmento, que es único. Por tanto, un programa compilado y enlazado bajo este modelo no puede superar los 64 Kb. Estos programas no asignan memoria para su pila, sino que utilizan la del DOS. Por ello, en el enlazado se muestra la advertencia *Warning: no stack*. Los registros CS, DS, ES y SS no cambian durante el programa y tienen todos el mismo valor. Los programas .EXE generados así pueden convertirse a .COM mediante la utilidad del DOS EXE2BIN, cuya sintaxis es

exe2bin prog.exe prog.com

Modelo Pequeño (Small)

Es el modelo usado por defecto. Puede contener 64 Kb de código y otros 64 Kb para todos los datos. Durante la ejecución del programa los valores de CS, DS, ES y SS no cambian, pero a diferencia con el modelo diminuto, CS y DS contienen valores diferentes.

Modelo Mediano (Medium)

Todos los datos se almacenan en el mismo segmento, por lo que no pueden superar los 64 Kb. Por el contrario, el código puede ocupar hasta 1 Mb (16 segmentos de 64 Kb).

Modelo Compacto (Compact)

El código está limitado a un segmento de 64 Kb, pero los datos pueden ocupar 1 Mb. De los 16 segmentos de datos, sólo uno (64 Kb) puede ser utilizado para los datos estáticos.

Modelo Grande (Large)

Tanto el código como los datos pueden ocupar 1 Mb cada uno, pero los datos estáticos están limitados a 64 Kb.

Modelo Enorme (Huge)

Mantiene las capacidades del modelo Grande, pero sin la limitación de 64 Kb para datos estáticos.

Para decidir qué modelo de memoria se debe utilizar, hay que tener en cuenta los siguientes aspectos:

- Cantidad de código y datos del programa.
- Los programas generados con los modelos de memoria más pequeños se ejecutan más rápidamente, por lo que hay que elegir el modelo de memoria más pequeño posible.
- Cualquier archivo .OBJ resultante de compilar un módulo simple no puede sobrepasar los 64 Kb de código ni los 64 Kb de datos, puesto que cada módulo compilado ha de encajar en un segmento de memoria. Cuando se utilizan los modelos Mediano, Grande y Enorme los programas pueden llegar hasta 1 Mb, pero a base de módulos no superiores a 64 Kb. Si el código de algún módulo supera este límite, se genera un mensaje

  Too much code defined in file

  Si los datos estáticos superan los 64 Kb, se genera el mensaje

  To much global data defined in file

  En ambos casos hay que dividir los módulos en ficheros más pequeños.

El compilador TCC

El programa TCC.EXE es el compilador independiente del entorno. Mediante TCC podemos obtener directamente programas .EXE, o bien archivos .OBJ para ser enlazados posteriormente. La sintaxis de TCC es

tcc <*opciones*> *fichero(s)*

La siguiente tabla muestra las opciones posibles

OPCIÓN	***SIGNIFICADO***
-A	*Sólo reconoce palabras clave ANSI.*
-a	*Alinea los datos en frontera de palabra.*
-a-	*Alinea los datos en frontera de byte.*
-B	*Acepta código ensamblador y llama a MASM para procesarlo.*
-C	*Acepta comentarios anidados.*
-c	*Compila a .OBJ (sin enlazar).*
*-D**nom***	*Define el nombre (**nom**) de una constante, como si fuera una directiva **#define** dentro del programa, para ser usada con la directiva **#if defined**.*
*-D**nom**=**cad***	*Hace lo mismo que una directiva **#define nom cad**.*
-d	*Asigna la misma posición de memoria a cadenas idénticas del código fuente.*
*-e**nom***	*Establece el nombre del fichero .EXE resultante.*
-f	*Usa la emulación de coma flotante.*

-f-	*Pone en off la emulación de coma flotante.*
-f87	*Usa el coprocesador matemático sin generar código de emulación.*
-G	*Optimiza el código para velocidad en lugar de para tamaño.*
*-g**N***	*Detiene la compilación tras **N** advertencias (Warnings).*
*-l**ruta***	*Establece la **ruta** para los archivos **#include**.*
*-i**N***	*Especifica la longitud de los identificadores.*
*-j**N***	*Detiene la compilación tras **N** errores.*
-K	*Establece por defecto para el tipo **char** el modificador **unsigned**.*
-K-	*Establece por defecto para el tipo **char** el modificador **signed**.*
*-L**ruta***	*Establece la **ruta** para las bibliotecas.*
-M	*Crea archivo de mapa de memoria.*
*-m**x***	*Establece un modelo de memoria según el carácter **x** (Pág. 198).*
-N	*Comprueba el desbordamiento de pila.*
*-n**ruta***	*Establece el directorio para los ficheros de salida (.OBJ y .EXE).*
-O	*Optimiza el código para tamaño en lugar de para velocidad.*
*-o**nom***	*Especifica el nombre del archivo .OBJ, sin llamar al enlazador.*
-p	*Usa el convenio de llamada de Pascal.*
-p-	*Usa el convenio de llamada de C.*
-r	*Permite a las variables **register** usar los registros SI y DI.*
-r-	*Pone a off la opción anterior.*
-S	*Genera como salida un fichero fuente en ensamblador (.ASM).*
*-U**nom***	*Deja sin definir las definiciones previas dadas por la opción -D.*
-w	*Muestra mensajes de advertencia.*
-w-	*No muestra mensajes de advertencia.*
*-w**xxx***	*Pone en on las advertencias de error **xxx**, código de 3 letras que especifica el tipo (ver Guía del Programador).*
*-w**xxx**-*	*Pone en off las advertencias de error **xxx**.*
-Y	*Fabrica la pila estándar.*
-y	*Incluye números de línea en el código.*
-Z	*Activa la optimización de registro. El programa recuerda el contenido de los registros y, si es posible, vuelve a utilizarlos en lugar de volver a cargar el valor de memoria.*
-z	*Especifica los nombres de los segmentos (Guía de Referencia).*
-1	*Genera instrucciones 80186/80286.*
-1-	*No genera instrucciones 80186/80286.*
-2	*Genera instrucciones 80286 en modo protegido.*

Por ejemplo, la orden

tcc -Ic:\tc\include -mc -eprog fich1 fich2 fich3.obj

contiene las opciones

- **-I** indica que los archivos **#include** se buscarán en **c:\tc\include**.
- **-m** especifica el modelo de memoria a usar. En este caso, **-mc** selecciona el modelo Compacto.
- **-e** especifica que el fichero resultante de la compilación será **PROG.EXE**.

La orden anterior compila los archivos FICH1.C y FICH2.C y obtiene los archivos FICH1.OBJ y FICH2.OBJ correspondientes que enlaza después con FICH3.OBJ para obtener PROG.EXE.

En los procesos de compilación hay opciones que se repiten en la mayoría de las ocasiones y que debemos teclear cada vez. Para evitar esto puede crearse un archivo de configuración denominado TURBOC.CFG con las opciones más comunes. TCC busca siempre este archivo en el directorio actual, y si no lo encuentra lo busca en el directorio en que está ubicado TCC, generalmente el directotio **c:\tc\bin**. Por ejemplo, el archivo TURBOC.CFG puede tener las siguientes líneas:

```
-Ic:\tc\include
-Lc:\tc\lib
-ms
-G
```

con lo que TCC buscará los archivos **#include** en **c:\tc\include** (-I), las bibliotecas en **c:\tc\lib** (-L), seleccionará el modelo Compacto de memoria (-mc) y optimizará el código para velocidad (-G).

El enlazador TLINK

Cuando ejecutamos TCC sin la opción **-c**, los archivos son compilados y enlazados para obtener el ejecutable. Pero puede usarse TCC de forma que sólo compile, sin enlazar, usando la opción **-c**. Por ejemplo

tcc -Ic:\tc\include -mc -G -c *fuentes*

compila los archivos de la lista *fuentes* a .OBJ sin enlazarlos después. Posteriormente debe usarse el enlazador independiente TLINK, cuya sintaxis es

tlink <*opciones*> *archivos_OBJ, archivo_EXE, archivo_MAP, bibliotecas*

Las opciones de TLINK se muestran a continuación:

OPCIÓN	***SIGNIFICADO***
/c	*Los símbolos PUBLIC y EXTERN distinguen mayúsculas y minúsculas.*
/d	*Mostrar una advertencia si hay símbolos duplicados en las bibliotecas.*
/e	*No usar palabras clave extendidas.*
/i	*Inicializar todos los segmentos*
/l	*Incluir los números de línea de la fuente para la depuración. Es necesario usar la opción* ***-y*** *con TCC.*
/m	*Incluir los símbolos públicos en el archivo de mapa.*
/n	*Ignorar las bibliotecas por defecto.*
/s	*Incluir un mapa detallado de los segmentos en el archivo de mapa.*
/t	*Crear un archivo .COM (sólo para el modelo Diminuto de memoria).*
/v	*Incluir toda la información completa de depuración.*
/x	*No crear un archivo de mapa.*
/3	*Usar procesamiento completo de 32 bits.*

archivos_OBJ es el nombre de todos los archivos .OBJ que se quieren enlazar (separados por blancos). El primero de ellos ha de ser un módulo de iniciación de Turbo C. Estos módulos están normalmente en el directorio **c:\tc\lib** y hay uno para cada modelo de memoria. El nombre de estos archivos es C0**x**.OBJ, siendo **x** un carácter que especifica el modelo de memoria usado. Los valores posibles de x se especifican en la tabla siguiente:

x	***MODELO***
t	*Diminuto (Tiny)*
s	*Pequeño (Small)*
m	*Mediano (Medium)*
c	*Compacto (Compact)*
l	*Grande (Large)*
h	*Enorme (Huge)*

Así, el archivo **c:\tc\lib\c0l.obj** es el módulo de iniciación de Turbo C para el modelo Grande de memoria.

archivo_EXE es el nombre del programa ejecutable a generar. Si no se especifica, se asume el nombre del primer archivo .OBJ que no sea un módulo de iniciación de Turbo C.

archivo_MAP es el nombre de un archivo que contiene un mapa de memoria del programa. Tiene extensión .MAP y, si no se especifica, asume como nombre el del archivo .EXE. El fichero de mapa no se crea TLINK lleva la opción **/x**.

bibliotecas es una lista de bibliotecas a enlazar. Aparte de las bibliotecas de usuario, debe enlazarse el correspondiente archivo de biblioteca estándar. Existe uno para cada modelo de memoria y se almacenan en **c:\tc\lib**. El nombre de estos archivos es C**x**.LIB, siendo **x** un carácter de la tabla anterior que especifica el modelo de memoria. Además de estas bibliotecas, si el programa usa operaciones en coma flotante debe enlazarse el archivo FP87.LIB si se dispone de coprocesador matemático. En caso contrario se enlazará el archivo EMU.LIB. Ambos archivos se encuentran, normalmente, en **c:\tc\lib**. Por último están los archivos que almacenan las rutinas matemáticas, que se llaman MATH**x**.LIB, siendo **x** el carácter que identifica el modelo de memoria.

Con las siguientes órdenes se compila el archivo PROG.C (que usa operaciones en coma flotante) a PROG.OBJ y posteriormente se enlaza a PROG.EXE sin generar fichero de mapa. Se usa el modelo Pequeño de memoria.

```
tcc -Ic:\tc\include -ms -G -c prog
tlink /x c:\tc\lib\c0s prog, , , c:\tc\lib\emu c:\tc\lib\cs
```

El enlazador TLINK también permite la sintaxis

tlink @*fichero_respuesta*

siendo *fichero_respuesta* un archivo de texto con todas las órdenes. Por ejemplo, un fichero de respuesta puede ser

```
/x +
c:\tc\lib\c0s prog1 prog2, +
prog3, +
, +
c:\tc\lib\cs
```

donde los signos + indican continuación en la línea siguiente.

El bibliotecario TLIB

Una biblioteca es un archivo .LIB que almacena módulos .OBJ. Un programa puede hacer llamadas a funciones que han sido compilados individualmente e incluidos en una biblioteca. Para ello se enlaza el programa con la biblioteca y en este proceso de enlazado sólo se tomarán de la biblioteca los módulos objeto que se usen. El bibliotecario de Turbo C se llama TLIB y la sintaxis de su uso es

tlib *nombre_biblioteca [op]modulo_OBJ*

donde *nombre_biblioteca* es el nombre del archivo .LIB, y *modulo_OBJ* es el módulo sobre el que se realiza la operación indicada por *op*, que puede ser una de las siguientes:

OPERADOR	***ACCIÓN***
+	*Añade un módulo a la biblioteca*
-	*Elimina un módulo de la biblioteca*
*	*Extrae un archivo .OBJ de la biblioteca*
-+ ó +-	*Reemplaza el módulo especificado con una nueva copia*
-* ó *-	*Extrae el módulo especificado y lo elimina de la biblioteca*

Por ejemplo, la orden

tlib mibiblio +funcion1 -funcion2

añade el archivo FUNCION1.OBJ a la biblioteca MIBIBLIO.LIB y elimina de la misma el archivo FUNCION2.OBJ.

Análogamente a TLINK, TLIB admite un fichero de respuestas, en cuyo caso la sintaxis es

tlib *nombre_biblioteca* **@***fichero_respuestas*

La utilidad MAKE

El programa MAKE automatiza la recompilación de programas formados por varios módulos. Básicamente, MAKE comprueba si alguno de los módulos de que está compuesto el programa ha sido modificado y realiza las acciones de compilación, enlazado u otras necesarias para la actualización del programa. Para ello, MAKE parte de un *archivo de construcción*, que está compuesto por sentencias como las siguientes:

archivo_objetivo1: lista_de_archivos_dependientes
    secuencia de órdenes

archivo_objetivo2: lista_de_archivos_dependientes
    secuencia de órdenes
...
...
archivo_objetivoN: lista_de_archivos_dependientes
    secuencia de órdenes

La línea que contiene *archivo_objetivo* debe empezar en la primera columna, y la *secuencia de órdenes* debe comenzar por una tabulador. Entre cada bloque debe haber, al menos, una línea en blanco.

Un *archivo objetivo* es aquél que se obtiene a partir de los *archivos dependientes*. Por ejemplo, PROG.C es un archivo dependiente de PROG.OBJ pues para crear PROG.OBJ hay que compilar PROG.C.

Por ejemplo, si el programa PROG está formado por los módulos MOD1 y MOD2, el archivo de construcción de MAKE podría tener el siguiente aspecto:

```
PROG.EXE: MOD1.OBJ  MOD2.OBJ
    tlink /x c0s mod1 mod2, prog,, cs

MOD1.OBJ: MOD1.C
    tcc -Ic:\tc\include -G -c -ms mod1

MOD2.OBJ: MOD2.C
    tcc -Ic:\tc\include -G -c -ms mod2
```

El programa MAKE busca un archivo de construcción llamado MAKEFILE. Basta entonces con teclear desde la línea de órdenes del DOS

make

Si el archivo tiene un nombre diferente, digamos PROG.MAK, debe usarse la opción **-f** como sigue:

make -fPROG.MAK

MAKE tiene varias opciones disponibles, que se pueden consultar mediante la opción **-?** (**make -?**).

Un ejemplo sencillo

Vamos a aplicar lo explicado en los apartados del capítulo con un ejemplo sencillo. En primer lugar, supondremos que nuestro directorio de trabajo es C:\TRABAJO y que en él tenemos el archivo TURBOC.CFG siguiente:

```
-Ic:\tc\include
-Lc:\tc\lib
-G
```

y el archivo EJEMPLO.C siguiente:

```
/* Programa EJEMPLO.C */

#include <stdio.h>
#inlcude <conio.h>

float suma (float, float);
float producto (float, float);

void main (void)
{
    float a, b;

    clrscr ();
    printf ("Teclea dos números: ");
    scanf ("%f %f", &a, &b);

    printf ("\nSuma: %f", suma (a, b));
    printf ("\nProducto: %f", producto (a, b));
}

float suma (float x, float y);
{
    return x + y;
}

float (producto (float x, float y)
{
    return x * y;
}
```

Para obtener el programa EJEMPLO.EXE basta con hacer:

```
C:\TRABAJO> tcc -ms ejemplo
Turbo C++ Version 1.00 Copyright (c) 1990 Borland International
ejemplo.c:
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International

    Available memory xxxxxx

C:\TRABAJO>
```

Después de esto, el directorio tendrá los archivos

ejemplo.c
turboc.cfg
ejemplo.obj
ejemplo.exe

El archivo EJEMPLO.OBJ es el objeto resultante de la compilación, y EJEMPLO.EXE es el ejecutable que se obtiene después del enlazado con los modulos de iniciación y bibliotecas de Turbo C.

Un inconveniente que podemos encontrar en este método es que las funciones **suma()** y **producto()** sólo sirven para el programa EJEMPLO.C. Si quisieramos usar estas funciones en otros programas, tendríamos que teclearlas de nuevo al editarlos. Esto no es demasiado problema en funciones como las del ejemplo, pero resulta muy incómodo para funciones con más código o que usemos habitualmente en nuestros programas. La solución consiste en guardar en archivos diferentes el programa EJEMPLO.C (la parte que contiene las declaraciones y la función **main()**) y las funciones **suma()** y **producto()**, por ejemplo, en archivos SUMA.C y PROD.C.

```
/* Programa EJEMPLO.C */

#include <stdio.h>
#inlcude <conio.h>

float suma (float, float);
float producto (float, float);

void main (void)
{
    float a, b;

    clrscr ();
    printf ("Teclea dos números: ");
    scanf ("%f %f", &a, &b);

    printf ("\nSuma: %f", suma (a, b));
    printf ("\nProducto: %f", producto (a, b));
}
/* Archivo SUMA.C: Función suma */
```

```
float suma (float x, float y);
{
    return x + y;
}
```

```
/* Archivo PROD.C: Función producto */

float (producto (float x, float y)
{
    return x * y;
}
```

Ahora compilaremos por separado cada uno de estos módulos y, posteriormente, haremos el enlazado.

C:\TRABAJO> tcc -c -ms ejemplo
Turbo C++ Version 1.00 Copyright (c) 1990 Borland International
ejemplo.c:

Available memory *xxxxxx*

C:\TRABAJO> tcc -c -ms prod
Turbo C++ Version 1.00 Copyright (c) 1990 Borland International
prod.c:

Available memory *xxxxxx*

C:\TRABAJO> tcc -c -ms suma
Turbo C++ Version 1.00 Copyright (c) 1990 Borland International
suma.c:

Available memory *xxxxxx*

C:\TRABAJO> tlink /x c0s ejemplo prod suma, ejemplo,, emu maths cs[1]

Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International

C:\TRABAJO>

Después de esto, en el directorio C:\TRABAJO tendremos los siguientes archivos:

ejemplo.c
prod.c
suma.c

[1] Para no tener que indicar en los archivos **c0s**, **emu**, **maths** y **cs** el camino de búsqueda **c:\tc\lib** (lo que puede dar problemas, al ser la línea de órdenes demasiado larga) es conveniente ejecutar previamente la orden **APPEND c:\tc\lib** (o bien incluirla en el archivo AUTOEXEC.BAT). Esta orden es similar a la orden PATH, en el sentido que indica a las aplicaciones dónde buscar los archivos que no se encuentren en el directorio actual.

turboc.cfg
ejemplo.obj
prod.obj
suma.obj
ejemplo.exe

Los *.OBJ son los resultantes de las compilaciones de los *.C, y EJEMPLO.EXE es el ejecutable final.

Este método de trabajo permite tener por separado nuestras funciones compiladas (en formato .OBJ), disponibles para ser enlazadas con cualquier programa. Además, si modificamos uno de los módulos basta con recompilarlo y, posteriormente, hacer el enlazado, sin recompilar los módulos que no han sido modificados. Por ejemplo, si hacemos algún cambio en EJEMPLO.C, bastará con hacer:

```
C:\TRABAJO> tcc -c -ms ejemplo
Turbo C++ Version 1.00 Copyright (c) 1990 Borland International
ejemplo.c:

    Available memory xxxxxx

C:\TRABAJO> tlink /x c0s ejemplo prod suma, ejemplo,, emu maths cs
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International

C:\TRABAJO>
```

Pero para estos casos es muy útil el programa MAKE. Creamos el siguiente archivo de construcción MAKE, al que llamaremos EJEMPLO.MAK:

```
ejemplo.exe: ejemplo.obj prod.obj suma.obj
    tlink /x c0s ejemplo prod suma, ejemplo,, emu maths cs

ejemplo.obj: ejemplo.c
    tcc -c -ms ejemplo

prog.obj: prod.c
    tcc -c -ms prod

suma.obj: suma.c
    tcc -c -ms suma
```

Suponiendo que tenemos en el directorio C:\TRABAJO los siguientes archivos

ejemplo.c
prod.c
suma.c

turboc.cfg
ejemplo.mak

después de

C:\TRABAJO> make -fejemplo.mak

tendremos, además, los archivos

ejemplo.obj
prod.obj
suma.obj
ejemplo.exe

Si se modifica alguno de los módulos, digamos SUMA.C, y repetimos la orden

C:\TRABAJO> make -fejemplo.mak

observaremos que MAKE se encarga de recompilar SUMA.C y enlazar el SUMA.OBJ resultante con el resto de módulos objeto y bibliotecas, sin recompilar EJEMPLO.C ni PROD.C.

Otra posibilidad consiste en tener almacenados nuestros módulos objeto en una biblioteca. Supongamos que tenemos en C:\TRABAJO los archivos

ejemplo.c
prod.c
suma.c
turboc.cfg
ejemplo.obj
prod.obj
suma.obj

y hacemos

C:\TRABAJO> tlib milib +suma +prod
TLIB 3.0 Copyright (c) 1987, 1990 Borland International

C:\TRABAJO> del suma.obj *(no es necesario)*

C:\TRABAJO> del suma.obj *(no es necesario)*

Los archivos en C:\TRABAJO serán

ejemplo.c
prod.c
suma.c
turboc.cfg
ejemplo.obj

milib.lib

El enlazado se hace ahora

```
C:\TRABAJO> tlink /x c0s ejemplo, ejemplo,, emu maths cs milib
Turbo Link Version 3.0 Copyright (c) 1987, 1990 Borland International

C:\TRABAJO>
```

De este modo podemos tener una colección de módulos objeto disponible para cualquier programa mediante la biblioteca MILIB.LIB.

13
La biblioteca de funciones de Turbo C

Introducción

A continuación se describen algunas de las funciones de la biblioteca de Turbo C, agrupadas por categorías. Algunas ya han sido explicadas en capítulos precedentes, por lo que no se volverán a tratar aquí. No se recogen en este capítulo todas las funciones, sino sólo algunas más comúnmente usadas. Un conocimiento pormenorizado de todas ellas se encuentra en el Library Reference de Turbo C.

Funciones de E/S

Ya se han estudiado las siguientes funciones en el Capítulo 4:

fprintf	**getch**	**getche**	**gets**	**printf**	**putchar**
puts	**scanf**				

y en el Capítulo 11:

fclose	**feof**	**ferror**	**fgetc**	**fgets**	**fopen**
fprintf	**fputc**	**fputs**	**fread**	**fscanf**	**fseek**
fwrite	**getc**	**getw**	**perror**	**putc**	**putw**
rewind					

A continuación se explican otras funciones de E/S:

cgets

Prototipo: char *cgets (char **cad*);
Include: conio.h

Propósito: Esta función lee una cadena de caracteres del teclado, almacenando dicha cadena (y su longitud) en la posición apuntada por *cad*.

cgets lee caracteres hasta que se teclea ↵ o hasta un número máximo permitido de caracteres. El ↵ se reemplaza por un terminador nulo '\0'.

Antes de llamar a **cgets** hay que guardar en *cad[0]* la longitud máxima permitida para la cadena *cad*. Después de acabar, **cgets** almacena en *cad[1]* el número de caracteres tecleados. La cadena tecleada se almacena a partir de *cad[2]*. Todo esto implica que para *cad* deben reservarse 2 bytes más que el número máximo de caracteres.

Devuelve: Un puntero a *cad[2]*.

Portabilidad: Sólo IBM-PCs y compatibles.

remove

Prototipo: int remove (const char *_fichero_);

Include: stdio.h

Propósito: Borra el fichero especificado mediante *fichero*. El fichero debe estar cerrado.

Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y la variable global **errno** asume alguno de los valores siguientes:

ENOENT	No existe el fichero
EACCES	Permiso denegado

Portabilidad: Está disponible para UNIX y está definida en ANSI C.

rename

Prototipo: int rename (const char *_viejofich_, const char *_nuevofich_);

Include: stdio.h

Propósito: Cambia el nombre del fichero *viejofich* a *nuevofich*. Ambos deben estar almacenados en la misma unidad. Si el directorio de ambos no coincide, **rename** mueve el fichero de un directorio a otro. No se permiten comodines.

Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y la variable global **errno** asume uno de los valores siguientes:

ENOENT	No existe el fichero
EACCES	Permiso denegado
ENOTSAM	Dispositivo no coincide

Portabilidad: Es compatible con ANSI.C

sprintf

Prototipo: int sprintf (char **cad*, const char **format* [, *argumento*, ...]);
Include: stdio.h
Propósito: Esta función es idéntica a **printf** pero en lugar de realizar la salida a pantalla, la realiza sobre la cadena *cad*.
Devuelve: El número de caracteres almacenados en *cad* (sin contar el terminador nulo). Si ocurre un error, devuelve EOF.
Portabilidad: Esta disponible para UNIX y está definida en ANSI C. Es compatible con K&R.

sscanf

Prototipo: int sscanf (const char **cad*, const char **format* [,*direcc*, ...]);
Include: stdio.h
Propósito: Funciona igual que **scanf** pero no lee los datos del teclado sino de la cadena *cad*.
Devuelve: El número de campos leídos, convertidos y almacenados en sus variables. Si se intenta leer un fin de cadena, devuelve EOF.
Portabilidad: Esta disponible para UNIX y está definida en ANSI C. Es compatible con K&R.

tmpfile

Prototipo: FILE *tmpfile (void);
Include: stdio.h
Propósito: Abre un fichero temporal en modo **w+b**. El fichero se borra cuando se cierra o acaba el programa.
Devuelve: Si no hay error devuelve un puntero al canal asociado al fichero. En caso contrario devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

tmpnam

Prototipo: char *tmpnam (char **nombre*);
Include: stdio.h
Propósito: Genera un nombre no existente para un fichero temporal y lo almacena en *nombre*. Es responsabilidad del programador abrir, cerrar o borrar el fichero.
Devuelve: Si *nombre* es la cadena nula, devuelve un puntero a una variable estática interna. En caso contrario devuelve un puntero a *nombre*.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

Funciones de cadenas de caracteres

Funciones estudiadas en el Capítulo 7:

strcat	**strchr**	**strcmp**	**strcpy**	**strlen**	**strlwr**
strrev	**strset**	**strupr**			

Otras funciones se explican a continuación.

strerror

Prototipo: char *strerror (int *nerror*);
Include: stdio.h, string.h
Propósito: Devuelve un puntero a una cadena que almacena un mensaje de error asociado a *nerror*.
Devuelve: Un puntero a una cadena con un mensaje de error. La cadena se almacena en un buffer estático que se sobreescribe en cada llamada a la función.
Portabilidad: Es compatible con ANSI C.

stricmp

Prototipo: int stricmp (const char **cad1*, const char **cad2*);
Include: string.h
Propósito: Es una función idéntica a **strcmp** salvo en que no diferencia mayúsculas y minúsculas.
Devuelve: < 0 si *cad1* < *cad2*
= 0 si *cad1* = *cad2*
> 0 si *cad1* > *cad2*
Portabilidad: Sólo DOS.

strncat

Prototipo: char *strncat (char **dest*, const char **orig*, size_t *max*);
Include: string.h
Propósito: Copia un máximo de *max* caracteres de la cadena *orig* en la cadena *dest* y añade el terminador nulo.
Devuelve: Un puntero a *dest*.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

strncmp

Prototipo: int strncmp (const char **cad1*, const char **cad2*, size_t *max*);
Include: string.h
Propósito: Es idéntica a **strcmp** pero no compara cadenas completas, sino sólo los primeros *max* caracteres.
Devuelve: < 0 si *cad1* < *cad2*
= 0 si *cad1* = *cad2*
> 0 si *cad1* > *cad2*
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

strncpy

Prototipo: char strncpy (const char **dest*, const char **orig*, size_t *max*);
Include: string.h
Propósito: Copia hasta *max* caracteres de *orig* en *dest*. La cadena *dest* podría no tener un terminador nulo si el tamaño de *orig* es *max* o mayor.
Devuelve: Un puntero a *dest*.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

strnset

Prototipo: char *strnset (char **cad*, int *carac*, size_t *n*);
Include: string.h
Propósito: Pone el carácter *carac* en los primeros *n* bytes de *cad*. La operación finaliza cuando se han escrito *n* caracteres o se encuentra el terminador nulo.
Devuelve: Un puntero a *cad*.
Portabilidad: Sólo DOS.

strrchr

Prototipo: char *strrchr (const char **cad*, int *carac*);
Include: string.h
Propósito: Busca en dirección inversa el carácter *carac* en la cadena *cad*, considerando el terminador nulo como parte de *cad*.
Devuelve: Si encuentra *carac* devuelve un puntero a esa dirección. En caso contrario devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

strstr

Prototipo: char *strstr (const char **cad*, const char **subcad*);
Include: string.h
Propósito: Busca la primera aparición de *subcad* en *cad*.
Devuelve: Si *subcad* está en *cad* devuelve un puntero al lugar de *cad* en que comienza *subcad*. En caso contrario devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

Funciones de memoria

memchr

Prototipo: void *memchr (const void **bloque*, int *carac*, size_t *n*);
Include: string.h, mem.h
Propósito: Busca la primera aparición del carácter *carac* en los primeros *n* bytes del bloque de memoria apuntado por *bloque*.
Devuelve: Un puntero a la primera aparición de *carac* en *bloque*. Si no lo encuentra devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX System V y es compatible con ANSI C.

memcmp

Prototipo: int memcmp (const void **bloque1*, const void **bloque2*, size_t *n*);
Include: string.h, mem.h
Propósito: Compara dos bloques de memoria de *n* bytes de longitud.
Devuelve: < 0 si *bloque1* < *bloque2*
= 0 si *bloque1* = *bloque2*
> 0 si *bloque1* > *bloque2*
Portabilidad: Está disponible para UNIX System V y es compatible con ANSI C.

memcpy

Prototipo: void *memcpy (void **dest*, const void **orig*, size_t *n*);
Include: string.h, mem.h
Propósito: Copia un bloque de *n* bytes desde *orig* hasta *dest*. Es misión del programador controlar que *orig* y *dest* no se superponen.
Devuelve: Un puntero a *dest*.

Portabilidad: Está disponible para UNIX System V y es compatible con ANSI C.

memicmp

Prototipo: int memicp (const void *_bloque1_, const void *_bloque2_, size_t _n_);
Include: string.h, mem.h
Propósito: Compara los primeros *n* bytes de *bloque1* y *bloque2*, sin diferenciar mayúsculas y minúsculas.
Devuelve: < 0 si *bloque1* < *bloque2*
= 0 si *bloque1* = *bloque2*
> 0 si *bloque1* > *bloque2*
Portabilidad: Está disponible para UNIX System V.

memmove

Prototipo: void *memmove (void *_dest_, const void *_orig_, size_t _n_);
Include: string.h, mem.h
Propósito: Copia un bloque de *n* bytes desde *orig* hasta *dest*. Incluso cuando los bloques se superponen, los bytes en las posiciones superpuestas se copian correctamente.
Devuelve: Un puntero a *dest*.
Portabilidad: Está disponible para UNIX System V y es compatible con ANSI C.

memset

Prototipo: void *memset (void *_bloque_, int _carac_, size_t _n_);
Include: string.h, mem.h
Propósito: Pone en los *n* primeros bytes de *bloque* el carácter *carac*.
Devuelve: Un puntero a *bloque*.
Portabilidad: Está disponible para UNIX System V y es compatible con ANSI C.

movedata

Prototipo: void movedata (unsigned *seg1*, unsigned *off1*, unsigned *seg2*, unsigned *off2*, size_t *n*);
Include: mem.h, string.h
Propósito: Copia *n* bytes desde la dirección *seg1:off1* hasta la dirección *seg2:off2*.
Devuelve:
Portabilidad: Sólo DOS.

movmem

Prototipo: void movmem (void **orig*, void **dest*, unsigned *n*);
Include: mem.h
Propósito: Mueve un bloque de *n* bytes desde *orig* hasta *dest*. Incluso si *orig* y *dest* se superponen los bytes se mueven correctamente.
Devuelve:
Portabilidad: Sólo Turbo C++.

Funciones de caracteres

Funciones estudiadas en el Capítulo 7:

isalnum isalpha isdigit islower isupper tolower toupper

Otras funciones

isascii

Prototipo: int isascii (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un carácter perteneciente al conjunto de caracteres ASCII no extendido, de 0 a 127.
Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX.

iscntrl

Prototipo: int iscntrl (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un carácter de control, es decir, de código ASCII de 0 a 31 ó 127.
Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX y es compatible con ANSI C.

isgraph

Prototipo: int isgraph (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un carácter imprimible distinto del espacio (código ASCII de 33 a 126).

Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX y es compatible con ANSI C.

isprint

Prototipo: int isprint (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un carácter imprimible incluyendo el espacio (código ASCII de 32 a 126).
Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX y es compatible con ANSI C.

ispunct

Prototipo: int ispunct (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un carácter de puntuación, es decir, un carácter de control incluyendo el espacio.
Devuelve: 1 en caso afirmativo, 0 en caso contrario
Portabilidad: Está disponible para UNIX y es compatible con ANSI C.

isspace

Prototipo: int isspace (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un espacio, retorno de carro, tabulador vertical, tabulador horizontal, salto de página o salto de línea.
Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX y es compatible con ANSI C. Es compatible con K&R.

isxdigit

Prototipo: int isxdigit (int *carac*);
Include: ctype.h
Propósito: Informa acerca de si *carac* es un dígito hexadecimal (0-9, a-f, A-F).
Devuelve: 1 en caso afirmativo, 0 en caso contrario.
Portabilidad: Está disponible para UNIX y es compatible con ANSI C.

Funciones matemáticas

Aparte de las funciones que se muestran en este apartado, Turbo C proporciona un grupo importante de funciones trigonométricas, logarítmicas, exponenciales, hiperbólicas y complejas.

exp

Prototipo: double exp (double *x*);
Include: math.h
Propósito: Calcula e^x.
Devuelve: El valor calculado. Si este valor es demasiado grande para representarse como **double** devuelve HUGE_VAL.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

fabs

Prototipo: double fabs (double *x*);
Include: math.h
Propósito: Calcula el valor absoluto de *x*.
Devuelve: El valor absoluto de *x*.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

floor

Prototipo: double floor (double *x*);
Include: math.h
Propósito: Redondea *x* hacia abajo.
Devuelve: El mayor entero que no es mayor que *x*.
Portabilidad: Está disponible para UNIX y está definida en ANSI.C

modf

Prototipo: double modf (double *x*, double **entera*);
Include: math.h
Propósito: Descompone *x* en parte entera (que se almacena en *entera*) y parte fraccionaria.
Devuelve: La parte fraccionaria.
Portabilidad: Sólo DOS.

pow

Prototipo: double pow (double *x*, double *y*);
Include: math.h

Propósito: Calcula x^y.
Devuelve: Si tiene éxito, el valor calculado. Si hay desbordamiento devuelve HUGE_VAL y asigna a la variable global **errno** el valor ERANGE. Si *x* vale 0 e *y* es menor o igual a 0 se produce un error de dominio y asigna a **errno** el valor EDOM.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

sqrt

Prototipo: double sqrt (double *x*);
Include: math.h
Propósito: Calcula la raíz cuadrada positiva de *x*.
Devuelve: Si tiene éxito devuelve el valor calculado. Si *x* es negativo se produce un error de dominio y se asigna a la variable global **errno** el valor EDOM.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

Funciones de sistema

absread

Prototipo: int absread (int *unidad*, int *nsec*, int *desde*, void **buffer*);
Include: dos.h
Propósito: Lee *nsec* sectores de la unidad especificada con *unidad* (A=0, B=1, ...) a partir del sector *desde* y los almacena en *buffer*. Utiliza para la lectura la interrupción DOS 0x25. El número de sectores a leer está limitado a 64 Kb.
Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y asigna a la variable global **errno** el valor devuelto por el sistema en el registro AX.
Portabilidad: Sólo DOS.

abswrite

Prototipo: int abswrite (int *unidad*, int *nsec*, int *desde*, void **buffer*);
Include: dos.h
Propósito: Escribe en la unidad especificada con *unidad* (A=0, B=1, ...) *nsec* sectores a partir del sector *desde*. La información debe estar almacenada en *buffer*. Utiliza la interrupción DOS 0x26. El número de sectores a escribir está limitado a 64 Kb.

Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y asigna a la variable global **errno** el valor devuelto por el sistema en el registro AX.

Portabilidad: Sólo DOS.

biosequip

Prototipo: int biosequip (void);

Include: bios.h

Propósito: Usa la interrupción 0x11 para devolver un entero con información sobre el equipamiento del sistema.

Devuelve: Una palabra cuyos bits deben interpretarse según se explica a continuación:

- 14-15: Número de impresoras paralelo instaladas.
- 13: Impresora serie.
- 12: Adaptador de juegos.
- 9-11: Número de puertos COM.
- 8: Chip de acceso directo a memoria (DMA) instalado. Si vale 0 indica que sí.
- 6-7: Número de unidades de disquette, si el bit-0 es 1.
  - 00 = 1 unidad.
  - 01 = 2 unidades.
  - 10 = 3 unidades.
  - 11 = 4 unidades.
- 4-5: Modo de vídeo inicial:
  - 00 = No usado.
  - 01 = 40x25 BN con adaptador color.
  - 10 = 80x25 BN con adaptador color.
  - 11 = 80x25 BN con adaptador monocromo.
- 2-3: RAM en placa base:
  - 00 = 16 Kb.
  - 01 = 32 Kb.
  - 10 = 48 Kb.
  - 11 = 64 Kb.
- 1: Coprocesador matemático.
- 0: Arranque desde disquette.

Portabilidad: Sólo IBM-PC's y compatibles.

bioskey

Prototipo: int bioskey (int *op*);

Include: bios.h

Propósito: Realiza varias operaciones sobre el teclado usando la interrupción 0x16. La operación a realizar depende del valor de *op*.

Devuelve: El valor devuelto depende de *op*:

0: Espera a que se pulse una tecla y devuelve su valor en 2 bytes: el byte bajo contiene el cógigo ASCII. Si este es 0 (tecla especial), en el byte alto devuelve el código extendido.
1: Revisa si se pulsa una tecla. Devuelve 0 si no se ha pulsado ninguna, y diferente de 0 en caso contrario.
2: Devuelve un byte con el estado de alguna de las teclas, representando cada bit el estado de una tecla diferente:
   0: MAYÚSCULAS DCHA pulsada.
   1: MAYÚSCULAS IZQDA pulsada.
   2: Tecla CONTROL pulsada.
   3: Tecla ALT pulsada.
   4: Tecla BLOQ DESPL activada.
   5: Tecla BLOQ NUM activada.
   6: Tecla BLOQ MAYÚS activada.
   7: Tecla INSERT activada.

Portabilidad: Sólo IBM-PC's y compatibles.

biosmemory

Prototipo: int biosmemory (void);
Include: bios.h
Propósito: Obtiene la cantidad de RAM mediante la interrupción 0x12 del BIOS. No incluye memoria de vídeo, extendida ni expandida.
Devuelve: La cantidad de RAM en bloques de 1 Kb.
Portabilidad: Sólo IBM-PC's y compatibles.

biosprint

Prototipo: int biosprint (int *op*, int *byte*, int *puerto*);
Include: bios.h
Propósito: Realiza varias operaciones sobre el puerto paralelo de la impresora usando la interrupción 0x17 del BIOS. El valor de *puerto* es 0 para LPT1, 1 para LPT2, etc. Las operaciones realizadas dependen del valor de *op*:
   0: Envía a la impresora el byte *byte*.
   1: Inicializa el puerto.
   2: Obtiene el estado de la impresora.

Devuelve: Un byte de estado de la impresora. Cada bit debe interpretarse como sigue:
   0: Error de time-out.
   1-2: Sin usar.
   3: Error de E/S.
   4: Impresora conctada.
   5: No hay papel.

6: Reconocimiento (ACK).
7: No ocupada.

Portabilidad: Sólo IBM-PC's y compatibles.

biostime

Prototipo: long biostime (int *op*, long *nuevot*);
Include: bios.h
Propósito: Lee o asigna un valor al reloj del sistema. Este reloj se inicializa a medianoche y se incrementa a razón de 18.2 ticks por segundo. Si *op* vale 0 lee el reloj; si vale 1, asigna al sistema el valor *nuevot*.
Devuelve: Si *op* vale 0 devuelve el valor actual del reloj del sistema.
Portabilidad: Sólo IBM-PC's y compatibles.

clock

Prototipo: clock_t clock (void);
Include: time.h
Propósito: Determina el tiempo de proceso. Puede usarse para calcular el tiempo transcurrido entre dos sucesos. Para determinar el tiempo en segundos, el valor devuelto por **clock** debe dividirse por la macro CLK_TCK.
Devuelve: El tiempo de proceso desde el comienzo del programa. Si hay error, devuelve -1.
Portabilidad: Es compatible con ANSI C.

delay

Prototipo: void delay (unsigned *ms*);
Include: dos.h
Propósito: Detiene la ejecución del programa durante *ms* milisegundos.
Devuelve:
Portabilidad: Sólo IBM-PC's y compatibles.

FP_OFF

Prototipo: unsigned FP_OFF (void far **p*);
Include: dos.h
Propósito: Es una macro que obtiene el valor de desplazamiento del puntero *p*.
Devuelve: El valor del desplazamiento.
Portabilidad: Sólo DOS.

FP_SEG

Prototipo: unsigned FP_SEG (void far ***p*);
Include: dos.h
Propósito: Es una macro que obtiene el valor de segmento del puntero *p*.
Devuelve: El valor de segmento.
Portabilidad: Sólo DOS.

getdate

Prototipo: void getdate (struct date ***fecha*);
Include: dos.h
Propósito: Obtiene la fecha del sistema mediante una estructura **date** apuntada por *fecha*. La estructura **date** esta definida como sigue:

```
struct date {
    int da_year;      //año
    int da_day;       //día
    int da_mon;       //mes
};
```

Devuelve:
Portabilidad: Sólo DOS.

gettime

Prototipo: void gettime (struct time ***hora*);
Include: dos.h
Propósito: Obtiene la hora del sistema mediante una estructura **time** apuntada por *hora*. La estructura **time** está definida como sigue:

```
struct time {
    unsigned char ti_min;      //minutos
    unsigned char ti_hour;     //horas
    unsigned char ti_hund;     //centésimas de seg.
    unsigned char ti_sec;      //segundos
};
```

Devuelve:
Portabilidad: Sólo DOS.

int86

Prototipo: int int86 (int *nint*, union REGS ***regin*, union REGS ***regout*);
Include: dos.h

Propósito: Ejecuta la interrupción especificada por *nint*. Antes de llamar a la función deben ponerse en la estructura apuntada por *regin* los valores adecuados en los registros del procesador. La estructura REGS está definida como sigue:

```
struct WORDREGS {
   unsigned int ax, bx, cx, dx, si, di, cflag;
};

struct BYTEREGS {
   unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};

union REGS {
   struct WORDREGS x;
   struct BYTEREGS h;
};
```

Después de ejecutada la función, la estructura apuntada por *regout* contiene los valores de los registros del procesador devueltos.

Devuelve: El valor dejado en AX por la rutina de interrupción. Si el indicador de acarreo (representado por **x.cflag**) está activado indicando un error, la función asigna a la variable global **_doserrno** el código de error.

Portabilidad: Sólo para la familia de procesadores 8086.

int86x

Prototipo: int int86x (int *nint*, union REGS **regin*, union REGS **regout*, struct SREGS **regseg*);

Include: dos.h

Propósito: Es idéntica a la anterior, pero utiliza la estructura **SREGS** que está definida como sigue:

```
struct SREGS {
   unsigned int es;
   unsigned int cs;
   unsigned int ss;
   unsigned int ds;
};
```

Esta función copia antes de ejecutar la interrupción, los valores de **ds** y **es** en los correspondientes registros. Esto permite a los programas usar punteros lejanos o el modelo grande de memoria para especificar qué segmento de memoria debe usarse en la interrupción.

Devuelve: El valor dejado en AX por la rutina de interrupción. Si el indicador de acarreo (representado por **x.cflag**) está activado indicando un error, la función asigna a la variable global **_doserrno** el código de error.

Portabilidad: Sólo para la familia de procesadores 8086.

intdos

Prototipo: int intdos (union REGS **regin*, union REGS **regout*);
Include: dos.h
Propósito: Es idéntica a **int86** pero sólo para la interrupción 0x21.
Devuelve: El valor dejado en AX por la rutina de interrupción. Si hay error se activa el indicador de acarreo **x.cflag**, y se asigna a la variable global **_doserrno** el código de error. La estructura **REGS** se muestra en la página 222 (**int86**).
Portabilidad: Sólo para la familia de procesadores 8086.

intdosx

Prototipo: int intdosx (union REGS **regin*, union REGS **regout*, struct SREGS **regseg*);
Include: dos.h
Propósito: Es idéntica a **int86x** pero sólo para la interrupción 0x21.
Devuelve: El valor dejado en AX por la interrupción. Si hay error se activa el indicador de acarreo **x.cflag**, y se asigna a la variable global **_doserrno** el código de error. Las estructuras **REGS** y **SREGS** se muestran en la página 222 (**int86** e **int86x**).
Portabilidad: Sólo para la familia de procesadores 8086.

peek

Prototipo: int peek (unsigned *segmento*, unsigned *despl*);
Include: dos.h
Propósito: Lee la palabra situada en la dirección *segmento*:*despl*.
Devuelve: El valor leído.
Portabilidad: Sólo para la familia de procesadores 8086.

peekb

Prototipo: char peekb (unsigned *segmento*, unsigned *despl*);
Include: dos.h
Propósito: Lee el byte situado en la dirección *segmento*:*despl*.
Devuelve: El valor leído.
Portabilidad: Sólo para la familia de procesadores 8086.

poke

Prototipo: void poke (unsigned *segmento*, unsigned *despl*, int *valor*);
Include: dos.h
Propósito: Almacena el entero *valor* en la dirección *segmento*:*despl*.

Devuelve:
Portabilidad: Sólo para la familia de procesadores 8086.

pokeb

Prototipo: void pokeb (unsigned *segmento*, unsigned *despl*, char *valor*);
Include: dos.h
Propósito: Almacena el byte *valor* en la dirección *segmento*:*despl*.
Devuelve:
Portabilidad: Sólo para la familia de procesadores 8086.

segread

Prototipo: void segread (struct SREGS **regseg*);
Include: dos.h
Propósito: Obtiene los valores de los registros de segmento. La estructura **SREGS** está definida en la página 221 (**int86x**).
Devuelve:
Portabilidad: Sólo para la familia de procesadores 8086.

setdate

Prototipo: void setdate (struct date **fecha*);
Include: dos.h
Propósito: Establece la fecha del sistema. La estructura **date** está definida en la página 221 (**getdate**).
Devuelve:
Portabilidad: Sólo DOS.

settime

Prototipo: void settime (struct time **hora*);
Include: dos.h
Propósito: Establece la hora del sistema. La estructura **time** está definida en la página 221 (**gettime**).
Devuelve:
Portabilidad: Sólo DOS.

sleep

Prototipo: void sleep (unsigned *seg*);
Include: dos.h
Propósito: Suspende la ejecución del programa durante *seg* segundos.

Devuelve:
Portabilidad: Esta disponible para UNIX.

Funciones de asignación dinámica de memoria

Ya se han estudiado en el Capítulo 9 las funciones:

free **malloc**

Otras funciones se explican a continuación.

calloc

Prototipo: void *calloc (size_t *n*, size_t *tam*);
Include: stdlib.h, alloc.h
Propósito: Asigna un bloque de *n* elementos de *tam* bytes del motón, inicializándolo a 0.
Devuelve: Si tiene éxito, un puntero al bloque asignado. En caso contrario devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX y está definida en ANSI C. Es compatible con K&R.

coreleft

Prototipo: unsigned coreleft (void);[1]
unsined long coreleft (void);[2]
Include: alloc.h
Propósito: Obtiene una medida de la cantidad de memoria libre en el montón.
Devuelve: El número de bytes libres del montón.
Portabilidad: Sólo DOS.

realloc

Prototipo: void *realloc (void **p*, size_t *tam*);
Include: stdlib.h, alloc.h
Propósito: Cambia el tamaño de memoria asignada apuntada por *p* al nuevo tamaño dado por *tam*.

[1] Para los modelos de memoria diminuto (tiny), pequeño (small) y mediano (medium), explicados en el Capítulo 12.
[2] Para los modelos de memoria compacto (compact), grande (large) y enorme (huge), explicados en el Capítulo 12.

Devuelve: Un puntero al bloque asignado. Este puede ser diferente del anterior. Si no tiene éxito, devuelve un puntero nulo.
Portabilidad: Está disponible para UNIX y está definido en ANSI C.

Funciones de directorio

chdir

Prototipo: int chdir (const char ***nuevodir*);
Include: dir.h
Propósito: Establece como directorio actual *nuevodir*.
Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y asigna a la variable global **errno** el valor ENOENT.
Portabilidad: Está disponible para UNIX.

findfirst

Prototipo: int findfirst (const char ***path*, struct ffblk ***bloque*, int *atrib*);
Include: dir.h, dos.h
Propósito: Busca el primer nombre de archivo que coincida con *path*. El nombre de archivo puede contener comodines * y ?. Si el archivo se encuentra, llena la información de la estructura **ffblk** apuntada por *bloque*. Esta estructura está definida como sigue:

```
struct ffblk {
   char ff_reserved[21];     //Reservado por DOS
   char ff_attrib;           //Atributos de archivo
   int ff_ftime;             //Hora (Campo de bits)
   int ff_date;              //Fecha (Campo de bits)
   long ff_fsize;            //Tamaño
   char ff_name[13];               //Nombre
};
```

En el parámetro *atrib* se selecciona el tipo de archivo que se buscará. Puede tener alguno de los siguientes valores:

FA_RDONLY	Archivo de sólo-lectura
FA_HIDDEN	Archivo oculto
FA_SYSTEM	Archivo de sistema
FA_LABEL	Etiqueta de volumen
FA_DIREC	Directorio
FA_ARCH	Archivo

Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y asigna a la variable global **errno** alguno de los siguientes valores:

ENOENT Archivo no encontrado
ENMFILE No más ficheros

Portabilidad: Sólo DOS

findnext

Prototipo: int findnext (struct ffblk *_bloque_);
Include: dir.h
Propósito: Continúa la búsqueda iniciada por **findfirst**.
Devuelve: Los mismos valores que **findfirst**.
Portabilidad: Sólo DOS.

getcurdir

Prototipo: int getcurdir (int *unidad*, char *_directorio_);
Include: dir.h
Propósito: Pone en la cadena *directorio* el directorio actual de trabajo para el disco *unidad* (0: defecto, 1: A, 2: B, ...).La cadena *directorio* debe tener como máximo MAXDIR caracteres. La macro MAXDIR está definida en **dir.h**.
Devuelve: Si tiene éxito devuelve 0. En caso contrario, -1.
Portabilidad: Sólo DOS.

getcwd

Prototipo: char *getcwd (char *_buffer_, int *longbuf*);
Include: dir.h
Propósito: Copia el directorio actual de trabajo y lo sitúa en la cadena *buffer*. Si el nombre del directorio tiene más de *longbuf* caracteres, se produce un error. Si *buffer* es nulo, la función asigna memoria para *buffer* con una llamada a **malloc**. En ese caso, puede liberarse la memoria asignada mediante **free**.
Devuelve: Un puntero a *buffer* si tiene éxito. En caso contrario devuelve un puntero nulo. Si ocurre un error, asigna a la variable global **errno** uno de los valores siguientes:
ENODEV No existe el dispositivo
ENOMEM No hay suficiente memoria
ERANGE Resultado fuera de rango

Portabilidad: Sólo DOS

getdisk

Prototipo: int getdisk (void);
Include: dir.h

Propósito: Obtiene el número de la unidad actual.
Devuelve: 0 para A:, 1 para B:, etc.
Portabilidad: Sólo DOS.

mkdir

Prototipo: int mkdir (const char *_directorio_);
Include: dir.h
Propósito: Crea el directorio indicado en la cadena *directorio*.
Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1, y asigna a la variable global **errno** alguno de los valores siguientes:

EACCES	Acceso denegado
ENOENT	Encaminamiento no válido

Portabilidad:

mktemp

Prototipo: char *mktemp (char *_nombre_);
Include: dir.h
Propósito: Crea un nombre de archivo único y lo pone en *nombre*. La cadena *nombre* debe estar inicializada como "XXXXXX". Esta función no abre el fichero.
Devuelve: Si tiene éxito, un puntero a nombre. En caso contrario, un puntero nulo.
Portabilidad: Está disponible para sistemas UNIX.

rmdir

Prototipo: int rmdir (const char *_directorio_);
Include: dir.h
Propósito: Borra el directorio cuyo nombre se indica en *directorio*. Para poder borrar el directorio debe cumplirse que:

- Está vacío
- No es el directorio de trabajo
- No es el directorio raíz

Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1 y asigna a la variable global **errno** alguno de los valores siguientes:

EACCES	Acceso denegado
ENOENT	Encaminamiento no válido

Portabilidad: Está disponible para UNIX y está definida en ANSI C.

searchpath

Prototipo: char *searchpath (const char **nombre*);
Include: dir.h
Propósito: Busca el archivo *nombre* usando la variable de entorno PATH.
Devuelve: Si tiene éxito devuelve un puntero al encaminamiento completo. En caso contrario, devuelve un puntero nulo.
Portabilidad: Sólo DOS.

setdisk

Prototipo: int setdisk (int *unidad*);
Include: dir.h
Propósito: Establece como unidad activa la especificada mediante *unidad* (0=A, 1=B, ...).
Devuelve: El número de unidades disponible.
Portabilidad: Sólo DOS.

Funciones de control de procesos

abort

Prototipo: void abort (void);
Include: stdlib.h, process.h
Propósito: Muestra el mensaje *Abnormal program termination* en **stderr** y finaliza el programa, enviando al proceso padre un código de retorno 3. No cierra ningún archivo.
Devuelve:
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

Funciones **exec**

Prototipos:
int execl (char *_nom_, char *_arg0_, ..., char *_argN_, NULL);
int execle (char *_nom_, char *_arg0_, ..., char *_argN_, NULL, char *_entorno[ ]_);
int execlp (char *_nom_, char *_arg0_, ..., char *_argN_, NULL);
int execlpe (char *_nom_, char *_arg0_, ..., char *_argN_, NULL, char *_entorno[ ]_);
int execv (char *_nom_, char *_arg[ ]_);
int execve (char *_nom_, char *_arg[ ]_, char *_entorno[ ]_);
int execvp (char *_nom_, char *_arg[ ]_);
int execvpe (char *_nom_, char *_arg[ ]_, char *_entorno[ ]_);

Include: process.h

Propósito: Las funciones **exec** ejecutan otros programas, llamados *procesos hijo*. Después de la llamada el proceso hijo sustituye al *proceso padre* en memoria, siempre que haya suficiente. El nombre del proceso hijo se especifica con el parámetro *nom*. Al proceso hijo se le pasan los argumentos individualmente mediante *arg0*, ..., *argN*, o bien en el vector *arg[ ]*. También se puede pasar una matriz de cadenas de entorno mediante el parámetro *entorno[ ]*.

El proceso hijo es buscado siguiendo las siguientes normas:

- Si *nom* no tiene incluida una extensión, se buscará un archivo con ese nombre. Si no se encuentra, se buscará el mismo archivo con extensión .COM. Si tampoco se encuentra, se buscará un archivo con extensión .EXE.
- Si *nom* lleva incluida una extensión, sólo se buscará ese archivo.

La manera en que se ejecutan estas funciones dependen de qué versión de función **exec** se utilice. Las versiones se diferencian por los sufijos **p**, **l**, **v** y **e** que se pueden añadir al nombre. El significado de estos sufijos es el siguiente:

p Si la función tiene este sufijo, el proceso hijo se busca usando la variable de entorno PATH del DOS. En caso contrario la búsqueda se realiza sólo en el directorio de trabajo.

l Los argumentos se pasan individualmente mediante los parámetros *arg0*, ..., *argN*. El último argumento debe ser un **NULL** (definido en stdio.h).

v Los argumentos se pasarán en una matriz *arg[ ]*.

e Se pasará una o varias cadenas de entorno mediante la matriz *entorno[ ]*. El último elemento de la matriz debe ser un **NULL**.

Las funciones **exec** deben pasar al proceso hijo al menos un argumento. Este argumento es, por convenio, *nom*, aunque cualquier otro no produce error.

Cuando se usa el sufijo **l**, *arg0* apunta a *nom*, y *arg1*, ..., *argN* apuntan a la lista de argumentos que se pasan.

Cuando se usa el sufijo **e**, Las cadenas de la matriz *entorno[ ]* deben tener la forma

variable de entorno = valor

La suma de longitudes de los argumentos que se pasan no debe superar los 128 bytes (sin contar los nulos).

Las funciones **exec** no cierran ficheros.

Devuelve: En caso de error devuelve -1 y asigna a la variable global **errno** alguno de los valores siguientes:

E2BIG	Demasiados argumentos
EACCES	Acceso denegado al proceso hijo
EMFILE	Demasiados ficheros abiertos
ENOENT	Archivo no encontrado
ENOEXEC	Error de formato en la función **exec**
ENOMEM	Memoria insuficiente

Portabilidad: Sólo DOS.

exit

Prototipo: void exit (int *st*);

Include: process.h, stdlib.h

Propósito: Provoca la terminación del programa. Cierra todos los ficheros abiertos y descarga todos los buffers de salida. Envía al proceso padre un código de retorno *st*. Normalmente se envía un valor 0 para la terminación normal, y un valor diferente de 0 si se ha producido un error.

Devuelve:

Portabilidad: Está disponible para UNIX y está definida en ANSI C.

Funciones **spawn**

Prototipos:

```
int spawnl (int modo, char *nom, char *arg0, ..., char *argN, NULL);
int spawnle (int modo, char *nom, char *arg0, ..., char *argN, NULL, char *entorno[ ]);
int spawnlp (int modo, char *nom, char *arg0, ..., char *argN, NULL);
int spawnlpe (int modo, char *nom, char *arg0, ..., char *argN, NULL, char *entorno[ ]);
int spawnv (int modo, char *nom, char *arg[ ]);
int spawnve (int modo, char *nom, char *arg[ ], char *entorno[ ]);
int spawnvp (int modo, char *nom, char *arg[ ]);
int spawnvpe (int modo, char *nom, char *arg[ ], char *entorno[ ]);
```

Include: process.h

Propósito: Estas funciones son idénticas a las funciones **exec**, salvo en tres detalles:

En primer lugar, en las funciones **spawn** el proceso hijo no sustituye necesariamente al proceso padre en memoria.

En segundo lugar, se añade un nuevo parámetro, *modo*, que puede tener los siguientes valores:

P_WAIT	Deja suspendida la ejecución del proceso padre hasta que termine el proceso hijo.
P_NOWAIT	Ejecuta concurrentemente el padre y el hijo. No está disponible en Turbo C.
P_OVERLAY	El proceso hijo sustituye al proceso padre en memoria.

La tercera diferencia está en cómo el proceso hijo es buscado. Se siguen las siguientes normas:

- Si *nom* no tiene incluida una extensión ni un punto, se buscará un archivo con ese nombre. Si no se encuentra, se buscará el mismo archivo con extensión .COM. Si tampoco se encuentra, se buscará un archivo con extensión .EXE.
- Si *nom* lleva incluida una extensión, sólo se buscará ese archivo.
- Si *nom* tiene un punto se buscará sólo un fichero con ese nombre y sin extensión.

Devuelve: En caso de error devuelve -1 y asigna a la variable global **errno** alguno de los valores siguientes:

E2BIG	Demasiados argumentos
EMFILE	Demasiados ficheros abiertos
ENOENT	Archivo no encontrado
ENOEXEC	Error de formato en la función **exec**
ENOMEM	Memoria insuficiente

Portabilidad: Sólo DOS.

Funciones de pantalla de texto

Ya se han estudiado las siguientes funciones en el Capítulo 4:

clreol	**clrscr**	**delline**	**gotoxy**	**highvideo**	**insline**
lowvideo	**movetxt**	**normvideo**	**textattr**	**textbackground**	**textcolor**
textmode	**wherex**	**wherey**			

Otras funciones de pantalla de texto se muestran a continuación:

cprintf

Prototipo: int cprintf (const char **format* [, *argumento*, ...]);
Include: conio.h
Propósito: Es prácticamente idéntica a **printf**. Las diferencias son:

- Escribe en la ventana activa y no en **stdout**.
- Impide que se superen los límites de la ventana.
- No se puede redireccionar la salida.
- Respeta los atributos de color actual.
- No convierte la nueva línea (**\n**) en salto de línea + retorno de carro. Es preciso añadir **\r**.

Devuelve: El número de caracteres escritos.
Portabilidad: Sólo IBM-PC's y compatibles.

cputs

Prototipo: int cputs (const char *_cadena_);
Include: conio.h
Propósito: Es prácticamente idéntica a **puts**. Las diferencias entre **puts** y **cputs** son las mismas que entre **printf** y **cprintf**.
Devuelve: Si tiene éxito devuelve el último carácter escrito. En caso contrario, devuelve EOF.
Portabilidad: Sólo IBM-PC's y compatibles.

cscanf

Prototipo: int cscanf (const char *_format_ [, _direcc_, ...]);
Include: conio.h
Propósito: Es idéntica a **scanf** salvo que lee la información de la consola en lugar de **stdin**. Respeta los atributos de color actual. No se puede redirigir.
Devuelve: El número de argumentos a los que realmente se les ha asignado valores.
Portabilidad: Sólo DOS.

gettext

Prototipo: int gettext (int *izq*, int *arriba*, int *dcha*, int *abajo*, void *_dest_);
Include: conio.h
Propósito: Almacena en *dest* un rectángulo de la pantalla de texto definido por las coordenadas *izq*, *arriba*, *dcha* y *abajo*. Las coordenadas son absolutas, no relativas a la ventana. La esquina superior izquierda está en (1,1). La copia se hace secuencialmente, es decir, de izquierda a derecha y de arriba hacia abajo. Cada posición en la pantalla ocupa 2 bytes en la memoria: el primero almacena el carácter y el segundo el atributo.
Devuelve: Si tiene éxito, devuelve 1. En caso contrario, devuelve 0.
Portabilidad: Sólo en IBM-PC's y en sistemas con BIOS compatible.

puttext

Prototipo: int puttext (int *izq*, int *arriba*, int *dcha*, int *abajo*, void *_fuente_);
Include: conio.h
Propósito: Escribe el contenido del área apuntada por *fuente* en el rectángulo de pantalla definido por las coordenadas *izq*, *arriba*, *dcha* y *abajo*. Generalmente el área *fuente* se ha llenado con una llamada a **gettext**.
Devuelve: Si tiene éxito, devuelve un valor diferente de 0. En caso contrario, devuelve 0.
Portabilidad: Sólo en IBM-PC's y en sistemas con BIOS compatible.

window

Prototipo: void window (int *izq*, int *arriba*, int *dcha*, int *abajo*);
Include: conio.h
Propósito: Define la ventana de texto activa. Las coordenadas *izq*, *arriba* son las de la esquina superior izquierda. Las coordenadas *dcha*, *abajo* son las de la esquina inferior derecha.
Devuelve:
Portabilidad: Sólo en IBM-PC's y compatibles.

Otras funciones

abs

Prototipo: int abs (int *x*);
Include: math.h, stdlib.h
Propósito: Obtiene el valor absoluto del entero *x*.
Devuelve: El valor calculado.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

atof

Prototipo: double atof (const char ***cad*);
Include: math.h, stdlib.h
Propósito: Convierte la cadena apuntada por *cad* en un **double**. Esta función reconoce la representación de un número en coma flotante con la siguiente estructura:

[blancos][signo][ddd][.][e|E[signo]ddd]

Cualquier carácter no admisible corta la entrada.
Devuelve: El valor calculado. Si se produce desbordamiento devuelve +/- HUGE_VAL y asigna a la variable global **errno** el valor ERANGE.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

atoi

Prototipo: int atoi (const char *_cad_);
Include: stdlib.h
Propósito: Convierte la cadena *cad* en un número entero. Esta función reconoce el formato:

[blancos][signo][ddd]

La entrada finaliza si se encuentra un carácter no válido.
Devuelve: El valor convertido. Si la conversión no es posible devuelve 0.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

atol

Prototipo: long atol (const char *_cad_);
Include: stdlib.h
Propósito: Convierte la cadena *cad* en un número **long**. Esta función reconoce el formato

[blancos][signo][ddd]

La entrada finaliza si se encuentra un carácter no válido.
Devuelve: El valor convertido. Si la conversión no es posible devuelve 0.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

itoa

Prototipo: char *itoa (int *valor*, char **cad*, int *base*);
Include: stdlib.h
Propósito: Convierte el entero *valor* en una cadena, y lo almacena en *cad*. El parámetro *base* especifica en qué base se realizará la conversión. Debe estar entre 2 y 36. Si *valor* es negativo y *base* es 10, el primer carácter de *cad* es el signo menos (-). El espacio asignado para *cad* debe ser suficientemente grande para almacenar el resultado. La función **itoa** puede devolver hasta 17 caracteres.
Devuelve: Un puntero a *cad*.
Portabilidad: Sólo DOS.

kbhit

Prototipo: int kbhit (void);
Include: conio.h
Propósito: Detecta si una tecla ha sido pulsada, sin retirar el carácter del buffer y sin detener el programa.

Devuelve: Si se ha pulsado una tecla devuelve un valor diferente de 0. En caso contrario devuelve 0.
Portabilidad: Sólo DOS.

labs

Prototipo: long int labs (long int *x*);
Include: math.h, stdlib.h
Propósito: Calcula el valor absoluto de *x*.
Devuelve: El valor calculado.
Portabilidad: Está disponible para UNIX y está definida en ANSI C.

ltoa

Prototipo: char *ltoa (long *valor*, char **cad*, int *base*);
Include: stdlib.h
Propósito: Convierte el entero largo *valor* en una cadena, y lo almacena en *cad*. El parámetro *base* especifica en qué base se realizará la conversión. Debe estar entre 2 y 36. Si *valor* es negativo y *base* es 10, el primer carácter de *cad* es el signo menos (-). El espacio asignado para *cad* debe ser suficientemente grande para almacenar el resultado. La función **ltoa** puede devolver hasta 33 caracteres.
Devuelve: Un puntero a *cad*.
Portabilidad: Sólo DOS.

nosound

Prototipo: void nosound (void);
Include: dos.h
Propósito: Apaga el altavoz después de una llamada a **sound**.
Devuelve:
Portabilidad: Sólo IBM-PC's y compatibles.

random

Prototipo: int random (int *num*);
Include: stdlib.h
Propósito: Genera un número aleatorio entero entre 0 y *num* - 1.
Devuelve: El valor generado.
Portabilidad:

randomize

Prototipo: void randomize (void);
Include: stdlib.h, time.h
Propósito: Inicializa el generador de números aleatorios.
Devuelve:
Portabilidad:

_setcursortype

Prototipo: void _setcursortype (int *tipocursor*);
Include: conio.h
Propósito: Establece el aspecto del cursor. Los valores posibles para *tipocursor* son:

_NOCURSOR	Desaparece el cursor
_SOLIDCURSOR	Cursor de bloque (▉)
_NORMALCURSOR	Cursor normal (_)

Devuelve:
Portabilidad: Sólo IBM-PC's y compatibles.

sound

Prototipo: void sound (unsigned *ciclos*);
Include: dos.h
Propósito: Activa el altavoz a una frecuencia de sonido de *ciclos* hertz (ciclos por segundo).
Devuelve:
Portabilidad: Sólo IBM-PC's y compatibles.

system

Prototipo: int system (const char **orden*);
Include: stdlib.h, process.h
Propósito: Llama al intérprete de comandos (COMMAND.COM) del DOS para ejecutar la orden especificada mediante *orden*. Esta orden puede ser una orden DOS, un fichero .BAT o cualquier programa.
Devuelve: Si tiene éxito devuelve 0. En caso contrario devuelve -1.
Portabilidad: Está disponible para UNIX y está definida en ANSI C. Es compatible con K&R.

ultoa

Prototipo: char *ultoa (unsigned long *valor*, char **cad*, int *base*);
Include: stdlib.h
Propósito: Convierte el **unsigned long** *valor* en una cadena, y lo almacena en *cad*. El parámetro *base* especifica en qué base se realizará la conversión. Debe estar entre 2 y 36. Si *valor* es negativo y *base* es 10, el primer carácter de *cad* es el signo menos (-). El espacio asignado para *cad* debe ser suficientemente grande para almacenar el resultado. La función **ultoa** puede devolver hasta 33 caracteres.
Devuelve: Un puntero a *cad*.
Portabilidad: Sólo DOS.