```
 i = 0;
delta = 'a' - 'A';
while (cad3[i] != 'n0')
 {
     if ((cad3[i] >= 'a') && (cad3[i] <= 'z'))
       cad3[i] -= delta; /* Convierte a mayuscula */
     else if ((cad3[i] >= 'A') && (cad3[i] <= 'Z'))
       cad3[i] += delta; /* Convierte a minúscula */
     i++;
 }
 printf( "La cadena resultante es: %s \n", cad3 );

}
```

Otros tipos de datos

En este capítulo veremos los restantes mecanismos que C ofrece para la creación y manejo de tipo de datos complejos. Concretamente las *estructuras* y las *uniones*. Por otra parte, se incluye también un apartado que estudia los tipos de datos *enumerados* y otro donde se trata la definición de nuevos tipos de datos por parte del programador.

Estructuras

En el capítulo 7 hemos estudiado el tipo abstracto de datos *tabla*, formado por un conjunto de elementos todos ellos del mismo tipo de datos. En una *estructura*, sin embargo, los elementos que la componen pueden ser de distintos tipos. Asi pues, una estructura puede agrupar datos de tipo carácter, enteros, cadenas de caracteres, matrices de números . . . , e incluso otras estructuras. En cualquier caso, es habitual que todos los elementos agrupados en una estructura tengan alguna relación entre ellos. En adelante nos referiremos a los elementos que componen una estructura como *campos*.

La definición de una estructura requiere especificar un nombre y el tipo de datos de todos los campos que la componen, así como un nombre mediante el cual pueda identificarse toda la agrupación. Para ello se utiliza la palabra reservada struct de la forma siguiente:

```
struct nombre_estructura
{
   tipo_campo_1     nombre_campo_1;
   tipo_campo_2     nombre_campo_2;
   .  .  .
   tipo_campo_N      nombre_campo_N;
};
```

El siguiente ejemplo define una estructura denominada cliente en la que puede almacenarse la ficha bancaria de una persona. El significado de los diferentes campos es obvio:

```
struct cliente
{
   char nombre[100];
   long int dni;

   char domicilio[200];

   long int no cuenta;
   float saldo;
}
```

Puede decirse que la definición de una estructura corresponde a la definición de una "`plantilla" genérica que se utilizara posteriormente para declarar variables. Por tanto la definición de una estructura no representa la reserva de ningún espacio de memoria.

Declaración de variables

La declaración de variables del tipo definido por una estructura puede hacerse de dos maneras diferentes. Bien en la misma definición de la estructura, bien posteriormente. La forma genérica para el primer caso es la siguiente:

```
struct nombre estructura
{
   tipo_campo_1  nombre_campo_1;
   tipo_campo_2  nombre_campo_2;
   .  .  .
   tipo_campo_N  nombre_campo_N;
} lista de variable;
```

Nótese que al declararse las variables al mismo tiempo que se define la estructura, el nombre de esta junto a la palabra reservada struct se hace innecesario y puede omitirse.

Por otra parte, suponiendo que la estructura nombre estructura se haya definido previamente, la declaración de variables en el segundo caso sería:

```
struct nombre_estructura lista_de_variables;
```

Siguiendo con el ejemplo anterior, según la primera variante de declaración, podríamos escribir:

```
struct
{
   char nombre[100];
   long int dni;
   char domicilio[200];
```

```
    long int no_cuenta;
    float saldo;
} antiguo_cliente, nuevo_cliente;
```

O bien, de acuerdo con la segunda variante donde se asume que la estructura cliente se ha definido previamente:

```
struct cliente antiguo cliente, nuevo cliente;
```

Acceso a los campos

Los campos de una estructura de manejan habitualmente de forma individual. El mecanismo que C proporciona para ello es el operador "punto" (.). La forma general para acceder a un campo en una variable de tipo estructura es el siguiente:

```
variable.nombre_campo
```

Así pues, podríamos acceder a los campos de la variable nuevo cliente como el nombre, número de dni o el saldo de la siguiente forma:

```
nuevo_cliente.nombre nuevo_cliente.dni nuevo_cliente.saldo
```

Asignación

La asignación de valores a los campos de una variable estructura puede realizarse de dos formas diferentes. Por un lado, accediendo campo a campo, y por otro, asignando valores para todos los campos en el momento de la declaración de la variable.

A continuación se muestran algunas posibles maneras de asignar datos a los campos individuales de la variable nuevo_cliente declarada previamente. Como puede verse, cada campo es tratado como si de una variable se tratase, con lo que pueden usarse funciones como strcpy para copiar una cadena de caracteres sobre un campo de ese tipo, o gets para leerla del teclado, etc.

```
strcpy( nuevo_cliente.nombre, "Federico Sancho Buch" );
nuevo_cliente.dni = 23347098;
gets( nuevo_cliente.domicilio );
scanf( "%ld",&nuevo_cliente.no cuenta );
nuevo_cliente.saldo += 10000.0;
```

Por otra parte, el siguiente ejemplo muestra la inicialización completa de todos los campos de una variable de tipo estructura en el momento de su declaración:

```
struct cliente nuevo cliente = {
                                "Federico Sancho Buch",
                                23347098,
                                "Rue del Percebe 13 - Madrid",
```

```
7897894,
1023459.34
};
```

Finalmente, también es posible copiar todos los datos de una variable estructura a otra variable estructura del mismo tipo mediante una simple asignación:

```
nuevo_cliente = antiguo_cliente;
```

No está permitido en ningún caso, comparar dos estructuras utilizando los operadores relacionales que vimos en la sección 3.4.3.

Ejemplo

El siguiente programa define las estructuras de datos para gestionar un conjunto de fichas personales. Nótese el uso de estructuras anidadas, así como el uso de tablas de estructuras. Nótese también el uso de la función gets para leer cadenas de caracteres. Se deja como ejercicio para el lector escribir este mismo programa usando scanf en lugar de gets para leer dichas cadenas.

```
#include <stdio.h>
struct datos
{
   char nombre[20];
   char apellido[20];
   long int dni;
};
struct tablapersonas
{
   int numpersonas;
   struct datos persona[100];
};
void main()

{
   int i;
   struct tablapersonas tabla;

   printf( "Numero de personas: " );
   scanf( "%d", &tabla.numpersonas );
   for (i= 0; i< tabla.numpersonas; i++)
   {
      printf( "\nNombre: " );
      gets( tabla.persona[i].nombre );
      printf( "\nApellido: " );
      gets( tabla.persona[i].apellido );
```

```
        printf( "\nDNI: " );
        scanf( "%ld", &tabla.persona[i].dni );
    }
}
```

Uniones

Al igual que las estructuras, las *uniones* también pueden contener múltiples campos de diferentes tipos de datos. Sin embargo, mientras que cada campo en una estructura posee su propia área de almacenamiento, en una unión, todos los campos que la componen se hallan almacenados en la misma área de memoria. El espacio de memoria reservado por el compilador corresponde al del campo de la unión que requiere mayor espacio de almacenamiento. El resto de campos comparten dicho espacio.
Así pues, los campos de una unión están solapados en memoria, por lo que, en un momento dado de la ejecución del programa, solo podría haber almacenado un dato en uno de los campos. Es responsabilidad del programador hacer que el programa mantenga control sobre qué campo contiene la información almacenada en la unión en cada momento. Intentar acceder al tipo de información equivocado puede producir resultados sin sentido.

La definición de una unión es muy similar a la de una estructura, excepto por el uso de la palabra reservada union:

```
union nombre_union
{
   tipo_campo_1  nombre_campo_1;
   tipo_campo_2  nombre_campo_2;
   .  .  .
   tipo_campo_N nombre_campo_N;
};
```

Tanto la declaración de variables de tipo unión como el acceso a los campos se expresa de forma similar a como se mostro en las estructuras.

Finalmente, diremos que una estructura puede ser el campo de una unión y viceversa. Igualmente, pueden definirse tablas de uniones, etc.

Ejemplo

El siguiente ejemplo define tres estructuras para almacenar la información asociada a tres tipos diferentes de maquinas voladoras (jet, helicoptero y carguero). Finalmente define una estructura genérica que puede contener, alternativamente, los datos de cualquiera de ellos (un aeroplano). Para controlar de que tipo es el objeto almacenado en la unión datos, se utiliza la variable tipo.

```
struct jet
{
   int num_pasajeros;
   .  .  .
```

```
};
struct helicoptero
{
   int capacidad_elevador;
   . . .
};
struct carguero
{
    int carga_maxima;

    . . .
};
union aeroplano
{
    struct jet jet_u;
    struct helicoptero helicopter_u;
    struct carguero carguero_u;
};

struct un aeroplano
{
   /* 0:jet, 1:helicoptero, 2:carguero */
   int tipo;
   union aeroplano datos;
};
```

Tipos de datos enumerados

Un objeto enumerado consiste en un conjunto ordenado de constantes enteras cuyos nombres indican todos los posibles valores que se le pueden asignar a dicho objeto.

La definición de una "plantilla" de un objeto enumerado requiere especificar un nombre mediante el cual pueda identificarse, así como la lista de constantes de los posibles valores que puede tomar. Para ello se utiliza la palabra reservada enum de la forma siguiente:

```
enum nombre_enumeracion { constante_1,
                          constante_2,
                          . . .
                          constante_N;
                               };
```

El siguiente ejemplo define una enumeración denominada día semana, cuyo significado es obvio.

```
enum dia_semana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO, DOMINGO
                };
```

Las constantes que definen los posibles valores de la enumeración son representadas internamente como constantes enteras. El compilador asigna a cada una de ellas un valor en función del orden (empezando por 0) en que aparecen en la definición. Así pues, en el ejemplo anterior tenemos que LUNES es 0, MARTES es 1, etc. Sin embargo, podría ser de interés modificar dicha asignación por defecto, asignando a las constantes otro valor numérico. Ver el siguiente ejemplo:

```
enum dia_semana { LUNES=2, MARTES=3, MIERCOLES=4,
                  JUEVES=5, VIERNES=6,
                  SABADO=7, DOMINGO=1
                };
```

También puede asignarse el mismo valor numérico a diferentes constantes, así como dejar que el compilador numere algunas por defecto. Ver el siguiente ejemplo:

```
enum dia_semana { LUNES=1, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO, DOMINGO=10, FESTIVO=10
                  };
```

donde los valores asociados a las constantes son, respectivamente: 1, 2, 3, 4, 5, 6, 10 y 10.

La declaración de variables puede hacerse de dos formas diferentes. Bien en la misma definición de la enumeración, bien posteriormente. Por su similitud con la declaración de variables en el caso de estructuras y uniones, veremos simplemente un ejemplo del segundo caso. El mismo código muestra algunos ejemplos de utilización de las constantes definidas en un tipo enumerado.

```
enum dia_semana{LUNES=1, MARTES, MIERCOLES, JUEVES,  VIERNES,
                SABADO, DOMINGO=10, FESTIVO=10
                };

void main()
{
    enum dia semana dia;

    . . .
    if (dia <= VIERNES)
        printf( "Es laborable" );
    . . .
    dia = MARTES;
    /* Muestra el valor de dia */
```

```
  printf( "Hoy es: %d", dia );
  . . .
}
```

Definición de nuevos tipos de datos

C ofrece al programador la posibilidad de definir nuevos tipos de datos mediante la palabra reservada typedef. Los nuevos tipos definidos pueden utilizarse de la misma forma que los tipos de datos predefinidos por el lenguaje. Es importante destacar que typedef tiene especial utilidad en la definición de nuevos tipos de datos estructurados, basados en tablas, estructuras, uniones o enumeraciones.

La sintaxis general para definir un nuevo tipo de datos es la siguiente:

```
typedef tipo_de_datos nombre_nuevo_tipo;
```

De esta forma se ha definido un nuevo tipo de datos de nombre nombre nuevo tipo cuya estructura interna viene dada por tipo_de_datos.

A continuación se muestran algunos ejemplos de definición de nuevos tipos de datos, así como su utilización en un programa:

```
typedef float Testatura;

typedef char Tstring [30];

typedef enum Tsexo = { HOMBRE, MUJER };
typedef struct
{
   Tstring nombre, apellido;
   Testatura alt;
   Tsexo sex;
}  Tpersona;

void main()
{
  Tpersona paciente;

   . . .
   scanf( "%f", &paciente.alt );
   gets( paciente.nombre );



printf( "%s", paciente.apellido );
   paciente.sex = MUJER;
}
```

Tiras de bits

C es un lenguaje muy versátil que permite programar con un alto nivel de abstracción. Sin embarco, C también permite programar a muy bajo nivel. Esta característica es especialmente útil, por ejemplo, para programas que manipulan dispositivos *hardware*, como el programa que controla el funcionamiento de un *modem*, etc. Este tipo de programas manipulan tiras de bits.

En C, una tira de bits se debe almacenar como una variable entera con o sin signo. Es decir, como una variable de tipo char, short, int, long, unsigned, etc. Seguidamente veremos las operaciones que C ofrece para la manipulación tiras de bits.

Operador de negación

Este operador también recibe el nombre de operador de *complemento a 1*, y se representa mediante el símbolo ~. La función de este operador consiste en cambiar los 1 por 0 y viceversa. Por ejemplo, el siguiente programa:

```
#include <stdio.h>
void main()
{
   unsigned short a;

   a= 0x5b3c;     /* a = 0101 1011 0011 1100 */
   b= ~a;         /*  b = 1010 0100 1100 0011 */
   printf( " a= %x     b= %x\n", a, b );
   printf( " a= %u     b= %u\n", a, b );
   printf( " a= %d     b= %d\n", a, b );
}
```

mostraría en pantalla los siguientes mensajes:

```
a= 0x5b3c  b= 0xa4c3
a= 23356   b= 42179
a= 23356   b= -23357
```

como resultado de intercambiar por sus complementarios los bits de la variable a.

x	**y**	**AND** (&)	**OR** (|)	**XOR** (^)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabla 8.1: Tabla de verdad de los operadores lógicos

Operadores lógicos

C permite realizar las operaciones lógicas *AND* (&), *OR* (|) y *XOR* (^), también llamadas respectivamente "Y", "O" y "O exclusivo". La tabla 8.1 muestra la tabla de verdad de estas operaciones, donde x e y son bits.

Cuando aplicamos estos operadores a dos variables la operación lógica se realiza bit a bit. Veámoslo en el siguiente ejemplo:

```
a= 0x5b3c;          /* a = 1101 1011 0001 1101 */
b= 0xa4c3;          /* b = 1010 0101 1100 1011 */
c= a & b;           /* c = 1000 0001 0000 1001 */
d= a | b;           /* d = 1111 1111 1101 1111 */
e= a ^ b;           /* e = 0111 1110 1101 0110 */
```

Operadores de desplazamiento de bits

Existen dos operadores de desplazamiento que permiten desplazar a derecha o izquierda una tira de bits. El operador de desplazamiento a la derecha se denota como >> y el operador de desplazamiento a la izquierda se denota como <<. Su uso es como sigue:

```
var1 << var2
var1 >> var2
```

donde var1 es la tira de bits desplazada y var2 es el numero de bits desplazados. En el desplazamiento a la izquierda se pierden los bits de más a la izquierda de var1, y se introducen ceros por la derecha. De forma similar, en el desplazamiento a la derecha se pierden los bits de mas a la derecha de var1, y por la izquierda se introducen, o bien ceros (si la variable es de tipo unsigned, o bien se repite el ´ultimo bit (si la variable es de tipo signed).

El siguiente ejemplo muestra el resultado de aplicar los operadores de desplazamiento de bits:

```
unsigned short a, d, e;
short b, c, f, g;

a= 28;              /* a = 0000 0000 0001 1100 */
b= -28;             /* b = 1111 1111 1110 0100 */
c= 3;               /* c = 0000 0000 0000 0011 */
d= a << c;          /* d = 0000 0000 1110 0000 = 224 */
e= a >> c;          /* e = 0000 0000 0000 0011 = 3 */
f= b << c;          /* f = 1111 1111 0010 0000 = -224 */
g= b >> c;          /* g = 1111 1111 1111 1100 = -3 */
```

Es importante señalar que los operadores de desplazamiento tienen un significado aritmético en base 10. El desplazamiento a la derecha de 1 bit es equivalente a dividir por 2, obteniéndose el cociente de la división entera. El desplazamiento a la

izquierda de 1 bit es equivalente a multiplicar por 2. Por lo tanto, cuando trabajemos con variables enteras:
var << n equivale a var * 2^n , y var >> n equivale a var / 2^n.

Punteros

Un puntero es una variable que contiene como valor una dirección de memoria. Dicha dirección corresponde habitualmente a la dirección que ocupa otra variable en memoria. Se dice entonces que el puntero *apunta a dicha variable*. La variable apuntada puede ser de cualquier tipo elemental, estructurado o incluso otro puntero.

Los punteros constituyen una parte fundamental del lenguaje C y son básicos para comprender toda la potencia y flexibilidad que ofrece el lenguaje. Son especialmente importantes en la programación a bajo nivel, donde se manipula directamente la memoria del ordenador. Algunas de las ventajas que aporta el uso de punteros en C son:

- Constituyen la única forma de expresar algunas operaciones.
- Su uso produce código compacto y eficiente.
- S
  on imprescindibles para el paso de parámetros por referencia a funciones.
- Tienen una fuerte relación con el manejo eficiente de tablas y estructuras.
- Permiten realizar operaciones de asignación dinámica de memoria y manipular estructuras de datos dinámicas.

Finalmente, cabe advertir al lector que los punteros son tradicionalmente la parte de C mas difícil de comprender. Además deben usarse con gran precaución, puesto que al permitir manipular directamente la memoria del ordenador, pueden provocar fallos en el programa.
Estos fallos suelen ser bastante difíciles de localizar y de solucionar.

Declaración y asignación de direcciones

En la declaración de punteros y la posterior asignación de direcciones de memoria a los mismos, se utilizan respectivamente los operadores unarios * y &. El operador & permite obtener la *dirección que ocupa una variable en memoria*. Por su parte, el operador de *indireccion* * permite obtener el *contenido de un objeto apuntado por un puntero*.

Declaración

En la declaración de variables puntero se usa también el operador *, que se aplica directamente a la variable a la cual precede. El formato para la declaración de variables puntero es el siguiente:

```
tipo_de_datos * nombre_variable_puntero;
```

Nótese que un puntero debe estar asociado a un tipo de datos determinado. Es decir, no puede asignarse la dirección de un short int a un puntero a long int. Por ejemplo:

```
int *ip;
```

declara una variable de nombre ip que es un puntero a un objeto de tipo int. Es decir, ip contendrá direcciones de memoria donde se almacenaran valores enteros.

No debe cometerse el error de declarar varios punteros utilizando un solo *. Por ejemplo:

```
int *ip, x;
```

declara la variable ip como un puntero a entero y la variable x como un entero (no un puntero a un entero).

El tipo de datos utilizado en la declaración de un puntero debe ser del mismo tipo de datos que las posibles variables a las que dicho puntero puede apuntar. Si el tipo de datos es void, se define un puntero genérico de forma que su tipo de datos implícito será el de la variable cuya dirección se le asigne. Por ejemplo, en el siguiente código, ip es un puntero genérico que a lo largo del programa apunta a objetos de tipos distintos, primero a un entero y posteriormente a un carácter.

```
void *ip;
int x;
char car;
.  .  .
ip = &x;     /* ip apunta a un entero */
ip = &car;   /* ip apunta a un carácter */
```

Al igual que cualquier variable, al declarar un puntero, su valor no está definido, pues es el correspondiente al contenido aleatorio de la memoria en el momento de la declaración. Para evitar el uso indebido de un puntero cuando aun no se le ha asignado una dirección, es conveniente inicializarlo con el valor nulo NULL, definido en el fichero de la librería estándar stdio.h. El siguiente ejemplo muestra dicha inicialización:

```
int *ip = NULL;
```

De esta forma, si se intentase acceder al valor apuntado por el puntero ip antes de asignarle una dirección válida, se produciría un error de ejecución que interrumpiría el programa.

Asignación de direcciones

El operador & permite obtener la dirección que ocupa una variable en memoria. Para que un puntero apunte a una variable es necesario asignar la dirección de dicha variable al puntero. El tipo de datos de puntero debe coincidir con el tipo de datos de la variable apuntada (excepto en el caso de un puntero de tipo void). Para visualizar la dirección de memoria contenida en un puntero, puede usarse el modificador de formato %p con la función printf:

```
double num;
double *pnum = NULL;
.   .   .
pnum = &num;
printf( "La dirección contenida en 'pnum' es: %p", pnum );
```

Indireccion

Llamaremos *indireccion* a la forma de referenciar el valor de una variable a través de un puntero que apunta a dicha variable. En general usaremos el termino indireccion para referirnos al hecho de referenciar el valor contenido en la posición de memoria apuntada por un puntero. La indireccion se realiza mediante el operador *, que precediendo al nombre de un puntero indica el valor de la variable cuya dirección está contenida en dicho puntero. A continuación se muestra un ejemplo del uso de la indireccion:

```
int x, y;
int *p;         /* Se usa * para declarar un puntero */
.   .   .
x = 20;
p = &x;
*p = 5498;   /* Se usa * para indicar el valor de la
                variable apuntada */
y = *p;       /* Se usa * para indicar el valor de la
                 variable apuntada */
```

Después de ejecutar la sentencia *p = 5498; la variable x, apuntada por p, toma por valor 5498.
Finalmente, después de ejecutar la sentencia y = *p; la variable y toma por valor el de la variable x, apuntada por p.

Para concluir, existe también la indireccion múltiple, en que un puntero contiene la dirección de otro puntero que a su vez apunta a una variable. El formato de la declaración es el siguiente:

```
tipo_de_datos ** nombre_variable_puntero;
```

En el siguiente ejemplo, pnum apunta a num, mientras que ppnum apunta a pnum. Asi pues, la sentencia **ppnum = 24; asigna 24 a la variable num.

```
int num;
int *pnum;
int **ppnum;
.  .  .
pnum = &num;
ppnum = &pnum;
**ppnum = 24;
printf( "%d", num );
```

La indireccion múltiple puede extenderse a mas de dos niveles, aunque no es recomendable por la dificultad que supone en la legibilidad del código resultante.

La utilización de punteros debe hacerse con gran precaución, puesto que permiten manipular directamente la memoria. Este hecho puede provocar fallos inesperados en el programa, que normalmente son difíciles de localizar. Un error frecuente consiste en no asignar una dirección valida a un puntero antes de usarlo. Veamos el siguiente ejemplo:

```
int *x;
.  .  .
*x = 100;
```

El puntero x no ha sido inicializado por el programador, por lo tanto contiene una dirección de memoria aleatoria, con lo que es posible escribir involuntariamente en zonas de memoria que contengan código o datos del programa. Este tipo de error no provoca ningún error de compilación, por lo que puede ser difícil de detectar. Para corregirlo es necesario que x apunte a una posición controlada de memoria, por ejemplo:

```
int y;
int *x;
.  .  .
x = &y;
*x = 100;
```

Operaciones con punteros

En C pueden manipularse punteros mediante diversas operaciones como asignación, comparación y operaciones aritméticas.

Asignación de punteros

Es posible asignar un puntero a otro puntero, siempre y cuando ambos apunten a un mismo tipo de datos. Después de una asignación de punteros, ambos apuntan a la misma variable, pues contienen la misma dirección de memoria. En el siguiente ejemplo, el puntero `p2` toma por valor la dirección de memoria contenida en `p1`. Así pues, tanto y como `z` toman el mismo valor, que corresponde al valor de la variable x, apuntada tanto por `p1` como por `p2`.

```
int x, y, z;
int *p1, *p2;
.  .  .
x = 4;
p1 = &x;
p2 = p1;
y = *p1;
z = *p2;
```

Comparación de punteros

Normalmente se utiliza la comparación de punteros para conocer las posiciones relativas que ocupan en memoria las variables apuntadas por los punteros. Por ejemplo, dadas las siguientes declaraciones,

```
int *p1, *p2, precio, cantidad;
*p1 = &precio;
*p2 = &cantidad;
```

la comparación `p1 > p2` permite saber cuál de las dos variables (precio o cantidad) ocupa una posición de memoria mayor. Es importante no confundir esta comparación con la de los valores de las variables apuntadas, es decir, `*p1 > *p2`.

Aritmética de punteros

Si se suma o resta un número entero a un puntero, lo que se produce implícitamente es un incremento o decremento de la dirección de memoria contenida por dicho puntero. El número de posiciones de memorias incrementadas o decrementadas depende, no sólo del número sumado o restado, sino también del tamaño del tipo de datos apuntado por el puntero. Es decir, una sentencia como:

```
nombre_puntero = nombre_puntero + N;
```

se interpreta internamente como:

```
nombre_puntero = dirección + N * tamaño_tipo_de_datos;
```

Por ejemplo, teniendo en cuenta que el tamaño de un float es de 4 bytes, y que la variable num se sitúa en la dirección de memoria 2088, ¿cuál es el valor de pnum al final del siguiente código?

```
float num, *punt, *pnum;
.  .  .
punt = &num;
pnum = punt + 3;
```

La respuesta es 2100. Es decir, 2088 + 3 * 4.

Es importante advertir que aunque C permite utilizar aritmética de punteros, esto constituye una práctica *no recomendable*. Las expresiones aritméticas que manejan punteros son difíciles de entender y generan confusión, por ello son una fuente inagotable de errores en los programas. Como veremos en la siguiente sección, no es necesario usar expresiones aritméticas con punteros, pues C proporciona una notación alternativa mucho más clara.

Punteros y tablas

En el lenguaje C existe una fuerte relación entre los punteros y las estructuras de datos de tipo tabla(vectores, matrices, etc.).

En C, el nombre de una tabla se trata internamente como un puntero que contiene la dirección del primer elemento de dicha tabla. De hecho, el nombre de la tabla es una constante de tipo puntero, por lo que el compilador no permitirá que las instrucciones del programa modifiquen la dirección contenida en dicho nombre. Así pues, dada una declaración como char tab[15] , el empleo de tab es equivalente al empleo de &tab[0]. Por otro lado, la operación tab = tab + 1 generara un error de compilación, pues representa un intento de modificar la dirección del primer elemento de la tabla.

C permite el uso de punteros que contengan direcciones de los elementos de una tabla para acceder a ellos. En el siguiente ejemplo, se usa el puntero ptab para asignar a car el tercer elemento de tab, leer de teclado el quinto elemento de tab y escribir por pantalla el decimo elemento del vector tab.

```
char car;
char tab[15];
char *ptab;
.  .  .
ptab = &tab;
ptab = ptab + 3;
car = *(ptab);        /* Equivale a car = tab[3]; */
scanf( "%c", ptab+5 );
printf( "%c", ptab+10 );
```

Pero la relación entre punteros y tablas en C va aún mías allá. Una vez declarado un puntero que apunta a los elementos de una tabla, pueden usarse los corchetes para indexar dichos elementos, como si de una tabla se tratase. Así, siguiendo con el ejemplo anterior, sería correcto escribir:

```
scanf( "%c", ptab[0] );/* ptab[0] equivale a tab[0] */
ptab[7] = 'R'; /* ptab[7] equivale a *(ptab +7) */
```

Por lo tanto, vemos que no es necesario usar expresiones aritméticas con punteros; en su lugar usaremos la sintaxis de acceso a una tabla. Es importante subrayar que este tipo de indexaciones sólo son válidas si el puntero utilizado apunta a los elementos de una tabla. Además, no existe ningún tipo de verificación al respecto, por lo que es responsabilidad del programador saber en todo momento si está accediendo a una posición de memoria dentro de una tabla o ha ido a parar fuera de ella y está sobrescribiendo otras posiciones de memoria.

El siguiente ejemplo muestra el uso de los punteros a tablas para determinar cuál de entre dos vectores de enteros es mías *fuerte*. La *fuerza* de un vector se calcula como la suma de todos sus elementos.

```
#include <stdio.h>
#define DIM 10
void main()
{
    int v1[DIM], v2[DIM];
    int i, fuerza1, fuerza2;
    int *fuerte;

    /* Lectura de los vectores. */
   for (i= 0; i< DIM; i++)
       scanf( "%d ", &v1[i] );

for (i= 0; i< DIM; i++)
    scanf( "%d ", &v2[i] );

/* C´alculo de la fuerza de los vectores. */
fuerza1 = 0;
fuerza2 = 0;
for (i= 0; i< DIM; i++)
{
    fuerza1 = fuerza1 + v1[i];
    fuerza2 = fuerza2 + v2[i];
}
if (fuerza1 > fuerza2)
   fuerte = v1;
else
   fuerte = v2;

/* Escritura del vector más fuerte. */
for (i= 0; i< DIM; i++)
    printf( "%d ", fuerte[i] );
```

```
}
```

En el caso de usar punteros para manipular tablas multidimensionales, es necesario usar formulas de transformación para el acceso a los elementos. Por ejemplo, en el caso de una matriz de n filas y m columnas, el elemento que ocupa la fila i y la columna j se referencia por medio de un puntero como puntero[i*m+j]. En el siguiente ejemplo se muestra el acceso a una matriz mediante un puntero.

```
float mat[3][5];
float *pt;

pt = mat;
pt[i*5+j] = 4.6; /* Equivale a mat[i][j]=4.6 */
```

Cuando usamos el puntero para acceder a la matriz, la expresión pt[k] significa acceder al elemento de tipo float que está k elementos por debajo del elemento apuntado por pt . Dado que en C las matrices se almacenan por filas, para acceder al elemento de la fila i columna j deberemos contar cuantos elementos hay entre el primer elemento de la matriz y el elemento [i][j]. Como la numeración comienza en cero, antes de la fila i hay exactamente i filas, y cada una tiene m columnas. Por lo tanto hasta el primer elemento de la fila i tenemos i x m elementos. Dentro de la fila i, por delante del elemento j, hay j elementos. Por lo tanto tenemos que entre mat[0][0] y mat[i][j] hay i x m x j elementos.

La figura 9.1 muestra como está dispuesta en memoria la matriz de este ejemplo y una explicación grafica del cálculo descrito.

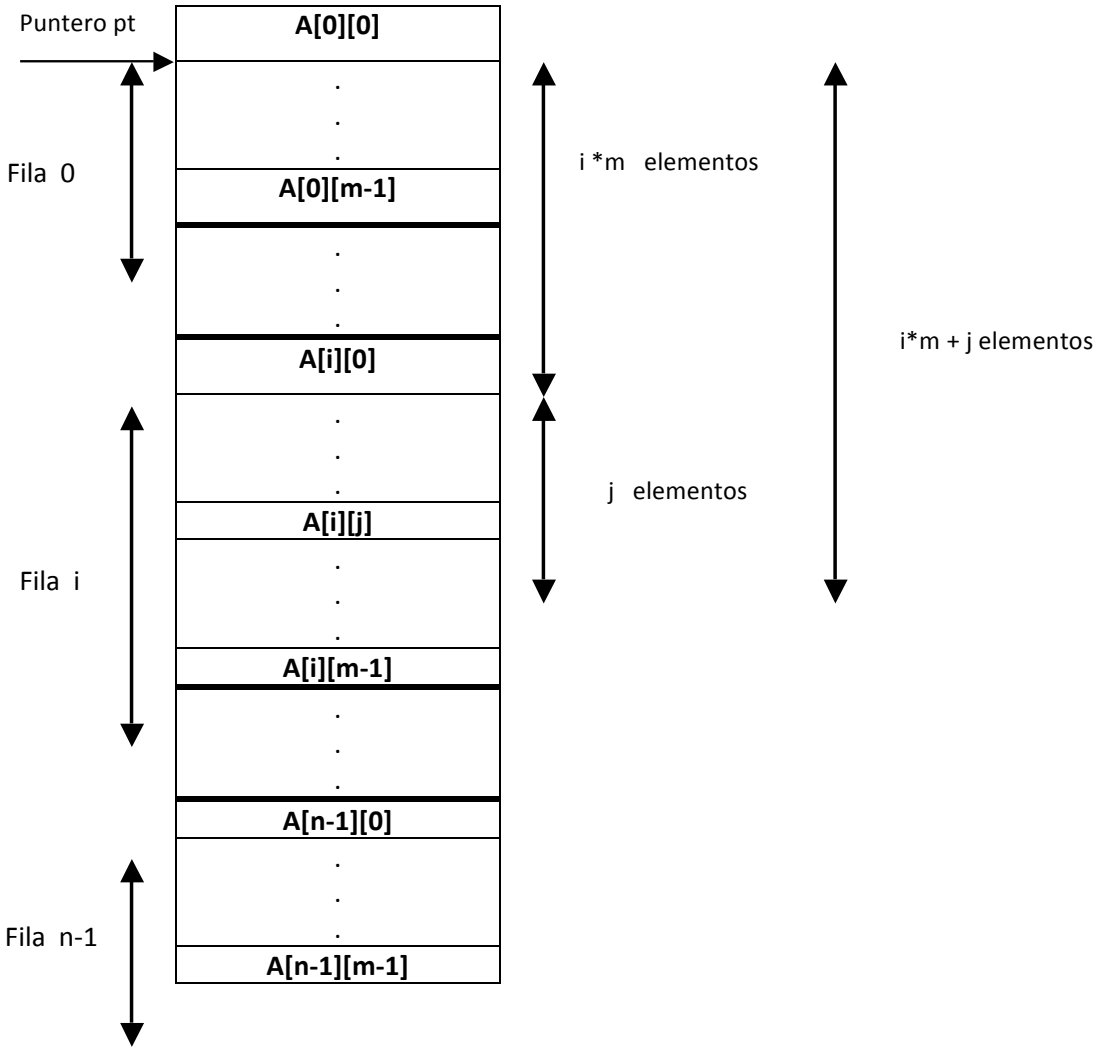

Figura 9.1: Acceso a una matriz mediante un puntero

Punteros y cadenas de caracteres

Una cadena de caracteres puede declararse e inicializarse sin necesidad de especificar explícitamente su longitud. La siguiente declaración es un ejemplo:

```
char mensaje[ ] = "Reiniciando sistema";
```

La cadena de caracteres mensaje ocupa en memoria 20 bytes (19 más el carácter nulo '\0'). El nombre mensaje corresponde a la dirección del primer carácter de la cadena. En esta declaración, mensaje es un puntero constante, por lo que no puede modificarse para que apunte a otro carácter distinto del primer carácter de la cadena. Sin embargo, puede usarse mensaje para acceder a caracteres individuales dentro de la cadena, mediante sentencias como mensaje[13]='S', etc.

La misma declaración puede hacerse usando un puntero:

```
char *pmens = "Reiniciando sistema";
```

En esta declaración, pmens es una variable puntero inicializada para apuntar a una cadena constante, que como tal no puede modificarse. Sin embargo, el puntero pmens puede volver a utilizarse para apuntar a otra cadena.

Puede decirse que en general a una tabla puede accederse tanto con índices como con punteros (usando la aritmética de punteros). Habitualmente es más cómodo el uso de índices, sin embargo en el paso de parámetros a una función (ver Cap. 10) donde se recibe la dirección de una tabla, es necesario utilizar punteros. Como veremos, dichos punteros podrán usarse como tales o usando la indexación típica de las tablas.

Punteros y estructuras

Los punteros a estructuras funcionan de forma similar a los punteros a variables de tipos elementales, con la salvedad de que en las estructuras se emplea un operador especifico para el acceso a los campos.

Puede accederse a la dirección de comienzo de una variable estructura en memoria mediante el empleo del operador de dirección &. Así pues, si var es una variable estructura, entonces &var representa la dirección de comienzo de dicha variable en memoria. Del mismo modo, puede declararse una variable puntero a una estructura como:

```
tipo_estructura *pvar;
```

donde tipo_estructura es el tipo de datos de las estructuras a las que pvar puede apuntar. Así pues, puede asignarse la dirección de comienzo de una variable estructura al puntero de la forma habitual:

```
pvar = &var;
```

Veamos un ejemplo. A continuaci´on se define, entre otros, el tipo de datos Tnif como:

```
typedef char Tstring [50];

typedef struct
{
   long int num;
   char letra;
} Tnif;

typedef struct
{
   Tnif nif;
```