3. COMPUTADORAS Y LOGICAS

El **sistema binario,** en matematicas e informática, es un sistema de numeración en el que los números se presentan utilizando solamente las cifras cero y uno (*0* y *1*). Es el que se utiliza en los ordenadores, pues trabajan internamente con dos niveles de voltaje, por lo que su sistema de numeración natural es el sistema binario (encendido *1,* apagado *0)*

REPRESENTACION

Un numero binario puede ser representado por cualquier secuencia de bits (digitos binario), que a su vez pueden ser presentados por cualquier mecanismo capaz de estar en dos estados mutuamente exclusivos. Las secuencias siguientes de simbolos podrían ser interpretadas todas como el mismo valor binario numérico:

```
0 1 0 1 0 0 1 1 0 1 0
I - I - - I I - I -
X o x o o x x o x o
y n y n n y y n y n
```

símbolo. En un ordenador, los valores numéricos pueden ser representados por dos voltajes diferentes y también se pueden usar polaridades magnéticas sobre un disco magnetico. Un "positivo", "si", o "sobre el estado" no es necesariamente el equivalente al valor numérico de uno; esto depende de la arquitectura usada.

De acuerdo con la representación acostumbrada de cifras que usan números árabes, los números binarios comúnmente son escritos usando los símbolos 0 y 1. Cuando son escritos, los números binarios son a menudo subindicados, prefijados o sufijados para indicar su base, o la raíz. Las notaciones siguientes son equivalentes:

- 100101 binario (declaración explícita de formato)
- 100101b (un sufijo que indica formato binario)
- 100101B (un sufijo que indica formato binario)
- bin 100101 (un prefijo que indica formato binario)
- 100101_2 (un subíndice que indica base 2 (binaria) notación)
- % 100101 (un prefijo que indica formato binario)
- 0b100101 (un prefijo que indica formato binario, común en lenguajes de programación)

CONVERSIÓN ENTRE BINARIO Y DECIMAL

Decimal a binario

Se divide el numero del sistema decimal entre 2, cuyo resultado entero se vuelve a dividir entre 2, y asi sucesivamente. Ordenados los restos, del ultimo al primero, este será el numero binario que buscamos.

Ejempl o

Transformar el numero decimal 131 en binario. El método es muy simple:

```
131 dividido por 2 da 65 y el resto es igual a 1
 65 dividido por 2 da 32 y el resto es igual a 1
 32 dividido por 2 da 16 y el resto es igual a 0
 16 dividido por 2 da  8 y el resto es igual a 0
  8 dividido por 2 da  4 y el resto es igual a 0
  4 dividido por 2 da  2 y el resto es igual a 0
  2 dividido por 2 da  1 y el resto es igual a 0
  1 dividido por 2 da  0 y el resto es igual a 1
```
Ordenamos los restos, del último al primero: 1000001

Decimal (con decimales) a binario

Para transformar un numero del sistema decimal en sistema binario:

1. Se inicia por el lado izquierdo, multiplicando cada numero por 2 (si la parte entera es mayor que 0 en binario será 1, y en caso contrario es 0)
2. En caso de ser 1, en la siguiente división se utilizan solo los decimales.
3. Después de realizar cada multiplicación, se coloca los números obtenidos en el orden de su obtención.
4. Algunos números se transforman en digitos periódicos, por ejemplo: 0,1

Ejemplo

```
0,1 (decimal) => 0,0 0011 0011 . . . (binario).
Procesos:
0,1 x 2 = 0,2 => 0
0,2 x 2 = 0,4 => 0
0,4 x 2 = 0,8 => 0
0,8 x 2 = 1,6 => 1
0,6 x 2 = 1,2 => 1
0,2 x 2 = 0,4 => 0  <- se repiten las cuatro cifras, periódicamente
0,4 x 2 = 0,8 => 0  <-
0,8 x 2 = 1,6 => 1  <-
0,6 x 2 = 1,2 => 1  <- . . .
En orden: 0 0011 0011 . . .
```

Binario a decimal

Para realizar la conversión de binario a decimal, realice lo siguien:

1. Inicie por el lado derecho del numero binario, cada numero multiplíquelo por 2 y elévelo a la potencia consecutiva (comenzando por la potencia 0).
2. Después de realizar cada una de las multiplicaciones, sume todas y el numero resultante será el equivalente al sistema decimal.

Ejemplos:

(Los números de arriba indican la potencia a la que hay que elevar 2)

5 4 3 2 1 0

$110101_2 = 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 32+16+0+4+0+1 = 53$

7 6 5 4 3 2 1 0

$10010111 = 1.2^7 + 0.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 128+0+$

$0+16+0+4+2 = 150.$

5 4 3 2 1 0
$110111_2 = 1.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 32+16+0+4+2+1 = 55$

También se puede optar por utilizar los valores que se presenta cada posición del numero binario a ser transformado, comenzando de derecha a izquierda, y sumando los valores de las posiciones que tiene un 1.

Ejemplo

El numero binario 1010010 corresponde en decimal al 82 se puede representar de la siguiente manera:

64 32 16 8 4 2 1
1 0 1 0 0 1 0_2

Entonces se suma los números 64, 16 y 2:

64 32 16 8 4 2 1
1 0 1 0 0 1 0_2 = 64+16+2 = 82.

Binario a decimal (con decimal binario)

1. Inicie por el lado izquierdo, cada numero multiplíquenlo por 2 y elévelo a la potencia consecutiva a la inversa (comenzando por la potencia -1).

2. Después de realizar cada una de las multiplicaciones, sume todas y el numero resultante será el equivalente al sistema decimal.

Ejemplo

- 0.101001 (binario) = 0.640625 (decimal). Proceso
-

```
1 * (2) elevado a (-1) =0.5
0 * (2) elevado a (-2) =0
1 * (2) elevado a (-3) =0.125
0 * (2) elevado a (-4) =0
0 * (2) elevado a (-5) =0
1 * (2) elevado a (-6) =0.015625
La suma es:  0.640625
```

Operaciones con numero binarios

Las posibles combinaciones al sumar dos bits son:

- 0+ 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10 al sumar 1+1 siempre nos llevamos 1 a la siguiente operación.

Ejemplo

```
   10011000
+  00010101
   10101101
```

El algoritmo de la resta en sistema binario es el mismo que en el sistema decimal. Pero conviene repasar la operación de restar en decimal para comprender la operación binaria, que es mas sencilla. Los términos que intervienen en la resta se llaman minuendo, sustraendo y diferencia.

La resta básica 0 – 0, 1 – 0 y 1 – 1 son evidentes:

- 0 – 0 = 0
- 1 – 0 = 1
- 1 – 1 = 0
- 0 – 1 = **1**(se transforma en 10–1 =1) (en sistema decimal equivalente a 2 – 1 = 1)

La resta 0 – 1 se resuelve, igual que en el sistema decimal, tomando una unidad prestada de la posición siguiente:

0 – 1 = **1** y me llevo 1, lo que equivale a decir en el sistema decimal, 2 – 1 = 1.

Ejemplo

```
 1 0 0 0 1              1 1 0 1 1 0 0 1
-0 1 0 1 0             -1 0 1 0 1 0 1 1
----------             ----------------
 0 0 1 1 1               0 0 1 0 1 1 1 0
```

En sistema decimal seria: 17 – 10 = 7 y 217 – 171 = 46.

Para simplificar las restas y reducir la posibilidad de cometer errores hay varios métodos:

- Dividir los números largos en grupos. En el siguiente ejemplo, vemos como se divide una resta larga en tres restas cortas:

```
 100110011101          1001    1001    1101
-010101110010    =    -0101   -0111   -0010
 010000101011          0100    0010    1011
```

- Utilizando el complemento a dos (C2). La resta de dos números binarios pueden obtenerse sumando al minuendo el <<complemento a dos>> del sustraendo.

Ejemplo

La siguiente resta, 91 – 46 = 45, en binario es:

```
  1011011                                    1011011
 -0101110      el C2 de 0101110 es 1010010  + 1010010
  0101101                                    10101101
```

Producto de números binarios

El logaritmo del producto en binario es igual que en número decimales; aunque se lleva cabo con mas sencillez, ya que el 0 multiplicando por cualquier numero da 0, y el 1es el elemento neutro del producto.

Por ejemplo, multipliquemos 10110 por 1001:

```
    10110
     1001

    10110
   00000
  00000
 10110
 11000110
```

División de números binarios

La división en binario es similar a la decimal, la única diferencia es que a la hora de hacer las restas, dentro de la división, estas deben ser realizadas en binario.

Ejemplo

Dividir 100010010 (274) entre 1101 (13):

```
 100010010 |1101
 -  0000     010101
 10001
 -  1101
   01000
  -  0000
     10000
   -   1101
        00111
      -  0000
         01110
        -  1101
         00001
```

CONVERSION ENTRE BINARIO Y OCTAL

Binario a octal

Para realizar la conversión de binario a octal, realice lo siguiente:

1) Agrupe la cantidad binaria en grupos de 3 en 3 iniciando por el lado derecho. Si al terminar de agrupar no completa 3 dígitos, entonces agregue ceros a la izquierda.

2) Posteriormente vea el valor que corresponde de acuerdo a la tabla:

Numero en binario	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Numero en octal	0	1	2	3	4	5	6	7

3)La cantidad corresponde en octal se agrupa de izquierda a derecha.

Ejemplos

- 110111 (binario) = 67 (octal). Proceso:

```
111 = 7
110 = 6
Agrupe de izquierda a derecha: 67
```

- 11001111 (binario) = 317 (octal). Procesos:

```
111 = 7
001 = 1
11  entonces agregue un cero, con lo que se obtiene 011 = 3
Agrupe de izquierda a derecha: 317
```

Octal a binario

Cada digito octal se lo convierte en su binario equivalente de 3 bits y se juntan en el mismo orden.

Ejemplo

- 247 (octal) = 010100111 (binario). El 2 es binario es 10, pero en binario de 3 bits es Oc(2) = B(010); el Oc(4) = B(100) y el Oc(7) = (111), luego el numero en binario será 010100111

CONVERSION ENTRE BINARIO Y HEXADECIMAL

Binario a hexadecimal

Para realizar la conversión de binario a hexadecimal, realice lo siguiente:

1) Agrupe la cantidad binaria en grupos de 4 en 4 iniciando por el lado derecho. Si al terminar de agrupar no completa 4 dígitos, entonces agregue ceros a la izquierda.
2) Posteriormente vea el valor que corresponde de acuerdo a la tabla:

Núm. en binario	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Núm. en hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

3) La cantidad correspondiente en hexadecimal se agrupa de derecha a izquierda.

- 110111010 (binario) = 1BA (hexadecimal). Procesos:

```
1010 = A
1011 = B
1 entonces agregue 0001 = 1
Agrupe de derecha a izquierda: 1BA
```

Hexadecimal a binario

Ídem que para pasar de octal a binario, solo que se remplaza por el equivalente de 4 bits, como de octal a binario.

Tabla de conversión entre decimal, binario, hexadecimal, octal, BCD, Exceso 3 y Código Gray o Reflejado

Decimal	Binario	Hexadecimal	Octal	BCD	Exceso 3	Gray o Reflejado
0	0000	0	0	0000	0011	0000
1	0001	1	1	0001	0100	0001
2	0010	2	2	0010	0101	0011
3	0011	3	3	0011	0110	0010
4	0100	4	4	0100	0111	0110
5	0101	5	5	0101	1000	0111
6	0110	6	6	0110	1001	0101
7	0111	7	7	0111	1010	0100

8	1000	8	10	1000	1011	1100
9	1001	9	11	1001	1100	1101
10	1010	A	12	0001 0000		1111
11	1011	B	13	0001 0001		1110
12	1100	C	14	0001 0010		1010
13	1101	D	15	0001 0011		1011
14	1110	E	16	0001 0100		1001
15	1111	F	17	0001 0101		1000

Construcciones condicionales

Una de las construcciones importantes que pueden especificarse en un programa es el hecho de realizar diferentes tareas en función de ciertas condiciones. Esto es, ejecutar una parte del código u otra, condicionalmente. Para ello será necesario especificar dichas condiciones (ver Sec. 3.4) y disponer de un mecanismo para indicar qué acciones tomar dependiendo de cómo se evalúe una determinada condición en un momento dado de la ejecución del programa.

Antes de empezar, un recordatorio. Como ya de comentó en la sección 3.4.3, C no dispone de valores booleanos o lógicos, que podrían usarse en la evaluación de condiciones. En su defecto, C "simula" los valores falso y cierto, como el valor numérico cero, y cualquier valor no cero (incluyendo negativos), respectivamente.

Así pues, en este capítulo veremos las distintas maneras que C ofrece para controlar el flujo de ejecución de un programa de forma condicional, que son:

- la construcción if,
- el operador condicional `?`, y
- la construcciòn `switch`.

Construcción if

La construcción if es similar a la existente en otros lenguajes de programación, aunque en C posee ciertas peculiaridades. El formato general de esta construcción para decidir si una determinada sentencia debe ejecutarse o no *(alternativa simple)* es el siguiente:

```
if (condición)
sentencia ;
```

La construcción if puede escribirse también de forma más general para controlar la ejecución de un grupo de sentencias, de la siguiente manera:

Obsérvese el uso del modificador de formato %f para la entrada y salida de valores de coma flotante, y el uso de la variable fin para controlar la terminación del bucle.

En el fichero de cabeceras math.h (#include <math.h>), perteneciente a la librería estándar (ver Ap. B), se definen una serie de funciones matemáticas para operar con valores reales, como: sqrt para la raíz cuadrada, sin y cos para senos y cosenos, etc.

4. PROGRAMACION EN LENGUAJE DE ALTO NIVEL: ABSTRACCION DE PROGRAMAS

Abstracción

La **abstracción** consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (característica de caja negra). El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: abstracción de datos (pertenecientes a los datos) y abstracción de control (perteneciente a las estructuras de control).

Los diferentes paradigmas de programación han aumentado su nivel de abstracción, comenzando desde los lenguajes de máquina, lo más próximo al ordenador y más lejano a la comprensión humana; pasando por los lenguajes de comandos, los imperativos, la orientación a objetos (POO), la Programación Orientada a Aspectos (POA); u otros paradigmas como la programación declarativa, etc.

La abstracción encarada desde el punto de vista de la programación orientada a objetos expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás. Además de distinguir entre los objetos provee límites conceptuales. Entonces se puede decir que la encapsulación separa las características esenciales de las no esenciales dentro de un objeto. Si un objeto tiene más características de las necesarias los mismos resultarán difíciles de usar, modificar, construir y comprender.

La misma genera una ilusión de simplicidad dado a que minimiza la cantidad de características que definen a un objeto.

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que sus esfuerzos pudiesen ser utilizados por otras personas se creó la POO que consiste en una serie de normas para garantizar la interoperabilidad entre usuarios de manera que el código se pueda reutilizar.

Tipos de datos estructurados: Tablas

En este capítulo veremos algunos de los mecanismos que C ofrece para la creación de tipos de datos complejos. ´ Estos se construyen, en un principio, a partir de los tipos de datos elementales predefinidos por el lenguaje (ver Cap. 6).

Comenzaremos hablando del tipo abstracto de datos *tabla*, tanto de una (*vectores*), dos (*matrices*) o múltiples dimensiones. En C existe un tratamiento especial para los vectores de caracteres, por lo que dedicaremos una parte de este capítulo a su estudio. Se deja para el capitulo 8 el estudio de otros tipos de datos estructurados, como las *estructuras*, las *uniones*, y los tipos de datos *enumerados*.

Las tablas en C son similares a las que podemos encontrar en otros lenguajes de programación. Sin embargo, se definen de forma diferente y poseen algunas peculiaridades derivadas de la estrecha relación con los punteros. Volveremos a esto más adelante en el capitulo 9.

Vectores

Los *vectores*, también llamados *tablas unidimensionales*, son estructuras de datos caracterizadas por:

- Una colección de datos del mismo tipo.
- Referenciados mediante un mismo nombre.
- Almacenados en posiciones de memoria físicamente contiguas, de forma que, la dirección de memoria más baja corresponde a la del primer elemento, y la dirección de memoria más alta corresponde a la del último elemento.
-

El formato general para la declaración de una variable de tipo vector es el siguiente:

```
tipo_de_datos nombre_tabla [tamaño];
```

donde:

- tipo de datos indica el tipo de los datos almacenados por el vector. Recordemos que todos los elementos del vector son forzosamente del mismo tipo. Debe aparecer necesariamente en la declaración, puesto que de ella depende el espacio de memoria que se reservara para almacenar el vector.

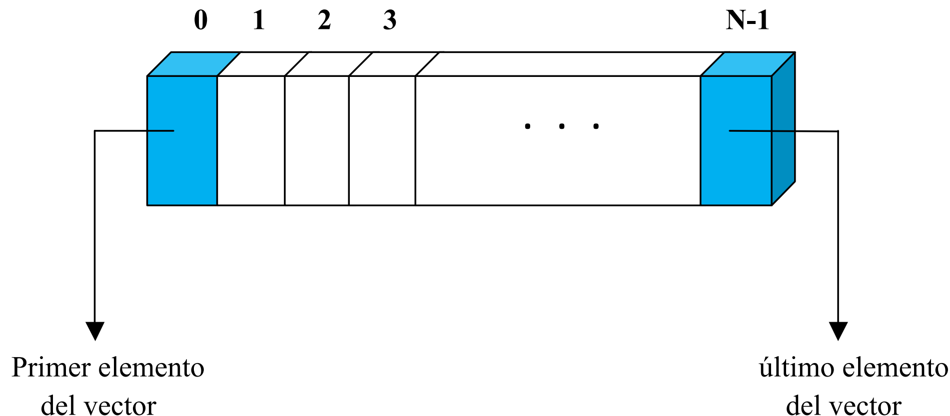


Figura 7.1: Representación gráfica de un vector de N *elementos*

- nombre_tabla es un identificador que usaremos para referiremos tanto al vector como un todo, como a cada uno de sus elementos.
- tamaño es una expresión entera constante que indica el número de elementos que contendrá el vector. El espacio ocupado por un vector en memoria se obtiene como el producto del número de elementos que lo componen y el tamaño de cada uno de estos.

Consulta

El acceso a un elemento de un vector se realiza mediante el nombre de este y un índice entre corchetes ([ ]). El índice representa la posición relativa que ocupa dicho elemento dentro del vector y se especifica mediante una expresión entera (normalmente una constante o una variable). Formalmente:

```
nombre_vector[índice];
```

A continuación se muestran algunas formas válidas de acceso a los elementos de un vector:

```
int contador[10];
int i, j, x;
. . .
x = contador[1];
x = contador[i];
x = contador[i * 2 + j];
. . .
```

Como muestra la figura 7.1, el primer elemento de un vector en C se sitúa en la posición 0, mientras que el ´ultimo lo hace en la posición N - 1 (N indica el número de elementos del vector). Por esta razón, el índice para acceder a los elementos del vector debe permanecer entre estos dos valores. Es responsabilidad del programador garantizar este hecho, para no acceder a posiciones de memoria fuera del vector, lo cual produciría errores imprevisibles en el programa.

Asignación

La asignación de valores a los elementos de un vector se realiza de forma similar a como se consultan.
Veámoslo en un ejemplo:

```
int contador[3];
. . .
contador[0] = 24;
contador[1] = 12;
contador[2] = 6;
```

En muchas ocasiones, antes de usar un vector (una tabla en general) por primera vez, es necesario dar a sus elementos un valor inicial. La manera más habitual de inicializar un vector en tiempo de ejecución consiste en recorrer secuencialmente todos

sus elementos y darles el valor inicial que les corresponda. Veámoslo en el siguiente ejemplo, donde todos los elementos de un vector de números enteros toman el valor 0:

```
#define TAM 100
void main()
{
    int vector[TAM], i;
    for (i= 0; i< TAM; i++)
        vector[i] = 0;
}
```

Nótese que el bucle recorre los elementos del vector empezando por el elemento 0 (i=0) y hasta el elemento TAM-1 (condición i<TAM).

Existe también un mecanismo que permite asignar un valor a todos los elementos de un tabla con una sola sentencia. Concretamente en la propia declaración de la tabla. La forma general de inicializar una tabla de cualquier número de dimensiones es la siguiente:

```
tipo_de datos_nombre_tabla [tam1]...[tamN] = { lista valores };
```

La lista de valores no deberá contener nunca más valores de los que puedan almacenarse en la tabla. Veamos algunos ejemplos:

```
int contador[3] = {24, 12, 6}; /* Correcto */
char vocales[5] = {'a', 'e', 'i', 'o', 'u'}; /* Correcto */
int v[4] = {2, 6, 8, 9, 10, 38}; /* Incorrecto */
```

Finalmente, cabe destacar que no está permitido en ningún caso comparar dos vectores (en general dos tablas de cualquier número de dimensiones) utilizando los operadores relacionales que vimos en la sección 3.4.3. Tampoco está permitida la copia de toda una tabla en otra con una simple asignación. Si se desea comparar o copiar toda la información almacenada en dos tablas, deberá hacerse elemento a elemento.

Los mecanismos de acceso descritos en esta sección son idénticos para las matrices y las tablas multidimensionales. La única diferencia es que será necesario especificar tantos índice como dimensiones posea la tabla a la que se desea acceder. Esto lo veremos en las siguientes secciones.

Ejemplos

A continuación se muestra un programa que cuenta el número de apariciones de los números 0, 1, 2 y 3 en una secuencia de enteros acabada en -1.

```
#include <stdio.h>
void main ()
{
    int num, c0=0, c1=0, c2=0, c3=0;

    scanf("%d", &num);
    while (num != -1)
    {
        if (num == 0) c0++;
        if (num == 1) c1++;
        if (num == 2) c2++;
        if (num == 3) c3++;
        scanf( "%d", &num );
     }
     printf( "Contadores:%d, %d, %d, %d\n", c0, c1, c2, c3 );
}
```

¿Qué ocurriría si tuviésemos que contar las apariciones de los cien primeros números enteros? ¿Deberíamos declarar cien contadores y escribir cien construcciones if para cada caso? La respuesta, como era de esperar, se halla en el uso de vectores. Veámoslo en el siguiente programa:

```
#include <stdio.h>
#define MAX 100
void main ()
{
    int i, num, cont[MAX];

    for (i= 0; i< MAX; i++)
        cont[i] = 0;
    scanf("%d", &num);
    while (num != -1) f
         if ((num >= 0) && (num <= MAX))
            cont[num]++;
        scanf( "%d", &num );
     {
   for (i= 0; i< MAX; i++)
       printf( "Contador [%d] = %d\n", i, cont[i] );
}
```

Veamos finalmente otro ejemplo en el que se muestra cómo normalizar un vector de números reales.
La normalización consiste en dividir todos los elementos del vector por la raíz cuadrada de la suma de sus cuadrados. Destaca el uso de la constante MAX para definir el número de elementos del vector y de la función sqrt para el cálculo de raices cuadradas.

```
#include <math.h>
```

```
#include<stdio.h>

int main()
{
int i,producto;
int contador[10]={12,34,45,11,2,4,6,87,99,10};

for(i=0;i<10;i++)
{
printf("\n contador[%d]: %d",i,contador[i]);
}

producto=contador[2]*contador[9]+contador[0];
printf("\n La operacion es: %d",producto);
return 0;
}
```

```
#include<stdio.h>

int main()
{
int i;
float producto;
float contador[10]={1.2,3.4,4.5,1.1,2.0,4.0,6.0,8.7,9.9,1.0};

for(i=0;i<10;i++)
{
printf("\n contador[%d]: %f",i,contador[i]);
}

producto=contador[2]*contador[9]+contador[0];

printf("\n La operacion es: %f",producto);
return 0;
}
```

```
#include <stdio.h>
#define MAX 10
void main()
{
    float vector[MAX], modulo;
    int i;

    /* Lectura del vector. */
    for (i= 0; i< MAX; i++)
        scanf("%f", &vector[i]);

/* Calcular modulo. */
modulo = 0.0;
for (i= 0; i< MAX; i++)
    modulo = modulo + (vector[i] * vector[i]);
modulo = sqrt(modulo);

/* Normalizar */
for (i= 0; i< MAX; i++)
    vector[i] /= modulo;

/* Escritura del vector. */
for (i= 0; i< MAX; i++ )
    printf( "%f ", vector[i] );
}
```

Matrices

Las *matrices*, también llamadas *tablas bidimensionales*, no son otra cosa que vectores con dos dimensiones. Por lo que los conceptos de acceso, inicialización, etc. son similares.

La declaración de una variable matriz tiene la forma siguiente:

```
tipo_de_datos_nombre_tabla [tamaño_dim1][tamaño_dim2];
```

Donde tamaño_dim1 y tamaño_dim2 indican, respectivamente, el número de filas y de columnas que componen la matriz. La figura 7.2 muestra la representación gráfica habitual de una matriz de datos.

Otro hecho importante es que las matrices en C se almacenan "por filas". Es decir, que los elementos de cada fila se sitúan en memoria de forma contigua. Así pues, en la matriz de la figura anterior, el primer elemento almacenado en memoria es el (0,0), el segundo el (0,1), el tercero el (0,2),. . ., (0,M-1), después (1,0), y asi sucesivamente hasta el ´último elemento, es decir (N-1,M-1).

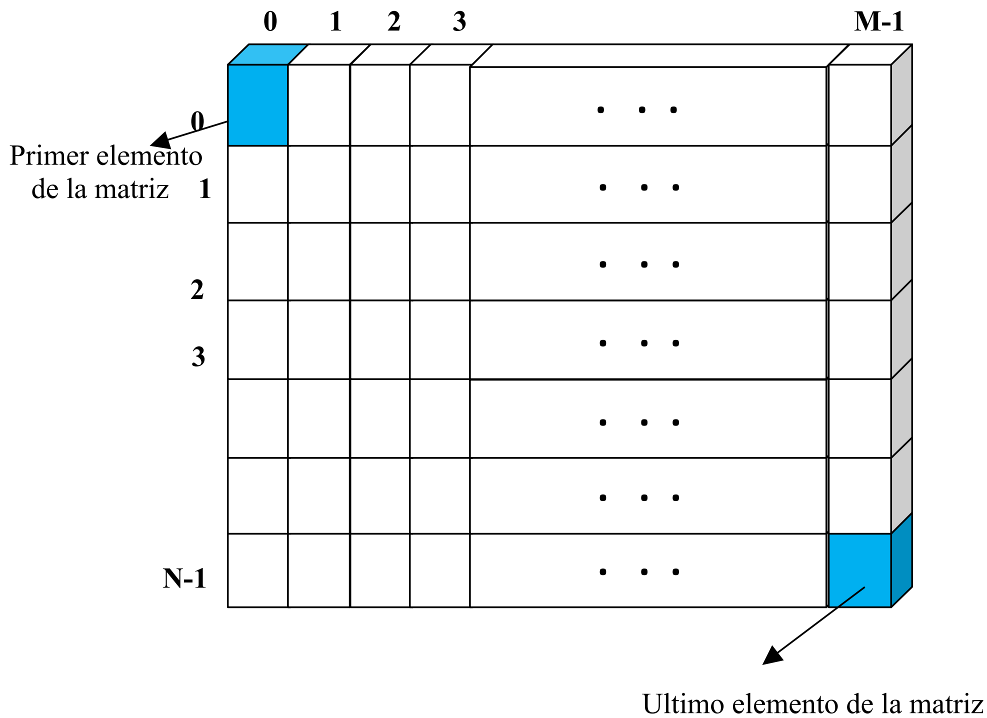


Figura 7.2: Representación gráfica de una matriz de N *filas y* M *columnas*

Consulta

El acceso a un elemento de una matriz se realiza mediante el nombre de esta y dos índices (uno para cada dimensión) entre corchetes. El primer índice representa la fila y el segundo la columna en que se encuentra dicho elemento. Tal como muestra la figura 7.2, el índice de las filas tomará valores entre 0 y el número de filas menos 1, mientras que el índice de las columnas tomará valores entre 0 y el número de columnas menos 1. Es responsabilidad del programador garantizar este hecho.

```
nombre_matriz[índice_1][índice_2];
```

Asignación

Comentaremos únicamente la inicialización de una matriz en la propia declaración. El siguiente ejemplo declara una matriz de tres filas y cuatro columnas y la inicializa. Por claridad, se ha usado identacion en los datos, aunque hubiesen podido escribirse todos en una sola línea.

```
int mat[3][4] = { 24, 12, 6, 17,
                  15, 28, 78, 32,
                  0, 44, 3200 , -34
                };
```

La inicialización de matrices, y en general de tablas multidimensionales, puede expresarse de forma más clara agrupando los valores mediante llaves ({ }), siguiendo la estructura de la matriz. Así pues, el ejemplo anterior también podría escribirse como:

```
int mat[3][4] = {{ 24, 12, 6, 17 },
                 { 15, 28, 78, 32 },
                 { 0, 44, 3200, -34 }
                };
```

Ejemplo
El siguiente ejemplo calcula la matriz traspuesta de una matriz de enteros. La matriz tendría unas dimensiones máximas según la constante MAX.

```
#include <stdio.h>
#define MAX 20
void main()
{
    int filas, columnas, i, j;
    int matriz[MAX][MAX], matrizT[MAX][MAX];

    /* Lectura matriz */
   printf( "Num. filas, Num. columnas: " );
   scanf( "%d%d", &filas, &columnas );
   printf ("Introducir matriz por filas:" );
   for (i= 0; i< filas; i++)
      for (j= 0; j< columnas; j++)
      {
          fprintf( "\nmatriz[%d][%d] = ", i, j );
          scanf( "%d", &matriz[i][j] );
      }

   /* Traspuesta */
   for (i= 0; i< filas; i++)
      for (j= 0; j< columnas; j++)
          matrizT[j][i] = matriz[i][j];

  /* Escritura del resultado */
  for (i= 0; i< filas; i++)
     for (j= 0; j< columnas; j++)
        printf( "\nmatrizT[%d][%d] = %d ",
                i, j, matrizT[i][j] );
```

```
#include<stdio.h>

int main()
{
int i,j,producto;

int mat[3][4] = {{ 24, 12, 6, 17 }, { 15, 28, 78, 32 },{ 0, 44, 3200,
-34 } };

for(i=0;i<3;i++)
for(j=0;j<4;j++)
{
printf("\n matriz[%d][%d]: %d",i,j,mat[i][j]);
}

producto=mat[2][1]*mat[1][2]+mat[0][0];

printf("\n La operacion es: %d",producto);
return 0;
}
```

```c
#include<stdio.h>

int main()
{
int i,j,producto;

int mat[3][3] = {{ 24, 12, 6}, {28, 78, 32 },{ 44, 3200, -34 } };

for(i=0;i<3;i++)
for(j=0;j<3;j++)
{
printf("\n matriz[%d][%d]: %d",i,j,mat[i][j]);

}

producto=mat[2][1]*mat[1][2]+mat[0][0];

printf("\n La operacion es: %d",producto);
return 0;
}
```

```
#include<stdio.h>

int main()
{
int i,j,suma[2][2];

int mat1[2][2] = {{12, 6}, {28, 32 }};
int mat2[2][2] = {{1, 1}, {2, 2 }};

for(i=0;i<2;i++)
for(j=0;j<2;j++)
{
suma[i][j]=mat1[i][j]+mat2[i][j];
printf("\n La suma de las matrices es: suma[%d][%d]: %d",i,j,suma[i]
[j]);
}

return 0;
}
```

}

Obsérvese que para recorrer todos los elementos de una matriz es necesario el empleo de dos bucles anidados que controlen los índices de filas y columnas (siempre entre 0 y el número de filas o columnas menos 1). En este ejemplo todos los recorridos se realizan "por filas", es decir, que primero se visitan todos los elementos de una fila, luego los de la siguiente, etc. Finalmente, cabe comentar que para el recorrido de tablas multidimensionales será necesario el empleo de tantos bucles como dimensiones tenga la tabla.

Tablas multidimensionales

Este tipo de tablas se caracteriza por tener tres o más dimensiones. Al igual que vectores y matrices, todos los elementos almacenados en ellas son del mismo tipo de datos.
La declaración de una tabla multidimensional tiene la forma siguiente:

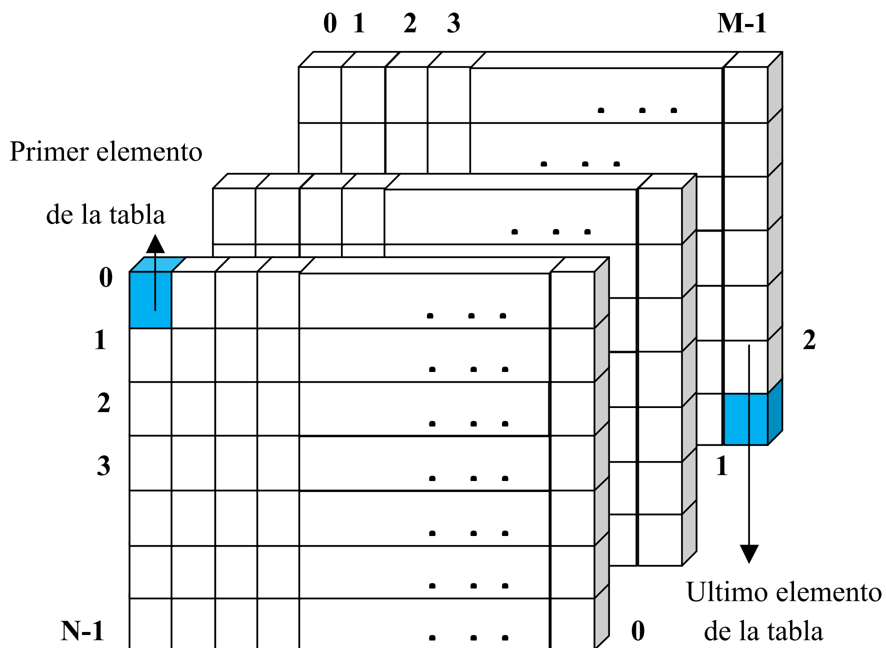


Figura 7.3: Representación gráfica de una tabla de tres dimensiones: N x M x 3

```
tipo_de datos_nombre_tabla [tamaño dim1] ...[tamaño dimN];
```

Para acceder a un elemento en particular será necesario usar tantos índices como dimensiones:

```
nombre_vector[índice 1] ...[índice N];
```

Aunque pueden definirse tablas de más de tres dimensiones, no es muy habitual hacerlo. La figura 7.3 muestra como ejemplo la representación gráfica habitual de una tabla de tres dimensiones.

Ejemplo

El siguiente ejemplo muestra el empleo de una tabla multidimensional. Concretamente, el programa utiliza una tabla de 3 dimensiones para almacenar 1000 números aleatorios. Para generar números aleatorios se usa la función rand de la librería estándar stdlib.h. También se ha usado la función getchar (stdio.h), que interrumpe el programa y espera a que el usuario presione una tecla.

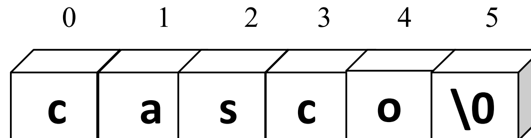
```
#include <stdio.h>
#include <stdlib.h>
#define DIM 10
void main()
{
  int tabla_random [DIM][DIM][DIM], a, b, c;

  for (a= 0; a< DIM; a++)
     for (b= 0; b< DIM; b++)
        for (c= 0; c< DIM; c++)
           tabla_random[a][b][c] = rand();

/* Muestra series de DIM en DIM elementos. */
for (a= 0; a< DIM; a++)
    for (b= 0; b < DIM; b++)
    {
       for (c= 0; c < DIM; c++)
       {
          printf( "\n tabla[%d][%d][%d] = ", a, b, c );
          printf( "%d", table_random[a][b][c] );
       }
       printf( "\nPulse ENTER para seguir" );
       getchar();
    }
}
```

Cadenas de caracteres

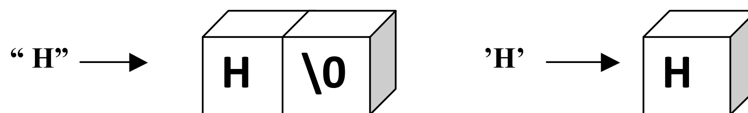
Las cadenas de caracteres son vectores de tipo carácter (char) que reciben un tratamiento especial para simular el tipo de datos "*string*", presente en otros lenguajes de programación.

Para que un vector de caracteres pueda ser considerado como una cadena de caracteres, el último de los elementos útiles del vector debe ser el carácter *nulo* (código ASCII 0). Según esto, si se quiere declarar una cadena formada por N caracteres, deberá declararse

un vector con N + 1 elementos de tipo carácter. Por ejemplo, la declaración char cadena[6]; reserva suficiente espacio en memoria para almacenar una cadena de 5 caracteres, como la palabra "casco":



En C pueden definirse constantes correspondientes a cadenas de caracteres. Se usan comillas dobles para delimitar el principio y el final de la cadena, a diferencia de las comillas simples empleadas con las constantes de tipo carácter. Por ejemplo, la cadena constante "H" tiene muy poco que ver con el carácter constante ’H’, si observamos la representación interna de ambos:



Asignación

Mientras que la consulta de elementos de una cadena de caracteres se realiza de la misma forma que con los vectores, las asignaciones tienen ciertas peculiaridades.

Como en toda tabla, puede asignarse cada carácter de la cadena individualmente. No deberá olvidarse en ningún caso que el ultimo carácter valido de la misma debe ser el carácter nulo (’\0’). El siguiente ejemplo inicializa la cadena de caracteres cadena con la palabra "casco". Nótese que las tres últimas posiciones del vector no se han usado. Es más, aunque se les hubiese asignado algún carácter, su contenido sería ignorado. Esto es, el contenido del vector en las posiciones posteriores al carácter nulo es ignorado.

```
char cadena[10];

. . .
cadena[0] = ’c’;
cadena[1] = ’a’;
cadena[2] = ’s’;
cadena[3] = ’c’;
cadena[4] = ’o’;
cadena[5] = ’\0’;
```

La inicialización de una cadena de caracteres durante la declaración puede hacerse del mismo modo que en los vectores, aunque no debe olvidarse incluir el

carácter nulo al final de la cadena. Sin embargo, existe un método de inicialización propio de las cadena de caracteres, cuyo formato general es:

```
char nombre [tamaño] = "cadena";
```

Usando este tipo de inicialización, el carácter nulo es añadido automáticamente al final de la cadena. Así pues, una inicialización típica de vectores como la siguiente:

```
char nombre[10] = { ’N’, ’U’, ’R’, ’I’, ’A’, ’\0’ };
```

puede hacerse también de forma equivalente como:

```
char nombre[10] = "NURIA";
```

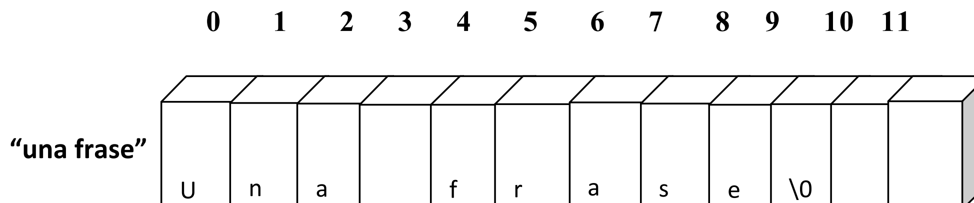
Finalmente, la inicialización anterior puede hacerse sin necesidad de especificar el tamaño del vector correspondiente. En este caso, el compilador se encarga de calcularlo automáticamente, reservando espacio de memoria suficiente para almacenar la cadena, incluyendo el carácter nulo al final. Así pues, la siguiente declaración reserva memoria para almacenar 6 caracteres y los inicializa adecuadamente con las letras de la palabra NURIA:

```
char nombre[] = "NURIA";
```

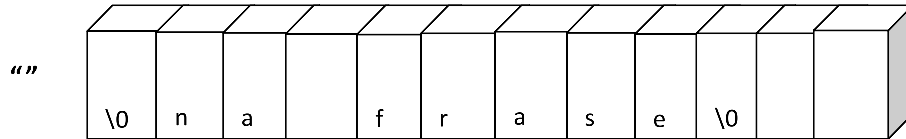
La cadena vacía

Otra curiosidad de las cadenas de caracteres se refiere a la cadena vacía, " " , que consta únicamente del carácter nulo. Puesto que los caracteres posteriores al carácter nulo son ignorados, convertir una cadena con cualquier valor almacenado a la cadena vacía es tan simple como asignar el carácter nulo a la posición 0 de dicha cadena. He aquí un ejemplo:

```
char cadena [12] = "Una frase";
. . .
cadena[0] = ’\0’; /* Ahora es una cadena vacia */
```



Cadena [0]='\0';



Manejo de cadenas de caracteres

Aunque C no incorpora en su definición operadores para el manejo de cadenas de caracteres, todo compilador de C proporciona una librería estándar (string.h) con funciones para facilitar su utilización. Destacar algunas de ellas:

- strlen para obtener la longitud de la cadena, sin contar el carácter nulo,
- strcpy para copiar una cadena en otra,
- strcat para concatenar dos cadenas,
- strcmp para comparar dos cadenas, etc.

Para más información sobre estas y otras funciones, consultar el apéndice B.

Entrada y salida

En cuanto a la entrada y salida de cadenas de caracteres, existe un formato especial %s que puede utilizarse en las funciones scanf y printf. Por ejemplo, la siguiente sentencia leerá una cadena de caracteres en la variable cad. Solo se asignaran caracteres mientras no sean caracteres blancos, tabuladores o saltos de línea. Por lo tanto, el empleo de %s solo tendrá sentido para la lectura de palabras.
Además del formato %s existen los formatos %[ˆabc] y %[abc], que permiten leer respectivamente una cadena de caracteres hasta encontrar algún carácter del conjunto {a, b, c}, o bien hasta no encontrar un carácter del conjunto {a, b, c}. En cualquier caso el carácter del conjunto {a, b, c} no es leído. Ver el apéndice B para más información sobre el empleo de scanf y la lectura de cadenas de caracteres.

```
char cad[20];
. . .
scanf("%s", cad);
```

Nótese que, en el ejemplo, no se ha antepuesto el símbolo & a la variable cad. Por el momento, tengámoslo en mente y esperemos hasta el capitulo 9 para comprender a que se debe este hecho.
La librería estándar de entrada y salida (stdio.h) proporciona además las funciones gets puts, que permiten leer de teclado y mostrar por pantalla una cadena de caracteres completa, respectivamente (ver el apéndice B para mas detalles).

Ejemplos

Para finalizar, veamos un par de ejemplos de manejo de cadenas de caracteres.

El siguiente programa cuenta el número de veces que se repite una palabra en una frase. El programa emplea la función de comparación de cadenas strcmp. Dicha función devuelve 0 en caso de que las cadenas comparadas sean iguales (ver Sec. B.1).

```
#include <stdio.h>
#include <string.h>
#define MAXLIN 100
void main()
{
    char pal[MAXLIN]; /* La que buscamos. */
    char palfrase[MAXLIN]; /* Una palabra de la frase. */
    char c;
    int total = 0;

    printf( "\nPALABRA:" );
    scanf( "%s", pal );
    printf( "\nFRASE:" );
    c = ' ';
    while (c != '\n')
    {
        scanf( "%s%c", palfrase, &c );
        if (strcmp(pal, palfrase) == 0)
          total++;
   }
   printf( "\nLa palabra %s aparece %d veces.", pal, total );
}
```

A continuación se muestra otro ejemplo que hace uso de las cadenas de caracteres. El programa lee dos cadenas de caracteres, las concatena, convierte las letras minúsculas en mayúsculas y viceversa, y finalmente escribe la cadena resultante.

```
#include <stdio.h>
#include <string.h>
void main()
{
   char cad1[80], cad2[80], cad3[160];
   int i, delta;
   printf( "Introduzca la primera cadena:\n" );
  gets(cad1);
  printf( "Introduzca la segunda cadena:\n" );
  gets( cad2 );
  /* cad3 = cad1 + cad2 */
  strcpy( cad3, cad1 );

  strcat( cad3, cad2 );
```

```
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.
- Son 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.

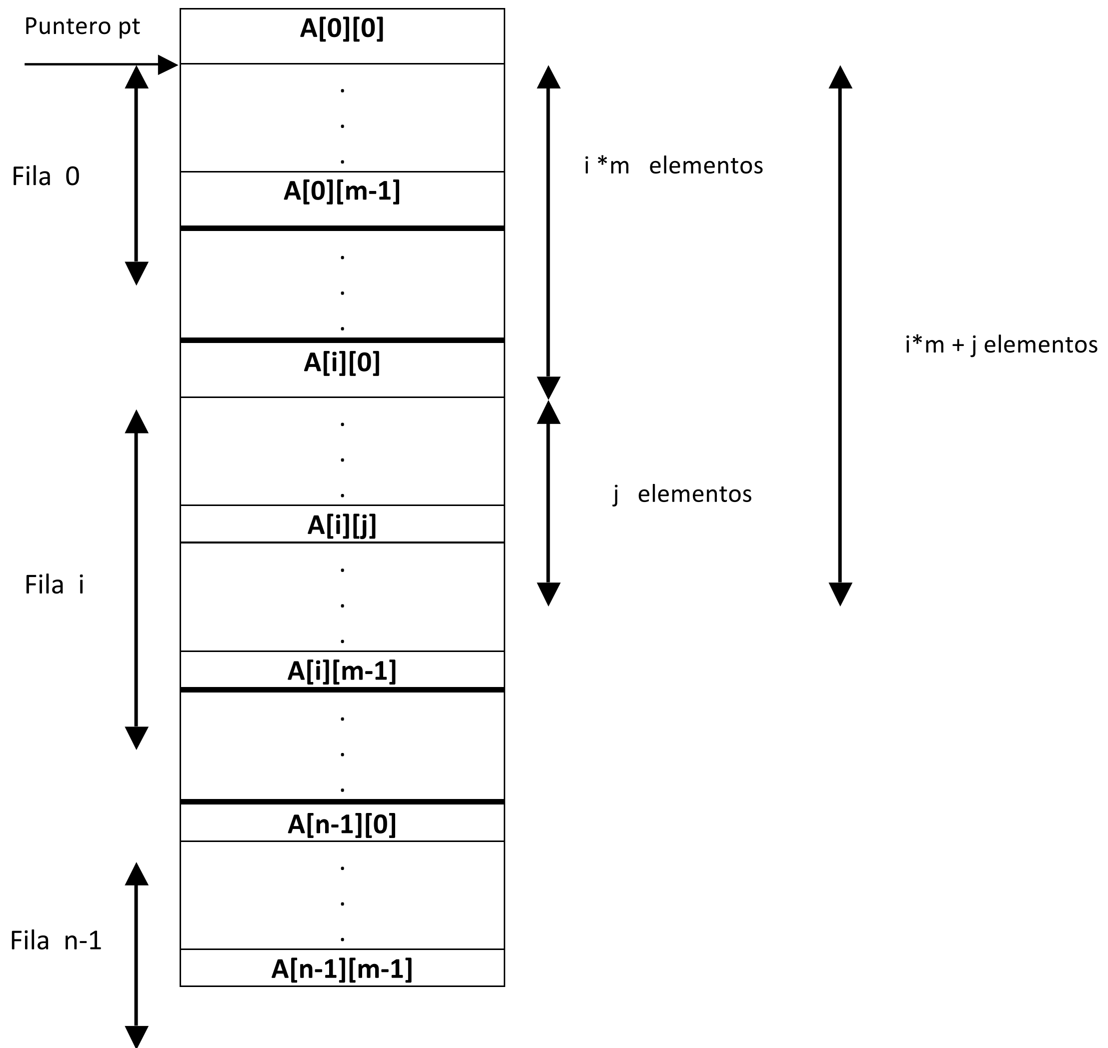


Figura 9.1: Acceso a una matriz mediante un puntero

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;
```

```
    Tstring nombre;
    Tstring direc;
    long int telf;
} Tempresa;
```

De forma que en el programa puede declararse una variable cliente de tipo Tnif y un puntero pc a dicho tipo, asi como asignar a pc la dirección de inicio de la variable estructura cliente :

```
Tnif cliente;
Tnif *pc;
. . .
pc = &cliente;
```

Para acceder a un campo individual en una estructura apuntada por un puntero puede usarse el operador de indireccion * junto con el operador punto (.) habitual de las estructuras. Vemos un ejemplo:

```
(*pc).letra = ’Z’;
scanf( "%ld", &(*pc).num );
```

Los paréntesis son necesarios ya que el operador punto (.) tiene más prioridad que el operador *.
Nótese que el operador de dirección & en la llamada a scanf se refiere al campo num y no al puntero pc.

El hecho de que sea obligatorio usar paréntesis en esta notación hace que se generen errores debido al olvido de los paréntesis adecuados. Para evitar estos errores, C posee otra notación para acceder a los campos de una estructura apuntada por un puntero.

El acceso a campos individuales puede hacerse también mediante el operador especial de los punteros a estructuras -> (guion seguido del símbolo de mayor que). Así pues, puede escribirse el mismo ejemplo anterior como:

```
pc->letra = ’Z’;
scanf( "%ld", &pc->num );
```

El operador -> tiene la misma prioridad que el operador punto (.). *Es recomendable usar el operador -> al manejar* structs *apuntados por punteros*.

El acceso a los campos de una estructura anidada a partir de un puntero puede hacerse combinando los operadores de punteros con el punto (.). Veamos el siguiente ejemplo:

```
Tempresa emp;
Tempresa *pe;
char inicial;
. . .
```

```
pe = &emp;
pe->nif.letra = 'Z';
scanf( "%ld", &pe->nif.num );
gets( pe->nombre );
inicial = pe->nombre[0];
. . .
```

5. PROGRAMACION ESTRUTURADA Y RECURSION

En programación Estructurada los programadores deben profundizar mas que lo usual al procederá realizar el diseño original del programa, pero el resultado final es más fácil de leer y comprender, el objetivo de u programador profesional al escribir programas de una manera estructurada, es realizarlos utilizando solamente un numero de bifurcaciones de control estandarizados.

EL resultado de aplicar la sistemática y disciplinada manera de elaboración de programas establecida por la Programación Estructurada es una programación de alta precisión como nunca antes había sido lograda. Las pruebas de los programas, desarrollados utilizando este método, se acoplan mas rápidamente y el resultado final con programas que pueden ser leídos, mantenidos y modificados por otros programadores con mucho mayor facilidad.

DEFINICIONES

Programación Estructurada es una técnica en la cual la estructura de un programa, esto es, la interpelación de sus partes realiza tan claramente como es posible mediante el uso de tres estructuras lógicas de control:

- Secuencia: Sucesión simple de dos o mas operaciones.
- Selección: bifurcación condicional de una o mas operaciones.
- Interacción: Repetición de una operación mientras se cumple una condición.

Un programa estructurado esta compuesto de segmentos, los cuales puedan estar constituidos por unas pocas instrucciones o por una pagina o más de codificación. Cada segmento tiene solamente una entrada y una salida, estos segmentos, asumiendo que no poseen lazos infinitos y no tienen instrucciones que jamas se ejecuten, se denominan programas propios.

Cuando varios programas propios se combinan utilizando las tres estructuras básicas de control mencionadas anteriormente, el resultado es también un programa propio.

Un programa no estructurado es un programa procedimental: las instrucciones se ejecutan en el mismo orden en que han sido escritas. Sin embargo, este tipo de programación emplea la instrucción "goto".

Una instrucción "goto" permite pasar el control a cualquier otra parte del programa. Cuando se ejecuta una instrucción "goto" la secuencia de ejecución del programa continúa a partir de la instrucción indicada por "goto". De esta forma, para comprender como funciona un programa es necesario simular su ejecución. Esto quiere decir que en la mayoría de los casos es muy difícil comprender la lógica de un programa de este tipo.

Algunos compiladores crean referencias cruzadas a las instrucciones apuntadas por los "goto", posibilitando una navegación rápida a través del código fuente. Sin embargo, es algo común en muchos lenguajes de programación el empleo de una variable en asociación con el destino del "goto", no permitiendo la creación automática de tablas de referencias cruzadas.

Top-Down

También conocida como de arriba-abajo y consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al problema. Consiste en efectuar una relación entre las etapas de la estructuración de forma que una etapa jerárquica y su inmediato inferior se relacionen mediante entradas y salidas de información.

Este diseño consiste en una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa. La utilización de la técnica de diseño Top-Down tiene los siguientes objetivos básicos:

- Simplificación del problema y de los subprogramas de cada descomposición.
- Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloque o módulos lo que hace mas sencilla su lectura y mantenimiento.

PROCEDIMIENTO Y FUNCIONES

Aunque con estos tres tipos de sentencias se puede construir cualquier clase de programa estructurado, se no garantiza su legibilidad. Para asegurarla se crean consecuencias lineales independientes. Cuando un bloque de instrucciones se maneja como si fuera una única instrucción, se ha creado procedimientos.

Este, a su vez, puede constar de otros tantos bloques independientes.

Por otra parte, cada bloque se instrucciones utiliza variables que no deben ser siempre las mismas. Las variables de trabajo de un bloque son transferidas desde el bloque superior a través de los argumentos de entrada al procedimiento o función. Los resultados obtenidos pueden, a su vez, emplearse por otros modulos.

Las variables implicadas en un procedimiento o función pueden ser de uso exclusivo de que este bloque, o compartirse con el modulo que los llama. Para asegurarse la flexibilidad de los modulos, cada una de las partes de que se componen de ser autónoma y contener, por tanto sus propias variables independientes.

Las modificaciones sucesivas no presentaran de esta forma problemas, ya que cada variable se usa en un modulo concreto.

```
#include<stdio.h>

struct data
{
float cantidad;
char nombre [30];
char apellido[30];
} rec;


int main(void)
{
printf("Nombre y apellido del donador,\n");
printf("separado por un espacio:");
scanf("%s%s", rec.nombre,rec.apellido);
printf("\n Catidad donada:");
scanf("%f",&rec.cantidad);
printf("\n El donador %s %s dio $%.2f.\n",
rec.nombre,rec.apellido,rec.cantidad);
return 0;
}
```

```c
/*Demostracion de estructuras usando arreglos*/

#include<stdio.h>

struct entrada
{
char nombre[20];
char apellido[20];
char telefono[10];
};

/*Declaracion de un arreglo de estructura*/

struct entrada lista[4];

int i;
int main(void)
{
/*Lazo al ingreso de datos de 4 personas*/

for(i=0; i<4; i++)
{
printf("\nDame el primer nombre: ");
scanf("%s",lista[i].nombre);
printf("Dame el el apellido: ");
scanf("%s",lista[i].apellido);
printf("Dame el telefono en formato 123-4567:");
scanf("%s",lista[i].telefono);
}

/*Imprime dos lineas blancas*/

printf("\n\n");

/*Lazo a los datos desplegados*/

for(i=0; i<4; i++)
{
printf("Nombre:%s %s",lista[i].nombre,lista[i].apellido);
printf("\t\t Telefono: %s\n", lista[i].telefono);
}

return 0;
}
```