9 Arrays (Matrices)

¿Qué es un array?

La definición sería algo así:

> Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre y se diferencian en el índice.

Pero ¿qué quiere decir esto y para qué lo queremos?. Pues bien, supongamos que somos un metereólogo y queremos guardar en el ordenador la temperatura que ha hecho cada hora del dia. Para darle cierta utilidad al final calcularemos la media de las temperaturas. Con lo que sabemos hasta ahora sería algo así (que nadie se moleste ni en probarlo):

```
#include <stdio.h>

int main()
     {
     /* Declaramos 24 variables, una para cada hora del dia */
      int temp1,  temp2,  temp3,  temp4,  temp5,  temp6,  temp7,
      temp8;
      int temp9,  temp10, temp11, temp12, temp13, temp14, temp15,
      temp16;
      int temp17, temp18, temp19, temp20, temp21, temp22, temp23,
      temp0;
       int media;

     /* Ahora tenemos que dar el valor de cada una */
     printf( "Temperatura de las 0: " );
     scanf( "%i", &temp0 );
     printf( "Temperatura de las 1: " );
     scanf( "%i", &temp1 );
     printf( "Temperatura de las 2: " );
     scanf( "%i", &temp2 );
     ...
     printf( "Temperatura de las 23: " );
     scanf( "%i", &temp23 );

      media = (temp0+temp1+temp2+temp3+temp4+ ... +temp23)/24;
     printf( "\nLa temperatura media es %i\n", media );
     }
```

NOTA: Los puntos suspensivos los he puesto para no tener que escribir todo y que no ocupe tanto, no se pueden usar en un programa.

Para acortar un poco el programa podríamos hacer algo así:

```
#include <stdio.h>

int main()
     {
     /* Declaramos 24 variables, una para cada hora del dia */
      int temp1,  temp2,  temp3,  temp4,  temp5,  temp6,  temp7,
      temp8;
      int temp9,  temp10, temp11, temp12, temp13, temp14, temp15,
      temp16;
      int temp17, temp18, temp19, temp20, temp21, temp22, temp23,
      temp0;
     int media;

     /* Ahora tenemos que dar el valor de cada una */
      printf( "Introduzca las temperaturas desde las 0 hasta las 23
      separadas por un espacion: " );
      scanf( "%i %i %i ... %i", &temp0, &temp1, &temp2, ... &temp23 );

      media = (temp0+temp1+temp2+temp3+temp4+ ... +temp23)/24;
      printf( "\nLa temperatura media es %i\n", media );
     }
```

Y esto con un ejemplo que tiene tan sólo 24 variables, ¡¡imagínate si son más!!

Y precisamente aquí es donde nos vienen de perlas los arrays. Vamos a hacer el programa con un array. Usaremos nuestros conocimientos de bucles for y de scanf.

```
#include <stdio.h>

int main()
     {
     int temp[24];/*Con esto ya tenemos declaradas las 24 variables */
     float media;
     int hora;

     /* Ahora tenemos que dar el valor de cada una */
     for( hora=0; hora<24; hora++ )
          {
          printf( "Temperatura de las %i: ", hora );
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / 24;
     printf( "\nLa temperatura media es %f\n", media );
     }
```

Como ves es un programa más rápido de escribir (y es menos aburrido hacerlo), y más cómodo para el usuario que el anterior.

Como ya hemos comentado cuando declaramos una variable lo que estamos haciendo es reservar una zona de la memoria para ella. Cuando declaramos un array lo que hacemos (en este ejemplo) es reservar espacio en memoria para 24 variables de tipo int. El tamaño del array (24) lo indicamos entre corchetes al definirlo. Esta es la parte de la definición que dice: *Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre*.

La parte final de la definición dice: *y se diferencian en el índice*. En ejemplo recorremos la matriz mediante un bucle for y vamos dando valores a los distintos elementos de la matriz. Para indicar a qué elemento nos referimos usamos un número entre corchetes (en este caso la variable hora), este número es lo que se llama **Indice**. El primer elemento de la matriz en **C** tiene el índice 0, El segundo tiene el 1 y así sucesivamente. De modo que si queremos dar un valor al elemento 4 (índice 3) haremos:

```
temp[ 3 ] = 20;
```

NOTA: No hay que confundirse. En la declaración del array el número entre corchetes es el número de elementos, en cambio cuando ya usamos la matriz el número entre corchetes es el índice.

Declaración de un Array

La forma general de declarar un array es la siguiente:

```
tipo_de_dato nombre_del_array[ dimensión ];
```

El **tipo_de_dato** es uno de los tipos de datos conocidos (int, char, float...) o de los definidos por el usuario con typdef. En el ejemplo era int.

El **nombre_del_array** es el nombre que damos al array, en el ejemplo era temp.

La **dimensión** es el número de elementos que tiene el array.

Como he indicado antes, al declarar un array reservamos en memoria tantas variables del *tipo_de_dato* como las indicada en *dimensión*.

Sobre la dimensión de un Array

Hemos visto en el ejemplo que tenemos que indicar en varios sitios el tamaño del array: en la declaración, en el bucle for y al calcular la media. Este es un programa pequeño, en un programa mayor probablemente habrá que escribirlo muchas más veces. Si en un momento dado queremos cambiar la dimensión del array tendremos que cambiar todos. Si nos equivocamos al escribir el tamaño (ponemos 25 en vez de 24) cometeremos un error y puede que no nos demos cuenta. Por eso es mejor usar una constante con nombre, por ejemplo ELEMENTOS.

```
#include <stdio.h>
#define ELEMENTOS       24
int main()
     {
      int temp[ELEMENTOS];/*Con esto ya tenemos declaradas las 24
      variables */
      float media;
      int hora;
     /* Ahora tenemos que dar el valor de cada una */
     for( hora=0; hora<ELEMENTOS; hora++ )
          {
          printf( "Temperatura de las %i: ", hora );
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / ELEMENTOS;
     printf( "\nLa temperatura media es %f\n", media );
     }
```

Ahora con sólo cambiar el valor de elementos una vez lo estaremos haciendo en todo el programa.

Inicializar un array

En **C** se pueden inicializar los arrays al declararlos igual que hacíamos con las variables. Recordemos que se podía hacer:

```
int hojas = 34;
```

Pues con arrays se puede hacer:

```
int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 14, 13, 12 };
```

Ahora el elemento 0 (que será el primero), es decir temperaturas[0] valdrá 15. El elemento 1 (el segundo) valdrá 18 y así con todos. Vamos a ver un ejemplo:

```
#include <stdio.h>
int main()
     {
      int hora;
      int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
      24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 14, 13, 12 };

     for (hora=0 ; hora<24 ; hora++ )
         {
        printf( "La temperatura a las %i era de %i grados.\n", hora,
        temperaturas[hora] );
         }
     }
```

Pero a ver quién es el habilidoso que no se equivoca al meter los datos, no es difícil olvidarse alguno. Hemos indicado al compilador que nos reserve memoria para un array de 24 elementos de tipo int. ¿Qué ocurre si metemos menos de los reservados? Pues no pasa nada, sólo que los elementos que falten valdrán cero.

```
#include <stdio.h>
int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
     int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
      24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };
      for (hora=0 ; hora<24 ; hora++ )
         {
            printf( "La temperatura a las %i era de %i grados.\n",
            hora, temperaturas[hora] );
         }
     }
```

El resultado será:

```
La temperatura a las 0 era de 15 grados.
La temperatura a las 1 era de 18 grados.
La temperatura a las 2 era de 20 grados.
La temperatura a las 3 era de 23 grados.
...
La temperatura a las 17 era de 17 grados.
La temperatura a las 18 era de 15 grados.
La temperatura a las 19 era de 14 grados.
La temperatura a las 20 era de 14 grados.
La temperatura a las 21 era de 0 grados.
La temperatura a las 22 era de 0 grados.
La temperatura a las 23 era de 0 grados.
```

Vemos que los últimos 3 elementos son nulos, que son aquellos a los que no hemos dado valores. El compilador no nos avisa que hemos metido menos datos de los reservados.

NOTA: Fíjate que para recorrer del elemento 0 al 23 (24 elementos) hacemos: for(hora=0; hora<24; hora++). La condición es que hora sea menos de 24. También podíamos haber hecho que hora != 24.

Ahora vamos a ver el caso contrario, metemos más datos de los reservados. Vamos a meter 25 en vez de 24. Si hacemos esto dependiendo del compilador obtendremos un error o al menos un warning (aviso). En unos compiladores el programa se creará y en otros no, pero aún así nos avisa del fallo. Debe indicarse que estamos intentando guardar un dato de más, no hemos reservado memoria para él.

Si la matriz debe tener una longitud determinada usamos este método de definir el número de elementos. En nuestro caso era conveniente, porque los dias siempre tienen 24 horas. Es conveniente definir el tamaño de la matriz para que nos avise si metemos más elementos de los necesarios.

En los demás casos podemos usar un método alternativo, dejar al ordenador que cuente los elementos que hemos metido y nos reserve espacio para ellos:

```
#include <stdio.h>
int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
      int temperaturas[] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
      24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };
     for ( hora=0 ; hora<24 ; hora++ )
         {
        printf( "La temperatura a las %i era de %i grados.\n", hora,
        temperaturas[hora] );
         }
     }
```

Vemos que no hemos especificado la dimensión del array temperaturas. Hemos dejado los corchetes en blanco. El ordenador contará los elementos que hemos puesto entre llaves y reservará espacio para ellos. De esta forma siempre habrá el espacio necesario,

ni más ni menos. La pega es que si ponemos más de los que queríamos no nos daremos cuenta.

Para saber en este caso cuantos elementos tiene la matriz podemos usar el operador sizeof. Dividimos el tamaño de la matriz entre el tamaño de sus elementos y tenemos el número de elementos.

```
#include <stdio.h>
int main()
     {
     int hora;
     int elementos;
     int temperaturas[] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
        24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };
     elementos = sizeof temperaturas / sizeof(int);
     for ( hora=0 ; hora<elementos ; hora++ )
          {
                printf( "La temperatura a las %i era de %i grados.\n",
                hora, temperas[hora] );
          }
     printf( "Han sido %i elementos.\n" , elementos );
     }
```

Recorrer un array

En el apartado anterior veíamos un ejemplo que mostraba todos los datos de un array. Veíamos también lo que pasaba si metíamos más o menos elementos al inicializar la matriz. Ahora vamos a ver qué pasa si intentamos imprimir más elementos de los que hay en la matriz, en este caso intentamos imprimir 28 elementos cuando sólo hay 24:

```
#include <stdio.h>

int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
      int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
      24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 13, 13, 12 };
      for (hora=0 ; hora<28 ; hora++ )
         {
            printf( "La temperatura a las %i era de %i grados.\n",
            hora, temperaturas[hora] );
         }
     }
```

Lo que se obtiene en mi ordenador es:

```
La temperatura a las 22 era de 15 grados.
...
La temperatura a las 23 era de 12 grados.
La temperatura a las 24 era de 24 grados.
La temperatura a las 25 era de 3424248 grados.
La temperatura a las 26 era de 7042 grados.
La temperatura a las 27 era de 1 grados.
```

Vemos que a partir del elemento 24 (incluído) tenemos resultados extraños. Esto es porque nos hemos salido de los límites del array e intenta acceder al elemento temperaturas[25] y sucesivos que no existen. Así que nos muestra el contenido de la memoria que está justo detrás de temperaturas[23] (esto es lo más probable) que puede ser cualquiera. Al contrario que otros lenguajes **C** no comprueba los límites de los array, nos deja saltárnoslos a la torera. Este programa no da error al compilar ni al ejecutar, tan sólo devuelve resultados extraños. Tampoco bloqueará el sistema porque no estamos escribiendo en la memoria sino leyendo de ella.

Otra cosa muy diferente es meter datos en elementos que no existen. Veamos un ejemplo **(ni se te ocurra ejecutarlo)**:

```
#include <stdio.h>
int main()
     {
     int temp[24];
     float media;
     int hora;
     for( hora=0; hora<28; hora++ )
          {
          printf( "Temperatura de las %i: ", hora );
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / 24;
     printf( "\nLa temperatura media es %f\n", media );
     }
```

Lo he probado en mi ordenador y se ha bloqueado. He tenido que apagarlo. El problema ahora es que estamos intentando escribir en el elemento temp[24] que no existe y puede ser un lugar cualquiera de la memoria. Como consecuencia de esto podemos estar cambiando algún programa o dato de la memoria que no debemos y el sistema se bloquea. Así que mucho cuidado con esto.

2.3. Vectores multidimensionales

Podemos declarar vectores de más de una dimensión muy fácilmente:

```
int a[10][5];
float b[3][2][4];
```

En este ejemplo, a es una matriz de 10 × 5 enteros y b es un vector de tres dimensiones con 3 × 2 × 4 números en coma flotante.

Puedes acceder a un elemento cualquiera de los vectores a o b utilizando tantos índices como dimensiones tiene el vector: a[4][2] y b[1][0][3], por ejemplo, son elementos de a y b, respectivamente.

La inicialización de los vectores multidimensionales necesita tantos bucles anidados como dimensiones tengan éstos:

```
int main(void)
{
  int a[10][5];
  float b[3][2][4];
  int i, j, k;

  for (i=0; i<10; i++)
    for (j=0; j<5; j++)
      a[i][j] = 0;

  for (i=0; i<3; i++)
    for (j=0; j<2; j++)
      for (k=0; k<4; k++)
        b[i][j][k] = 0.0;

  return 0;
}
```

También puedes inicializar explícitamente un vector multidimensional:

```
int c[3][3] = { {1, 0, 0},
                {0, 1, 0},
                {0, 0, 1} };
```

2.3.1. Sobre la disposición de los vectores multidimensionales en memoria

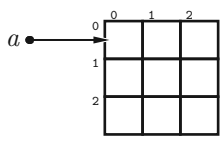
Cuando el compilador de C detecta la declaración de un vector multidimensional, reserva tantas posiciones contiguas de memoria como sea preciso para albergar todas sus celdas.

Por ejemplo, ante la declaración **int** a[3][3], C reserva 9 celdas de 4 bytes, es decir, 36 bytes. He aquí como se disponen las celdas en memoria, suponiendo que la zona de memoria asignada empieza en la dirección 1000:

996:					
1000:					a[0][0]
1004:					a[0][1]
1008:					a[0][2]
1012:					a[1][0]
1016:					a[1][1]
1020:					a[1][2]
1024:					a[2][0]
1028:					a[2][1]
1032:					a[2][2]
1036:					

Cuando accedemos a un elemento a[i][j], C sabe a qué celda de memoria acceder sumando a la dirección de a el valor (i*3+j)*4 (el 4 es el tamaño de un **int** y el 3 e sel número de columnas).

Aun siendo conscientes de cómo representa C la memoria, nosotros trabajaremos con una representación de una matriz de 3×3 como ésta:



Como puedes ver, lo relevante es que a es asimilable a un puntero a la zona de memoria en la que están dispuestos los elementos de la matriz.

EJERCICIOS

▶ **125** Este programa es incorrecto. ¿Por qué? Aun siendo incorrecto, produce cierta salida por pantalla. ¿Qué muestra?

matriz_mal.c

```c
#include <stdio.h>

#define TALLA 3

int main(void)
{
  int a[TALLA][TALLA];
  int i, j;

  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      a[i][j] = 10*i+j;

  for (j=0; j<TALLA*TALLA; j++)
    printf("%d\n", a[0][j]);

  return 0;
}
```

2.3.2. Un ejemplo: cálculo matricial

Para ilustrar el manejo de vectores multidimensionales construiremos ahora un programa que lee de teclado dos matrices de números en coma flotante y muestra por pantalla su suma y su producto. Las matrices leídas serán de 3×3 y se denominarán a y b. El resultado de la suma se almacenará en una matriz s y el del producto en otra p.

Aquí tienes el programa completo:

matrices.c

```c
#include <stdio.h>

#define TALLA 3

int main(void)
{
  float a[TALLA][TALLA], b[TALLA][TALLA];
  float s[TALLA][TALLA], p[TALLA][TALLA];
  int i, j, k;

  /* Lectura de la matriz a */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++) {
      printf("Elemento␣(%d,␣%d):␣", i, j); scanf("%f", &a[i][j]);
    }

  /* Lectura de la matriz b */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++) {
      printf("Elemento␣(%d,␣%d):␣", i, j); scanf("%f", &b[i][j]);
    }

  /* Cálculo de la suma */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      s[i][j] = a[i][j] + b[i][j];

  /* Cálculo del producto */
```

```
29    for (i=0; i<TALLA; i++)
30      for (j=0; j<TALLA; j++) {
31        p[i][j] = 0.0;
32        for (k=0; k<TALLA; k++)
33          p[i][j] += a[i][k] * b[k][j];
34      }
35
36    /* Impresión del resultado de la suma */
37    printf("Suma\n");
38    for (i=0; i<TALLA; i++) {
39      for (j=0; j<TALLA; j++)
40        printf("%8.3f", s[i][j]);
41      printf("\n");
42    }
43
44    /* Impresión del resultado del producto */
45    printf("Producto\n");
46    for (i=0; i<TALLA; i++) {
47      for (j=0; j<TALLA; j++)
48        printf("%8.3f", p[i][j]);
49      printf("\n");
50    }
51
52    return 0;
53  }
```

Aún no sabemos definir nuestras propias funciones. En el próximo capítulo volveremos a ver este programa y lo modificaremos para que use funciones definidas por nosotros.

EJERCICIOS

▶ **126** En una estación meteorológica registramos la temperatura (en grados centígrados) cada hora durante una semana. Almacenamos el resultado en una matriz de 7×24 (cada fila de la matriz contiene las 24 mediciones de un día). Diseña un programa que lea los datos por teclado y muestre:

- La máxima y mínima temperaturas de la semana.
- La máxima y mínima temperaturas de cada día.
- La temperatura media de la semana.
- La temperatura media de cada día.
- El número de días en los que la temperatura media fue superior a 30 grados.

▶ **127** Representamos diez ciudades con números del 0 al 9. Cuando hay carretera que une directamente a dos ciudades i y j, almacenamos su distancia en kilómetros en la celda `d[i][j]` de una matriz de 10×10 enteros. Si no hay carretera entre ambas ciudades, el valor almacenado en su celda de d es cero. Nos suministran un vector en el que se describe un trayecto que pasa por las 10 ciudades. Determina si se trata de un trayecto válido (las dos ciudades de todo par consecutivo están unidas por un tramo de carretera) y, en tal caso, devuelve el número de kilómetros del trayecto. Si el trayecto no es válido, indícalo con un mensaje por pantalla.

La matriz de distancias deberás inicializarla explícitamente al declararla. El vector con el recorrido de ciudades deberás leerlo de teclado.

▶ **128** Diseña un programa que lea los elementos de una matriz de 4×5 flotantes y genere un vector de talla 4 en el que cada elemento contenga el sumatorio de los elementos de cada fila. El programa debe mostrar la matriz original y el vector en este formato (evidentemente, los valores deben ser los que correspondan a lo introducido por el usuario):

```
         0      1      2      3      4       Suma
0 [ +27.33 +22.22 +10.00  +0.00 -22.22]  -> +37.33
1 [  +5.00  +0.00  -1.50  +2.50 +10.00]  -> +16.00
2 [  +3.45  +2.33  -4.56 +12.56 +12.01]  -> +25.79
3 [  +1.02  +2.22 +12.70 +34.00 +12.00]  -> +61.94
4 [  -2.00 -56.20  +3.30  +2.00  +1.00]  -> -51.90
```

El programa que hemos presentado adolece de un serio inconveniente si nuestro objetivo era construir un programa «general» para multiplicar matrices: sólo puede trabajar con matrices de `TALLA` × `TALLA`, o sea, de 3×3. ¿Y si quisiéramos trabajar con matrices de tamaños arbitrarios? El primer problema al que nos enfrentaríamos es el de que las matrices han de tener una talla máxima: no podemos, con lo que sabemos por ahora, reservar un espacio de memoria para las matrices que dependa de datos que nos suministra el usuario en tiempo de ejecución. Usaremos, pues, una constante `MAXTALLA` con un valor razonablemente grande: pongamos 10. Ello permitirá trabajar con matrices con un número de filas y columnas menor o igual que 10, aunque será a costa de malgastar memoria.

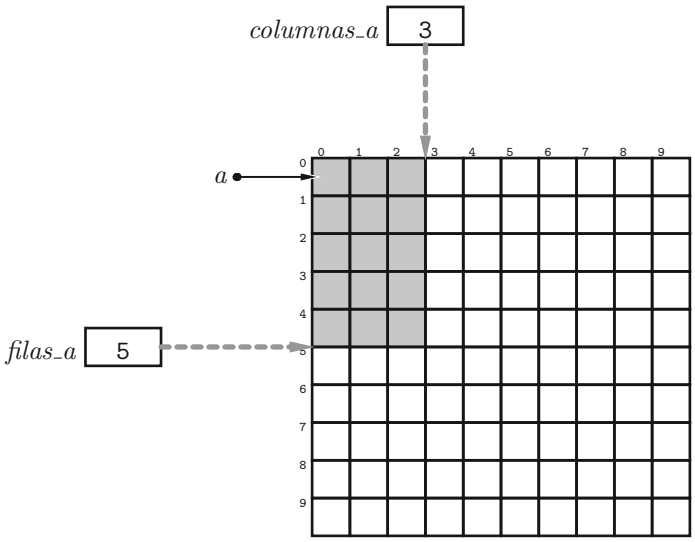
matrices.c

```
#include <stdio.h>

#define MAXTALLA 10

int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
  ...
```

El número de filas y columnas de a se pedirá al usuario y se almacenará en sendas variables: *filas_a* y *columnas_a*. Este gráfico ilustra su papel: la matriz a es de 10×10, pero sólo usamos una parte de ella (la zona sombreada) y podemos determinar qué zona es porque *filas_a* y *columnas_a* nos señalan hasta qué fila y columna llega la zona útil:



Lo mismo se aplicará al número de filas y columnas de `b`. Te mostramos el programa hasta el punto en que leemos la matriz a:

matrices.c

```
#include <stdio.h>

#define MAXTALLA 10

int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
  int filas_a, columnas_a, filas_b, columnas_b;
  int i, j, k;

  /* Lectura de la matriz a */
  printf("Filas de a:   "); scanf("%d", &filas_a);
```

```
14    printf("Columnas de a: "); scanf("%d", &columnas_a);
15
16    for (i=0; i<filas_a; i++)
17      for (j=0; j<columnas_a; j++) {
18        printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
19      }
20    ...
```

(Encárgate tú mismo de la lectura de `b`.)

La suma sólo es factible si `filas_a` es igual a `filas_b` y `columnas_a` es igual a `columnas_b`.

matrices.c

```
1  #include <stdio.h>
2
3  #define MAXTALLA 10
4
5  int main(void)
6  {
7    float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
8    float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
9    int filas_a, columnas_a, filas_b, columnas_b;
10   int filas_s, columnas_s;
11   int i, j, k;
12
13   /* Lectura de la matriz a */
14   printf("Filas de a    : "); scanf("%d", &filas_a);
15   printf("Columnas de a: "); scanf("%d", &columnas_a);
16   for (i=0; i<filas_a; i++)
17     for (j=0; j<columnas_a; j++) {
18       printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
19     }
20
21   /* Lectura de la matriz b */
22   ...
23
24   /* Cálculo de la suma */
25   if (filas_a == filas_b && columnas_a == columnas_b) {
26     filas_s = filas_a;
27     columnas_s = columnas_a;
28     for (i=0; i<filas_s; i++)
29       for (j=0; j<filas_s; j++)
30         s[i][j] = a[i][j] + b[i][j];
31   }
32
33   /* Impresión del resultado de la suma */
34   if (filas_a == filas_b && columnas_a == columnas_b) {
35     printf("Suma\n");
36     for (i=0; i<filas_s; i++) {
37       for (j=0; j<columnas_s; j++)
38         printf("%8.3f", s[i][j]);
39       printf("\n");
40     }
41   }
42   else
43     printf("Matrices no compatibles para la suma.\n");
44
45   ...
```

Recuerda que una matriz de $n \times m$ elementos se puede multiplicar por otra de $n' \times m'$ elementos sólo si m es igual a n' (o sea, el número de columnas de la primera es igual al de filas de la segunda) y que la matriz resultante es de dimensión $n \times m'$.

matrices_1.c matrices.c

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

```
2
3  #define MAXTALLA 10
4
5  int main(void)
6  {
7    float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
8    float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
9    int filas_a, columnas_a, filas_b, columnas_b;
10   int filas_s, columnas_s, filas_p, columnas_p;
11   int i, j, k;
12
13   /* Lectura de la matriz a */
14   printf("Filas de a:::: "); scanf("%d", &filas_a);
15   printf("Columnas de a: "); scanf("%d", &columnas_a);
16   for (i=0; i<filas_a; i++)
17     for (j=0; j<columnas_a; j++) {
18       printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
19     }
20
21   /* Lectura de la matriz b */
22   printf("Filas de a:::: "); scanf("%d", &filas_b);
23   printf("Columnas de a: "); scanf("%d", &columnas_b);
24   for (i=0; i<filas_b; i++)
25     for (j=0; j<columnas_b; j++) {
26       printf("Elemento (%d, %d): ", i, j); scanf("%f", &b[i][j]);
27     }
28
29   /* Cálculo de la suma */
30   if (filas_a == filas_b && columnas_a == columnas_b) {
31     filas_s = filas_a;
32     columnas_s = columnas_a;
33     for (i=0; i<filas_s; i++)
34       for (j=0; j<filas_s; j++)
35         s[i][j] = a[i][j] + b[i][j];
36   }
37
38   /* Cálculo del producto */
39   if (columnas_a == filas_b) {
40     filas_p = filas_a;
41     columnas_p = columnas_b;
42     for (i=0; i<filas_p; i++)
43       for (j=0; j<columnas_p; j++) {
44         p[i][j] = 0.0;
45         for (k=0; k<columnas_a; k++)
46           p[i][j] += a[i][k] * b[k][j];
47       }
48   }
49
50   /* Impresión del resultado de la suma */
51   if (filas_a == filas_b && columnas_a == columnas_b) {
52     printf("Suma\n");
53     for (i=0; i<filas_s; i++) {
54       for (j=0; j<columnas_s; j++)
55         printf("%8.3f", s[i][j]);
56       printf("\n");
57     }
58   }
59   else
60     printf("Matrices no compatibles para la suma.\n");
61
62   /* Impresión del resultado del producto */
63   if (columnas_a == filas_b) {
64     printf("Producto\n");
```

```
65      for (i=0; i<filas_p; i++) {
66        for (j=0; j<columnas_p; j++)
67          printf("%8.3f", p[i][j]);
68        printf("\n");
69      }
70    }
71    else
72      printf("Matrices␣no␣compatibles␣para␣el␣producto.\n");
73
74    return 0;
75  }
```

EJERCICIOS

▶ **129** Extiende el programa de calculadora matricial para efectuar las siguientes operaciones:

- Producto de una matriz por un escalar. (La matriz resultante tiene la misma dimensión que la original y cada elemento se obtiene multiplicando el escalar por la celda correspondiente de la matriz original.)
- Transpuesta de una matriz. (La transpuesta de una matriz de $n \times m$ es una matriz de $m \times n$ en la que el elemento de la fila i y columna j tiene el mismo valor que el que ocupa la celda de la fila j y columna i en la matriz original.)

▶ **130** Una matriz tiene un valle si el valor de una de sus celdas es menor que el de cualquiera de sus 8 celdas vecinas. Diseña un programa que lea una matriz (el usuario te indicará de cuántas filas y columnas) y nos diga si la matriz tiene un valle o no. En caso afirmativo, nos mostrará en pantalla las coordenadas de *todos* los valles, sus valores y el de sus celdas vecinas.

La matriz debe tener un número de filas y columnas mayor o igual que 3 y menor o igual que 10. Las casillas que no tienen 8 vecinos no se consideran candidatas a ser valle (pues no tienen 8 vecinos).

Aquí tienes un ejemplo de la salida esperada para esta matriz de 4×5:

$$\begin{pmatrix} 1 & 2 & 9 & 5 & 5 \\ 3 & 2 & 9 & 4 & 5 \\ 6 & 1 & 8 & 7 & 6 \\ 6 & 3 & 8 & 0 & 9 \end{pmatrix}$$

```
Valle en fila 2 columna 4:
9 5 5
9 4 5
8 7 6
Valle en fila 3 columna 2:
3 2 9
6 1 8
6 3 8
```

(Observa que al usuario se le muestran filas y columnas numeradas desde 1, y no desde 0.)

▶ **131** Modifica el programa del ejercicio anterior para que considere candidato a valle a cualquier celda de la matriz. Si una celda tiene menos de 8 vecinos, se considera que la celda es valle si su valor es menor que el de todos ellos.

Para la misma matriz del ejemplo del ejercicio anterior se obtendría esta salida:

```
Valle en fila 1 columna 1:
x x x
x 1 2
x 3 2
Valle en fila 2 columna 4:
9 5 5
9 4 5
8 7 6
Valle en fila 3 columna 2:
3 2 9
6 1 8
```