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

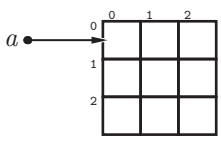
Cuando el compilador de C detecta la declaración de un vector multidimensional, reserva tantas posiciones contiguas de memoria como sea preciso para albergar todas sus celdas.

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.

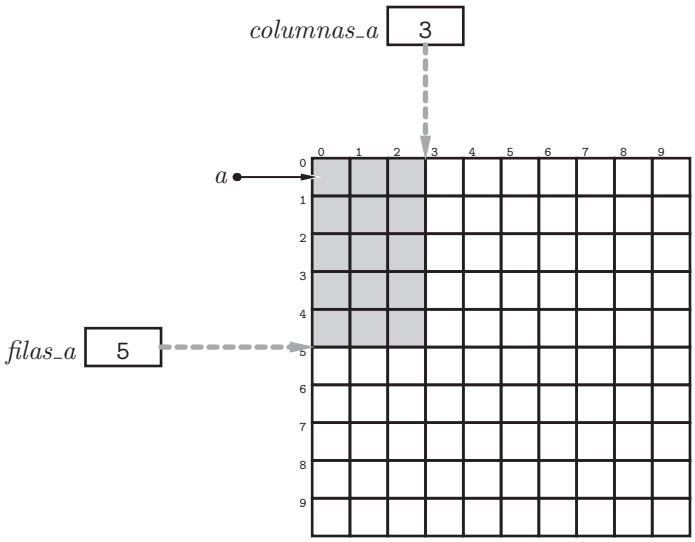
matrices.c

```
#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
```

```
21
22    return 0;
23  }
```

2.2. Cadenas estáticas

Las cadenas son un tipo de datos básico en Python, pero no en C. Las cadenas de C son vectores de caracteres (elementos de tipo **char**) con una peculiaridad: el texto de la cadena termina siempre en un carácter nulo. El carácter nulo tiene código ASCII 0 y podemos representarlo tanto con el *entero* 0 como con el *carácter* `'\0'` (recuerda que `'\0'` es una forma de escribir el valor entero 0). ¡Ojo! No confundas `'\0'` con `'0'`: el primero vale 0 y el segundo vale 48.

Las cadenas estáticas en C son, a diferencia de las cadenas Python, mutables. Eso significa que puedes modificar el contenido de una cadena durante la ejecución de un programa.

2.2.1. Declaración de cadenas

Las cadenas se declaran como vectores de caracteres, así que debes proporcionar el número máximo de caracteres que es capaz de almacenar: su *capacidad*. Esta cadena, por ejemplo, se declara con capacidad para almacenar 10 caracteres:

```
char a[10];
```

Puedes inicializar la cadena con un valor en el momento de su declaración:

```
char a[10] = "cadena";
```

Hemos declarado a como un vector de 10 caracteres y lo hemos inicializado asignándole la cadena `"cadena"`. Fíjate: hemos almacenado en a una cadena de menos de 10 caracteres. No hay problema: la *longitud* de la cadena almacenada en a es *menor* que la *capacidad* de a.

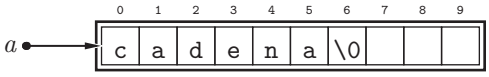
2.2.2. Representación de las cadenas en memoria

A simple vista, `"cadena"` ocupa 6 bytes, pues contamos en ella 6 caracteres, pero no es así. En realidad, `"cadena"` ocupa 7 bytes: los 6 que corresponden a los 6 caracteres que ves más uno correspondiente a *un carácter nulo al final*, que se denomina *terminador de cadena* y es invisible.

Al declarar e inicializar una cadena así:

```
char a[10] = "cadena";
```

la memoria queda de este modo:



Es decir, es como si hubiésemos inicializado la cadena de este otro modo equivalente:

```
1 char a[10] = { 'c', 'a', 'd', 'e', 'n', 'a', '\0' };
```

Recuerda, pues, que hay dos valores relacionados con el tamaño de una cadena:

- su *capacidad*, que es la talla del vector de caracteres;
- su *longitud*, que es el número de caracteres que contiene, sin contar el terminador de la cadena. La longitud de la cadena debe ser siempre *estrictamente menor* que la capacidad del vector para no desbordar la memoria reservada.

¿Y por qué toda esta complicación del terminador de cadena? Lo normal al trabajar con una variable de tipo cadena es que su longitud varíe conforme evoluciona la ejecución del programa, pero el tamaño de un vector es fijo. Por ejemplo, si ahora tenemos en a el texto `"cadena"` y más tarde decidimos guardar en ella el texto `"texto"`, que tiene un carácter menos, estaremos pasando de esta situación:

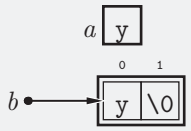
Una cadena de longitud uno no es un carácter

Hemos dicho en el capítulo anterior que una cadena de un sólo carácter, por ejemplo "*y*", no es lo mismo que un carácter, por ejemplo 'y'. Ahora puedes saber por qué: la diferencia estriba en que "*y*" ocupa dos bytes, el que corresponde al carácter 'y' y el que corresponde al carácter nulo '\0', mientras que 'y' ocupa un solo byte.

Fíjate en esta declaración de variables:

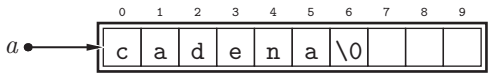
```
1 char a = 'y';
2 char b[2] = "y";
```

He aquí una representación gráfica de las variables y su contenido:

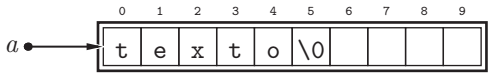


Recuerda:

- Las comillas simples definen un carácter y un carácter ocupa un solo byte.
- Las comillas dobles definen una cadena. Toda cadena incluye un carácter nulo invisible al final.

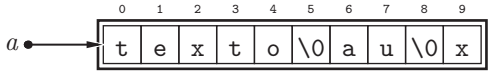


a esta otra:



Fíjate en que la zona de memoria asignada a *a* sigue siendo la misma. El «truco» del terminador ha permitido que la cadena decrezca. Podemos conseguir también que crezca a voluntad... pero siempre que no se rebase la capacidad del vector.

Hemos representado las celdas a la derecha del terminador como cajas vacías, pero no es cierto que lo estén. Lo normal es que contengan valores arbitrarios, aunque eso no importa mucho: el convenio de que la cadena termina en el primer carácter nulo hace que el resto de caracteres no se tenga en cuenta. Es posible que, en el ejemplo anterior, la memoria presente realmente este aspecto:

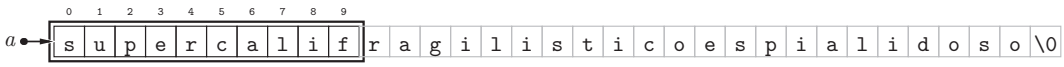


Por comodidad representaremos las celdas a la derecha del terminador con cajas vacías, pues no importa en absoluto lo que contienen.

¿Qué ocurre si intentamos inicializar una zona de memoria reservada para sólo 10 **char**s con una cadena de longitud mayor que 9?

```
char a[10] = "supercalifragilisticoespialidoso"; // ¡Mal!
```

Estaremos cometiendo un gravísimo error de programación que, posiblemente, no detecte el compilador. Los caracteres que no caben en *a* se escriben en la zona de memoria que sigue a la zona ocupada por *a*.



Ya vimos en un apartado anterior las posibles consecuencias de ocupar memoria que no nos ha sido reservada: puede que modifiques el contenido de otras variables o que trates de escribir en una zona que te está vetada, con el consiguiente aborto de la ejecución del programa.

Como resulta que en una variable con capacidad para, por ejemplo, 80 caracteres sólo caben realmente 79 caracteres aparte del nulo, adoptaremos una curiosa práctica al declarar variables de cadena que nos permitirá almacenar los 80 caracteres (además del nulo) sin crear una constante confusión con respecto al número de caracteres que caben en ellas:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1]; /* Reservamos 81 caracteres: 80 caracteres más el terminador */

  return 0;
}
```

2.2.3. Entrada/salida de cadenas

Las cadenas se muestran con *printf* y la adecuada marca de formato sin que se presenten dificultades especiales. Lo que sí resulta problemático es leer cadenas. La función *scanf* presenta una seria limitación: sólo puede leer «palabras», no «frases». Ello nos obligará a presentar una nueva función (*gets*)... que se lleva fatal con *scanf*.

Salida con *printf*

Empecemos por considerar la función *printf*, que muestra cadenas con la marca de formato `%s`. Aquí tienes un ejemplo de uso:

salida_cadena.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1] = "una cadena";

  printf("El valor de cadena es %s.\n", cadena);

  return 0;
}
```

Al ejecutar el programa obtienes en pantalla esto:

```
El valor de cadena es una cadena.
```

Puedes alterar la presentación de la cadena con modificadores:

salida_cadena_con_modificadores.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1] = "una cadena";

  printf("El valor de cadena es (%s).\n", cadena);
  printf("El valor de cadena es (%20s).\n", cadena);
  printf("El valor de cadena es (%-20s).\n", cadena);

  return 0;
}
```

```
El valor de cadena es (una cadena).
El valor de cadena es (          una cadena).
El valor de cadena es (una cadena          ).
```

¿Y si deseamos mostrar una cadena carácter a carácter? Podemos hacerlo llamando a *printf* sobre cada uno de los caracteres, pero recuerda que la marca de formato asociada a un carácter es %c:

salida_caracter_a_caracter.c

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1] = "una cadena";
  int i;

  i = 0;
  while (cadena[i] != '\0') {
    printf("%c\n", cadena[i]);
    i++;
  }

  return 0;
}
```

Este es el resultado de la ejecución:

```
u
n
a

c
a
d
e
n
a
```

Entrada con *scanf*

Poco más hay que contar acerca de *printf*. La función *scanf* es un reto mayor. He aquí un ejemplo que pretende leer e imprimir una cadena en la que podemos guardar hasta 80 caracteres (sin contar el terminador nulo):

lee_una_cadena.c

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1];

  scanf("%s", cadena);
  printf("La cadena leída es %s\n", cadena);

  return 0;
}
```

¡Ojo! ¡No hemos puesto el operador & delante de *cadena*! ¿Es un error? No. *Con las cadenas no hay que poner el carácter & del identificador al usar scanf*. ¿Por qué? Porque *scanf* espera

una dirección de memoria y el identificador, por la dualidad vector-puntero, ¡es una dirección de memoria!

Recuerda: *cadena*[0] es un **char**, pero *cadena*, sin más, es la dirección de memoria en la que empieza el vector de caracteres.

Ejecutemos el programa e introduzcamos una palabra:

```
una ↲
La cadena leída es una
```

EJERCICIOS

▶ **99** ¿Es válida esta otra forma de leer una cadena? Pruébala en tu ordenador.

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1];

  scanf("%s", &cadena[0]);
  printf("La␣cadena␣leída␣es␣%s.\n", cadena);

  return 0;
}
```

Cuando *scanf* recibe el valor asociado a *cadena*, recibe una dirección de memoria y, a partir de ella, deja los caracteres leídos de teclado. Debes tener en cuenta que si los caracteres leídos exceden la capacidad de la cadena, se producirá un error de ejecución.

¿Y por qué *printf* no muestra por pantalla una simple dirección de memoria cuando ejecutamos la llamada *printf*("La␣cadena␣leída␣es␣\%s.\n", *cadena*)? Si es cierto lo dicho, *cadena* es una dirección de memoria. La explicación es que la marca %s es interpretada por *printf* como «me pasan una dirección de memoria en la que empieza una cadena, así que he de mostrar su contenido carácter a carácter hasta encontrar un carácter nulo».

Lectura con *gets*

Hay un problema práctico con *scanf*: sólo lee una «palabra», es decir, una secuencia de caracteres no blancos. Hagamos la prueba:

lee_frase_mal.c — lee_frase_mal.c

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1];

  scanf("%s", cadena);
  printf("La␣cadena␣leída␣es␣%s.\n", cadena);

  return 0;
}
```

Si al ejecutar el programa tecleamos un par de palabras, sólo se muestra la primera:

```
una frase
La cadena leída es una.
```

¿Qué ha ocurrido con los restantes caracteres tecleados? ¡Están a la espera de ser leídos! La siguiente cadena leída, si hubiera un nuevo *scanf*, sería "frase". Si es lo que queríamos, perfecto, pero si no, el desastre puede ser mayúsculo.

¿Cómo leer, pues, una frase completa? No hay forma sencilla de hacerlo con *scanf*. Tendremos que recurrir a una función diferente. La función *gets* lee *todos* los caracteres que hay hasta encontrar un salto de línea. Dichos caracteres, excepto el salto de línea, se almacenan a partir de la dirección de memoria que se indique como argumento y se añade un terminador.

Aquí tienes un ejemplo:

```
#include <stdio.h>

#define MAXLON 11

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];

  printf("Introduce una cadena: "); gets(a);
  printf("Introduce otra cadena: "); gets(b);
  printf("La primera es %s y la segunda es %s\n", a, b);

  return 0;
}
```

Ejecutemos el programa:

```
Introduce una cadena: uno dos ↲
Introduce otra cadena: tres cuatro ↲
La primera es uno dos y la segunda es tres cuatro
```

Overflow exploit

El manejo de cadenas C es complicado... y peligroso. La posibilidad de que se almacenen más caracteres de los que caben en una zona de memoria reservada para una cadena ha dado lugar a una técnica de *cracking* muy común: el *overflow exploit* (que significa «aprovechamiento del desbordamiento»), también conocido por *smash the stack* («machacar la pila»).

Si un programa C lee una cadena con *scanf* o *gets* es vulnerable a este tipo de ataques. La idea básica es la siguiente. Si c es una variable local a una función (en el siguiente capítulo veremos cómo), reside en una zona de memoria especial: la pila. Podemos desbordar la zona de memoria reservada para la cadena c escribiendo un texto más largo del que cabe en ella. Cuando eso ocurre, estamos ocupando memoria en una zona de la pila que no nos «pertenece». Podemos conseguir así escribir información en una zona de la pila reservada a información como la dirección de retorno de la función. El *exploit* se basa en asignar a la dirección de retorno el valor de una dirección en la que habremos escrito una rutina especial en código máquina. ¿Y cómo conseguimos introducir una rutina en código máquina en un programa ajeno? ¡En la propia cadena que provoca el desbordamiento, codificándola en binario! La rutina de código máquina suele ser sencilla: efectúa una simple llamada al sistema operativo para que ejecute un intérprete de órdenes Unix. El intérprete se ejecutará con los mismos permisos que el programa que hemos reventado. Si el programa atacado se ejecutaba con permisos de `root`, habremos conseguido ejecutar un intérprete de órdenes como `root`. ¡El ordenador es nuestro!

¿Y cómo podemos proteger a nuestros programas de los *overflow exploit*? Pues, para empezar, no utilizando nunca *scanf* o *gets* directamente. Como es posible leer de teclado carácter a carácter (lo veremos en el capítulo dedicado a ficheros), podemos definir nuestra propia función de lectura de cadenas: una función de lectura que controle que nunca se escribe en una zona de memoria más información de la que cabe.

Dado que *gets* es tan vulnerable a los overflow exploit, el compilador de C te dará un aviso cuando la uses. No te sorprendas, pues, cuando veas un mensaje como éste: «`the 'gets' function is dangerous and should not be used`».

Lectura de cadenas y escalares: *gets* y *sscanf*

Y ahora, vamos con un problema al que te enfrentarás en más de una ocasión: la lectura alterna de cadenas y valores escalares. La mezcla de llamadas a *scanf* y a *gets*, produce efectos

curiosos que se derivan de la combinación de su diferente comportamiento frente a los blancos. El resultado suele ser una lectura incorrecta de los datos o incluso el bloqueo de la ejecución del programa. Los detalles son bastante escabrosos. Si tienes curiosidad, te los mostramos en el apartado B.3.

Presentaremos en este capítulo una solución directa que deberás aplicar siempre que tu programa alterne la lectura de cadenas con blancos y valores escalares (algo muy frecuente). La solución consiste en:

- Si va a leer una cadena usar *gets*.
- Y si vas a leer un valor escalar, proceder en dos pasos:
  - leer una línea completa con *gets* (usa una avariable auxiliar para ello),
  - y extraer de ella los valores escalares que se deseaba leer con ayuda de la función *sscanf*.

La función *sscanf* es similar a *scanf* (fíjate en la «s» inicial), pero no obtiene información leyéndola del teclado, sino que la extrae de una cadena.

Un ejemplo ayudará a entender el procedimiento:

lecturas.c

```
#include <stdio.h>

#define MAXLINEA 80
#define MAXFRASE 40

int main(void)
{
  int a, b;
  char frase[MAXFRASE+1];
  char linea[MAXLINEA+1];

  printf("Dame el valor de un entero:");
  gets(linea); sscanf(linea, "%d", &a);

  printf("Introduce ahora una frase:");
  gets(frase);

  printf("Y ahora, dame el valor de otro entero:");
  gets(linea); sscanf(linea, "%d", &b);

  printf("Enteros leídos: %d, %d.\n", a, b);
  printf("Frase leída: %s.\n", frase);

  return 0;
}
```

En el programa hemos definido una variable auxiliar, *linea*, que es una cadena con capacidad para 80 caracteres más el terminador (puede resultar conveniente reservar más memoria para ella en según qué aplicación). Cada vez que deseamos leer un valor escalar, leemos en *linea* un texto que introduce el usuario y obtenemos el valor escalar con la función *sscanf*. Dicha función recibe, como primer argumento, la cadena en *linea*; como segundo, una cadena con marcas de formato; y como tercer parámetro, la dirección de la variable escalar en la que queremos depositar el resultado de la lectura.

Es un proceso un tanto incómodo, pero al que tenemos que acostumbrarnos... de momento.

2.2.4. Asignación y copia de cadenas

Este programa, que pretende copiar una cadena en otra, parece correcto, pero no lo es:

```
#define MAXLON 10

int main(void)
```

```
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];

  copia = original;

  return 0;
}
```

Si compilas el programa, obtendrás un error que te impedirá obtener un ejecutable. Recuerda: los identificadores de vectores estáticos se consideran punteros *inmutables* y, a fin de cuentas, las cadenas son vectores estáticos (más adelante aprenderemos a usar vectores dinámicos). Para efectuar una copia de una cadena, has de hacerlo carácter a carácter.

```
#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];
  int i;

  for (i = 0; i <= MAXLON; i++)
    copia[i] = original[i];

  return 0;
}
```

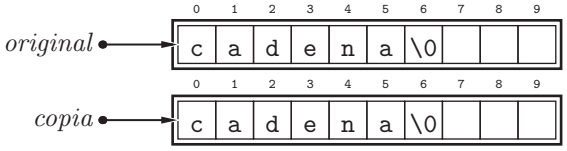
Fíjate en que el bucle recorre los 10 caracteres que realmente hay en *original* pero, de hecho, sólo necesitas copiar los caracteres que hay hasta el terminador, *incluyéndole a él*.

```
#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];
  int i;

  for (i = 0; i <= MAXLON; i++) {
    copia[i] = original[i];
    if (copia[i] == '\0')
      break;
  }

  return 0;
}
```



EJERCICIOS

▶ **100** ¿Qué problema presenta esta otra versión del mismo programa?

```
#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];
  int i;

```

```
  for (i = 0; i <= MAXLON; i++) {
    if (copia[i] == '\0')
      break;
    else
      copia[i] = original[i];
  }

  return 0;
}
```

Aún podemos hacerlo «mejor»:

```
#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];
  int i;

  for (i = 0; original[i] != '\0'; i++) {
    copia[i] = original[i];
  copia[i] = '\0';

  return 0;
}
```

¿Ves? La condición del **for** controla si hemos llegado al terminador o no. Como el terminado no llega a copiarse, lo añadimos tan pronto finaliza el bucle. Este tipo de bucles, aunque perfectamente legales, pueden resultar desconcertantes.

El copiado de cadenas es una acción frecuente, así que hay funciones predefinidas para ello, accesibles incluyendo la cabecera `string.h`:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];

  strcpy(copia, original); // Copia el contenido de original en copia.

  return 0;
}
```

Ten cuidado: *strcpy* (abreviatura de «string copy») no comprueba si el destino de la copia tiene capacidad suficiente para la cadena, así que puede provocar un desbordamiento. La función *strcpy* se limita a copiar carácter a carácter hasta llegar a un carácter nulo.

Tampoco está permitido asignar un literal de cadena a un vector de caracteres fuera de la zona de declaración de variables. Es decir, este programa es incorrecto:

```
#define MAXLON 10

int main(void)
{
  char a[MAXLON+1];

  a = "cadena"; // ¡Mal!

  return 0;
}
```

Si deseas asignar un literal de cadena, tendrás que hacerlo con la ayuda de *strcpy*:

Una versión más del copiado de cadenas

Considera esta otra versión del copiado de cadenas:

```
#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];
  int i;

  i = 0;
  while ( (copia[i] = original[i++]) != '\0') ;
  copia[i] = '\0';

  return 0;
}
```

El bucle está vacío y la condición del bucle **while** es un tanto extraña. Se aprovecha de que la asignación es una operación que devuelve un valor, así que lo puede comparar con el terminador. Y no sólo eso: el avance de i se logra con un postincremento en el mismísimo acceso al elemento de *original*. Este tipo de retruécanos es muy habitual en los programas C. Y es discutible que así sea: los programas que hacen este tipo de cosas no tienen por qué ser más rápidos y resultan más difíciles de entender (a menos que lleves mucho tiempo programando en C).

Aquí tienes una versión con una condición del bucle **while** diferente:

```
  i = 0;
  while (copia[i] = original[i++]) ;
  copia[i] = '\0';
```

¿Ves por qué funciona esta otra versión?

```
#include <string.h>

#define MAXLON 10

int main(void)
{
  char a[MAXLON+1];

  strcpy(a, "cadena");

  return 0;
}
```

EJERCICIOS

▶ **101** Diseña un programa que lea una cadena y copie en otra una versión encriptada. La encriptación convertirá cada letra (del alfabeto inglés) en la que le sigue en la tabla ASCII (excepto en el caso de las letras «z» y «Z», que serán sustituidas por «a» y «A», respectivamente.) No uses la función *strcpy*.

▶ **102** Diseña un programa que lea una cadena que posiblemente contenga letras mayúsculas y copie en otra una versión de la misma cuyas letras sean todas minúsculas. No uses la función *strcpy*.

▶ **103** Diseña un programa que lea una cadena que posiblemente contenga letras mayúsculas y copie en otra una versión de la misma cuyas letras sean todas minúsculas. Usa la función *strcpy* para obtener un duplicado de la cadena y, después, recorre la copia para ir sustituyendo en ella las letras mayúsculas por sus correspondientes minúsculas.

Copias (más) seguras

Hemos dicho que *strcpy* presenta un fallo de seguridad: no comprueba si el destino es capaz de albergar todos los caracteres de la cadena original. Si quieres asegurarte de no rebasar la capacidad del vector destino puedes usar *strncpy*, una versión de *strcpy* que copia la cadena, pero con un límite al número máximo de caracteres:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];

  strncpy(copia, original, MAXLON+1); // Copia, a lo sumo, MAXLON+1 caracteres.

  return 0;
}
```

Pero tampoco *strncpy* es perfecta. Si la cadena original tiene más caracteres de los que puede almacenar la cadena destino, la copia es imperfecta: no acabará en '\0'. De todos modos, puedes encargarte tú mismo de terminar la cadena en el último carácter, por si acaso:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
  char original[MAXLON+1] = "cadena";
  char copia[MAXLON+1];

  strncpy(copia, original, MAXLON+1);
  copia[MAXLON] = '\0';

  return 0;
}
```

2.2.5. Longitud de una cadena

El convenio de terminar una cadena con el carácter nulo permite conocer fácilmente la longitud de una cadena:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i;

  printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
  gets(a);
  i = 0;
  while (a[i] != '\0')
    i++;
  printf("Longitud de la cadena: %d\n", i);

```

```
17   return 0;
18 }
```

El estilo C

El programa que hemos presentado para calcular la longitud de una cadena es un programa C correcto, pero no es así como un programador C expresaría esa misma idea. ¡No hace falta que el bucle incluya sentencia alguna!:

```
1  #include <stdio.h>
2
3  #define MAXLON 80
4
5  int main(void)
6  {
7    char a[MAXLON+1];
8    int i;
9
10   printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
11   gets(a);
12   i = 0;
13   while (a[i++] != '\0') ; // Observa que no hay sentencia alguna en el while.
14   printf("Longitud de la cadena: %d\n", i-1);
15
16   return 0;
17 }
```

El operador de postincremento permite aumentar en uno el valor de i justo después de consultar el valor de $a[i]$. Eso sí, hemos tenido que modificar el valor mostrado como longitud, pues ahora i acaba valiendo uno más.

Es más, ni siquiera es necesario efectuar comparación alguna. El bucle se puede sustituir por este otro:

```
    i = 0;
    while (a[i++]) ;
```

El bucle funciona correctamente porque el valor '\0' significa «falso» cuando se interpreta como valor lógico. El bucle itera, pues, hasta llegar a un valor falso, es decir, a un terminador.

Algunos problemas con el operador de autoincremento

¿Qué esperamos que resulte de ejecutar esta sentencia?

```
1   int a[5] = {0, 0, 0, 0, 0};
2
3   i = 1;
4   a[i] = i++;
```

Hay dos posibles interpretaciones:

- Se evalúa primero la parte derecha de la asignación, así que i pasa a valer 2 y se asigna ese valor en $a[2]$.
- Se evalúa primero la asignación, con lo que i pasa a se asigna el valor 1 en $a[1]$ y, después, se incrementa el valor de i, que pasa a valer 2.

¿Qué hace C? No se sabe. La especificación del lenguaje estándar indica que el resultado está indefinido. Cada compilador elige qué hacer, así que ese tipo de sentencias pueden dar problemas de portabilidad. Conviene, pues, evitarlas.

Calcular la longitud de una cadena es una operación frecuentemente utilizada, así que está predefinida en la biblioteca de tratamiento de cadenas. Si incluímos la cabecera `string.h`, podemos usar la función *strlen* (abreviatura de «string length»):

while o for

Los bucles **while** pueden sustituirse muchas veces por bucles **for** equivalentes, bastante más compactos:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i;

  printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
  gets(a);
  for (i=0; a[i] != '\0'; i++) ; // Tampoco hay sentencia alguna en el for.
  printf("Longitud de la cadena: %d\n", i);

  return 0;
}
```

También aquí es superflua la comparación:

```
    for (i=0; a[i]; i++) ;
```

Todas las versiones del programa que hemos presentado son equivalentes. Escoger una u otra es cuestión de estilo.

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

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int l;

  printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
  gets(a);
  l = strlen(a);
  printf("Longitud de la cadena: %d\n", l);

  return 0;
}
```

Has de ser consciente de qué hace *strlen*: lo mismo que hacía el primer programa, es decir, recorrer la cadena de izquierda a derecha incrementando un contador hasta llegar al terminador nulo. Esto implica que tarde tanto más cuanto más larga sea la cadena. Has de estar al tanto, pues, de la fuente de ineficiencia que puede suponer utilizar directamente *strlen* en lugares críticos como los bucles. Por ejemplo, esta función cuenta las vocales minúsculas de una cadena leída por teclado:

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

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i, contador;

```

```
11    printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
12    gets(a);
13    contador = 0;
14    for (i = 0; i < strlen(a); i++)
15      if (a[i] == 'a' || a[i] == 'e' || a[i] == 'i' || a[i] == 'o' || a[i] == 'u')
16        contador++;
17    printf("Vocales minúsculas: %d\n", contador);
18
19    return 0;
20  }
```

Pero tiene un problema de eficiencia. Con cada iteración del bucle **for** se llama a *strlen* y *strlen* tarda un tiempo proporcional a la longitud de la cadena. Si la cadena tiene, pongamos, 60 caracteres, se llamará a *strlen* 60 veces para efectuar la comparación, y para cada llamada, *strlen* tardará unos 60 pasos en devolver lo mismo: el valor 60. Esta nueva versión del mismo programa no presenta ese inconveniente:

```
1   #include <stdio.h>
2   #include <string.h>
3
4   #define MAXLON 80
5
6   int main(void)
7   {
8     char a[MAXLON+1];
9     int i, longitud, contador;
10
11    printf("Introduce una cadena (máx. %d cars.): ", MAXLON);
12    gets(a);
13    longitud = strlen(cadena);
14    contador = 0;
15    for (i = 0; i < longitud; i++)
16      if (a[i] == 'a' || a[i] == 'e' || a[i] == 'i' || a[i] == 'o' || a[i] == 'u')
17        contador++;
18    printf("Vocales minúsculas: %d\n", contador);
19
20    return 0;
21  }
```

EJERCICIOS

▶ **104** Diseña un programa que lea una cadena y la invierta.

▶ **105** Diseña un programa que lea una *palabra* y determine si es o no es palíndromo.

▶ **106** Diseña un programa que lea una *frase* y determine si es o no es palíndromo. Recuerda que los espacios en blanco y los signos de puntuación no se deben tener en cuenta a la hora de determinar si la frase es palíndromo.

▶ **107** Escribe un programa C que lea dos cadenas y muestre el índice del carácter de la primera cadena en el que empieza, por primera vez, la segunda cadena. Si la segunda cadena no está contenida en la primera, el programa nos lo hará saber.

(Ejemplo: si la primera cadena es `"un ejercicio de ejemplo"` y la segunda es `"eje"`, el programa mostrará el valor 3.)

▶ **108** Escribe un programa C que lea dos cadenas y muestre el índice del carácter de la primera cadena en el que empieza por *última* vez una aparición de la segunda cadena. Si la segunda cadena no está contenida en la primera, el programa nos lo hará saber.

(Ejemplo: si la primera cadena es `"un ejercicio de ejemplo"` y la segunda es `"eje"`, el programa mostrará el valor 16.)

▶ **109** Escribe un programa que lea una línea y haga una copia de ella eliminando los espacios en blanco que haya al principio y al final de la misma.

▶ **110** Escribe un programa que lea repetidamente líneas con el nombre completo de una persona. Para cada persona, guardará temporalmente en una cadena sus iniciales (las letras con mayúsculas) separadas por puntos y espacios en blanco y mostrará el resultado en pantalla. El programa finalizará cuando el usuario escriba una línea en blanco.

▶ **111** Diseña un programa C que lea un entero n y una cadena a y muestre por pantalla el valor (en base 10) de la cadena a si se interpreta como un número en base n. El valor de n debe estar comprendido entre 2 y 16. Si la cadena a contiene un carácter que no corresponde a un dígito en base n, notificará el error y no efectuará cálculo alguno.

Ejemplos:

- si a es `"ff"` y n es 16, se mostrará el valor 255;
- si a es `"f0"` y n es 15, se notificará un error: «`f no es un dígito en base 15`»;
- si a es `"1111"` y n es 2, se mostrará el valor 15.

▶ **112** Diseña un programa C que lea una línea y muestre por pantalla el número de palabras que hay en ella.

2.2.6. Concatenación

Python permitía concatenar cadenas con el operador `+`. En C no puedes usar `+` para concatenar cadenas. Una posibilidad es que las concatenes tú mismo «a mano», con bucles. Este programa, por ejemplo, pide dos cadenas y concatena la segunda a la primera:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];
  int longa, longb;
  int i;

  printf("Introduce un texto (máx. %d cars.): ", MAXLON);  gets(a);
  printf("Introduce otro texto (máx. %d cars.): ", MAXLON); gets(b);

  longa = strlen(a);
  longb = strlen(b);
  for (i=0; i<longb; i++)
    a[longa+i] = b[i];
  a[longa+longb] = '\0';
  printf("Concatenación de ambos: %s", a);

  return 0;
}
```

Pero es mejor usar la función de librería *strcat* (por «string concatenate»):

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

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];

  printf("Introduce un texto (máx. %d cars.): ", MAXLON);
  gets(a);
  printf("Introduce otro texto (máx. %d cars.): ", MAXLON);
  gets(b);
  strcat(a, b);  // Equivale a la asignación Python a = a + b
  printf("Concatenación de ambos: %s", a);

  return 0;
}
```

Si quieres dejar el resultado de la concatenación en una variable distinta, deberás actuar en dos pasos:

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

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1], c[MAXLON+1];

  printf("Introduce un texto (máx. %d cars.): ", MAXLON);
  gets(a);
  printf("Introduce otro texto (máx. %d cars.): ", MAXLON);
  gets(b);
  strcpy(c, a);  // Ésta seguida de...
  strcat(c, b);  // ... ésta equivale a la sentencia Python c = a + b
  printf("Concatenación de ambos: %s", c);

  return 0;
}
```

Recuerda que es responsabilidad del programador asegurarse de que la cadena que recibe la concatenación dispone de capacidad suficiente para almacenar la cadena resultante.

Por cierto, el operador de repetición de cadenas que encontrábamos en Python (operador `*`) no está disponible en C ni hay función predefinida que lo proporcione.

Un carácter no es una cadena

Un error frecuente es intentar añadir un carácter a una cadena con *strcat* o asignárselo como único carácter con *strcpy*:

```
char linea[10] = "cadena";
char caracter = 's';

strcat(linea, caracter); // ¡Mal!
strcpy(linea, 'x'); // ¡Mal!
```

Recuerda: los dos datos de *strcat* y *strcpy* han de ser *cadenas* y no es aceptable que uno de ellos sea un *carácter*.

EJERCICIOS

▶ **113** Escribe un programa C que lea el nombre y los dos apellidos de una persona en tres cadenas. A continuación, el programa formará una sóla cadena en la que aparezcan el nombre y los apellidos separados por espacios en blanco.

▶ **114** Escribe un programa C que lea un verbo regular de la primera conjugación y lo muestre por pantalla conjugado en presente de indicativo. Por ejemplo, si lee el texto `programar`, mostrará por pantalla:

```
yo programo
tú programas
él programa
nosotros programamos
vosotros programáis
ellos programan
```

2.2.7. Comparación de cadenas

Tampoco los operadores de comparación (`==`, `!=`, `<`, `<=`, `>`, `>=`) funcionan con cadenas. Existe, no obstante, una función de `string.h` que permite paliar esta carencia de C: *strcmp* (abreviatura

de «string comparison»). La función *strcmp* recibe dos cadenas, *a* y *b*, y devuelve un entero. El entero que resulta de efectuar la llamada *strcmp*`(`*a*`, `*b*`)` codifica el resultado de la comparación:

- es menor que cero si la cadena *a* es menor que *b*,
- es 0 si la cadena *a* es igual que *b*, y
- es mayor que cero si la cadena *a* es mayor que *b*.

Naturalmente, menor significa que va delante en orden alfabético, y mayor que va detrás.

EJERCICIOS

▶ **115** Diseña un programa C que lea dos cadenas y, si la primera es *menor o igual* que la segunda, imprima el texto «menor o igual».

▶ **116** ¿Qué valor devolverá la llamada *strcmp*`("21", "112")`?

▶ **117** Escribe un programa que lea dos cadenas, *a* y *b* (con capacidad para 80 caracteres), y muestre por pantalla -1 si *a* es menor que *b*, 0 si *a* es igual que *b*, y 1 si *a* es mayor que *b*. Está prohibido que utilices la función *strcmp*.

2.2.8. Funciones útiles para manejar caracteres

No sólo `string.h` contiene funciones útiles para el tratamiento de cadenas. En `ctype.h` encontrarás unas funciones que permiten hacer cómodamente preguntas acerca de los caracteres, como si son mayúsculas, minúsculas, dígitos, etc:

- *isalnum*`(`*carácter*`)`: devuelve cierto (un entero cualquiera distinto de cero) si *carácter* es una letra o dígito, y falso (el valor entero 0) en caso contrario,
- *isalpha*`(`*carácter*`)`: devuelve cierto si *carácter* es una letra, y falso en caso contrario,
- *isblank*`(`*carácter*`)`: devuelve cierto si *carácter* es un espacio en blanco o un tabulador,
- *isdigit*`(`*carácter*`)` devuelve cierto si *carácter* es un dígito, y falso en caso contrario,
- *isspace*`(`*carácter*`)`: devuelve cierto si *carácter* es un espacio en blanco, un salto de línea, un retorno de carro, un tabulador, etc., y falso en caso contrario,
- *islower*`(`*carácter*`)`: devuelve cierto si *carácter* es una letra minúscula, y falso en caso contrario,
- *isupper*`(`*carácter*`)`: devuelve cierto si *carácter* es una letra mayúscula, y falso en caso contrario.

También en `ctype.h` encontrarás un par de funciones útiles para convertir caracteres de minúscula a mayúscula y viceversa:

- *toupper*`(`*carácter*`)`: devuelve la mayúscula asociada a *carácter*, si la tiene; si no, devuelve el mismo carácter,
- *tolower*`(`*carácter*`)`: devuelve la minúscula asociada a *carácter*, si la tiene; si no, devuelve el mismo carácter.

EJERCICIOS

▶ **118** ¿Qué problema presenta este programa?

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 int main(void)
5 {
6   char b[2] = "a";
7
8   if (isalpha(b))
```