Capítulo VII

Las Matrices[1]

Las variables que hemos utilizado hasta ahora eran lo que se denomina variables escalares (o simples): en un momento dado, una variable de este tipo contiene un solo valor.

Corno hemos dicho ya en el primer capítulo, la noción de estructura de datos permite a un lenguaje evolucionado manipular variables más elaboradas que permiten dar un nombre, no ya a un solo valor, sino a un conjunto de valores. La estructura de datos más extendida, y presente en todos los lenguajes, es la **matriz.** Veremos que se distingue:

- la matriz de **una dimensión** (se llama también «de un índice»): se trata de una lista ordenada de valores del mismo tipo, designada por un nombre único, siendo recuperado cada valor de la lista por un «número de orden» que se denomina índice; la matriz de una dimensión es próxima, de hecho, de la noción matemática de vector;
- la matriz de **dos dimensiones** (de «dos índices»): es la más próxima a la idea usual que tenemos de «matriz» o «tabla», es decir, un conjunto de líneas y columnas; en esta ocasión, cada valor de la matriz se recupera por medio de dos índices;
- la matriz de más de dos dimensiones; no se trata en realidad más que de una generalización del caso anterior.
- Vamos a empezar de manera natural por estudiar en detalle las matrices de una dimensión, después de haber mostrado cómo la noción de matriz podría ser prácticamente indispensable en la realización de un programa.

1. La noción de matriz de una dimensión

1.1 Cuando la noción de variable se muestra insuficiente

Supongamos que queremos escribir un programa que lee 5 valores antes de visualizar los diferentes cuadrados; el «diálogo con el usuario» presentaría este aspecto:

```
Introduzca 5 números enteros:

11  9  14  25  63

NÚMERO     CUADRADO
11              121
 9               81
14              196
25              625
63             3969
```

[1] DELANNOY, Claude (1994); El libro de C primer lenguaje *1ª Edición;* Barcelona; Ediciones Gestión 2000, S. A.; página 135.

Observe que el programa debe conservar los cinco valores proporcionados antes de comenzar a escribir los resultados correspondientes al primero. En este caso, no es posible ya utilizar una sola variable a la que se asignaría sucesivamente cada uno de los diferentes valores entrados (¡método que serviría, sin embargo si sólo se tratara de calcular la suma!). Desde luego, puede pensar, podemos utilizar tantas variables diferentes como valores a leer, por ejemplo, aquí: A, By C, D y E o también V7, V2y V3, V4 y V5. Este método presentaría sin embargo numerosos inconvenientes:

• sería necesario encontrar un nombre de variable para cada valor; ¡esto es manejable para 5 valores, pero corre el riesgo de convertirse en algo ingobernable con 100 o 1.000 valores!
no existiría ninguna relación entre las diferentes variables; por ejemplo, haría falta introducir tantas instrucciones de lectura diferentes como valores a leer; lo mismo se puede decir para la escritura de los resultados.

1.2 La solución: la matriz

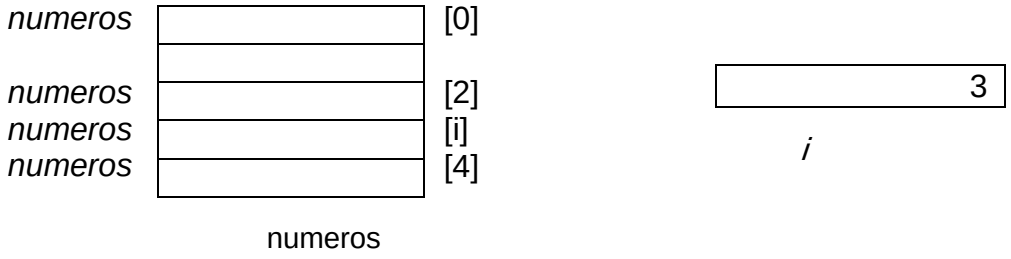
En C, como en la mayor parte de lenguajes, vamos a poder utilizar una matriz (aquí, de una dimensión), es decir:

• atribuir un solo nombre al conjunto de nuestros 5 valores, por ejemplo números;

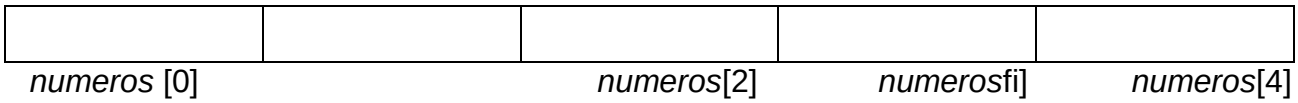
• recuperar cada valor por su nombre seguido, entre corchetes ([...]), por un valor comprendido entre 0 y 4 que se denomina un índice; **atención, en lenguaje C, el primer valor de un índice es 0 y no 1.**

Así, *numeros*[0] designa el primer valor de la matriz *numeros*, *numeros*[2] designa la tercera, *numeros*[4] la quinta... Más generalmente, si *i* es una variable numérica de tipo *int*, cuyo valor está comprendido entre 0 y 4, podremos hablar de *numeros*[i].

Veamos un esquema que ilustra lo que acabamos de decir:



Evidentemente, esta representación «vertical» es de hecho arbitraria. Habríamos podido perfectamente realizar este esquema:



De manera general, en lenguaje C, un índice puede ser cualquier expresión aritmética entera (y no simplemente una variable o una constante como en nuestros ejemplos). Así, las notaciones siguientes designan elementos de la matriz *numeros* (suponiendo que *i* y *j* son enteros):

numeros [*i* + 2] *numeros*[2 * *i* + *j*]

La condición sin embargo de que los valores de las expresiones mencionadas como índice estén comprendidas (aquí) entre 0 y 4.

Hay que observar que en lenguaje C cada valor de una matriz es recuperado por medio de un índice que es un número. En la vida normal, utilizamos a menudo otros métodos para recuperar un valor. Así, ante una matriz de 20 notas de alumnos, un profesor preferirá hablar de la nota de Pedro más que de la nota de rango 3.

2. Cómo utilizar una matriz de una dimensión

2.1 Asignarle el espacio y precisar su tipo

Como para una variable, el compilador debe estar capacitado para reservar el espacio de una matriz y debe conocer el tipo de sus diferentes elementos. Estas informaciones le serán proporcionadas por una instrucción de declaración. Así, la matriz *numeros* de la que hemos hablado en el apartado anterior será declarada como:

```
int  numeros[5];    /* dejar un espacio antes del [5] es facultativo */
```

Observe el orden en el que se proporcionan las informaciones: tipo, nombre y número de elementos. Además, no confunda la notación de un elemento de matriz *numeros[5]* con la misma notación que aparece en la declaración de la matriz y que, en esta ocasión, indica su longitud.

Se pueden mencionar varias matrices del mismo tipo y, eventualmente, de longitud diferente en una misma instrucción de declaración. Por ejemplo, la instrucción:

```
int  res [100] ,  x [20] ;
```

Reservaría los espacios de dos matrices de enteros, la primera llamada *res* con 100 elementos y la segunda, llamada *x,* con 20 elementos.

Importante

La longitud de una matriz, es decir, el espacio de memoria necesario, depende a la vez de su tipo (es decir, del tipo de sus elementos) y de su dimensión. En lenguaje C (como en buena parte de los demás lenguajes), esta longitud debe ser conocida en el momento de la compilación, es decir, al fin y al cabo, en el momento en que se escribe el programa. Es pues imposible disponer de matrices cuyo número de elementos sea determinado durante la ejecución del programa, y menos aún que este número evolucione a lo largo de la ejecución. Esta necesidad puede sin embargo ser satisfecha utilizando las técnicas llamadas de «gestión dinámica de la memoria», que quedan fuera del alcance de esta obra.

2.2 Manipular los elementos de una matriz

Un elemento de matriz (se habla también de variable indexada) **se emplea exactamente como una variable de tipo idéntico.** Como todas las matrices que =x-entraremos en este capítulo tendrán elementos de un tipo «escalar» (entero, coma flotante, carácter), observará que un elemento de matriz podrá por ejemplo:

- ser objeto de una asignación
- figurar en una expresión aritmética
- figurar en la lista de una instrucción de lectura o de escritura.

Los apartados siguientes mostrarán numerosos ejemplos de aprendizaje, cuyo único objetivo es hacer que se familiarice con la manipulación de las matrices.

Encontraremos más adelante, en el capítulo dedicado a las estructuras, matrices cuyos elementos no serán simplemente escalares, sino «estructuras»; en este caso, solamente la primera de las tres posibilidades citadas anteriormente (asignación) será utilizable.

2-3 Asignación de valores a una matriz

Con esta declaración: int x [4];

Las instrucciones:

```
x[0] = 12;    /* se puede escribir x[0], x[0], x[0]... */

x[1] = 5;

x[2] = 8;

x[3] = 20;  /* observe que el último de los 4 elementos de x tiene el número 3 */
```

Colocan respectivamente, los valores 12, 5, 8 y 20 en cada uno de los elementos de la matriz x, lo que puede esquematizarse por ejemplo así:

12	5	8	20

x

De igual modo, con estas instrucciones:

```
char voc[5];
….
voc[0] = 'a';
voc [ 1 ] = ' e';
voc[2] = 'i';
voc[3] = 'o';
voc [ 4 ] = ' u';
```

Se obtiene en cada uno de los elementos de la matriz *voc,* los caracteres correspondientes a las 5 vocales del alfabeto, que se puede esquematizar así:

a	*e*	*i*	*o*	*U*

voc

Si se quiere colocar el mismo valor, por ejemplo 1, en cada uno de los elementos de la matriz anterior, es inútil utilizar 4 instrucciones de asignación distintas, como en:

```
x[0] = 1;
x[1] = 1;
x[2] = 1;
x[3] = 1;
```

Basta, en efecto, con utilizar una repetición incondicional que utilice un contador (llamado, por ejemplo, *i),* cuyo valor progrese de 0 a 3:

```
for (i=0; i<4; i=i+l)
  x[i] = 1;
```

En este sentido, no. confunda el índice que sirve para recuperar un elemento de la matriz con el valor de dicho elemento; dicho de otro modo, no confunda *i* conx[/].

Importante

Por lo que hemos visto, podemos asignar un valor a un elemento cualquiera de una matriz. Sin embargo, no existe en lenguaje C una instrucción que opere directamente sobre todos los valores de una matriz; por ejemplo, hemos visto que, para atribuir el mismo valor a todos los elementos de una matriz, era necesario repetir una instrucción de asignación de este valor a un elemento de rango *i.* Del mismo modo, si *ti* y *t2* son dos matrices del mismo tipo y de la misma longitud:

```
int ti[100], t2[100];
```

La única forma de copiar todos los valores de í2 en *ti* será recurrir a una repetición del tipo:

```
for (i=0; i<100; i=i+l) tI[i] = t2[i];
```

Una instrucción del tipo *t2 = ti* no tendría sentido y, de todos modos, sería rechazada por el compilador.

2.4 Lectura de elementos de una matriz

Si se ha declarado una matriz *x* como: `int x[4];`

Se puede leer un valor entero para su primer elemento por medio de:

```
scanf ("%d", &x[0]);
```

Recordemos que &Jt[0] significa «dirección» de x[0], es decir, la dirección del primer elemento (de tipo *int*) de la matriz *x.* Es inútil utilizar paréntesis escribiendo por ejemplo &(jx[0]); hacerlo no constituiría, sin embargo, un error.

Del mismo modo:

```
scanf ("%d%d", &x[l], &x[3]);
```

Leerá dos valores enteros que serán asignados al segundo elemento (recuperado por el índice 1 y no 2 porque, no lo olvide, el primer elemento de una matriz se recupera con el índice 0, y no 1) y al cuarto elemento de la matriz *x.*

Evidentemente, será posible leer valores para cada uno de los elementos de *x* utilizando una repetición apropiada: `for (i=0; i<4; i=i+l)`

```
    scanf ("%d", &x[i]); /* leer el elemento de índice i */
```

Observe que, con estas instrucciones, debido a las reglas de lectura de *scanf,* el usuario podrá presentar su respuesta como quiera: a razón de un valor por línea, colocando los 4 valores en una misma línea, proporcionando bien uno, bien dos valores por línea,...

Por el contrario, esta libertad no existirá si se desea guiar al usuario, indicándole, por ejemplo, un número para cada valor, como en:

```
for (i=0; i<4; i=i+1)
{       printf ("introduzca el valor número %d: ", i);
        scanf ("%d", &x[i]);
}
```

Más exactamente, el usuario podrá siempre proporcionar más de un valor por línea, pero el diálogo será algo desconcertante, como en este ejemplo:

```
Introduzca el valor  número  0:  5  12
Introduzca el valor  número   1:  introduzca el valor
número 2: 25
Introduzca el valor  número       3:      45
```

A título indicativo, obtendríamos en los 4 elementos de x, respectivamente valores 5, 12, 25 y 45.

La observación precedente sobre la manipulación global de matrices, a nivel de la asignación, se aplica también a la lectura. No existe, en lenguaje C, ninguna instrucción que permita leer directamente el conjunto de los valores de una matriz. Sin embargo, una instrucción del tipo:

```
scanf ("%d", &t); /* equivalente, en realidad, a scanf ("%d",&t[0]); */
```

No sería rechazada por el compilador. En efecto, en C la dirección de una matriz no es en realidad más que la dirección de su primer elemento; es más, veremos que la simple notación 4 designa la propia dirección, de modo que la instrucción siguiente es acepta;

```
scanf ("%d", t); /* equivalente a scanf ("%d", &t[0]); */
```

2.5 Escritura de elementos en una matriz

Una vez más, basta con aplicar a un elemento de matriz (variable indexada) lo que ya sabemos hacer con una variable de tipo idéntico. Veamos un ejemplo simple:

```
in()
 {
int tab[6];
int i;
tab[0]= 0;            /* 0 para el primer elemento */
 for (i=1; i<5; i=i+1)          /* i<5 o, si se prefiere, i<=4 */
 tab[i]= 1;          /* 1 en los elementos de índice 1 a 4 */
 tab[5]= 2;          /* 2 en el último elemento */
 for (i=0; i<6; i=i+1)
 printf("%d ", tab[i]);          /* escribimos cada uno de los elementos con el
                                    mismo formato */
}
```

```
0   1 1 1 1 2
```

Evidentemente, la repetición de una única instrucción de escritura no es utilizable más que si se acepta escribir todos los elementos de la matriz con el mismo formato, lo que es generalmente el caso. Observe que aquí hemos previsto un espacio después del código %d; si no fuera éste el caso, los valores serían escritos sin espacio de separación, lo que no sería demasiado legible:

```
011112
```

Si se desea obtener un valor por línea, bastará con reemplazar la instrucción de escritura por:

```
printf ("%d\n", tab[i]);
```

2.6 Atención a los «desbordamientos de índices»

Supongamos que ha declarado esta matriz:

```
int t[15]
```

Y que quiere utilizar el elemento t[20] o incluso el elemento t[i] con valores de i incorrectos (es decir, aquí, no comprendidos entre 0 y 14).

En el primer caso (t[20]), se podría esperar que el compilador señale la anómala; en la práctica, raramente será el caso.
En el segundo caso, hay que saber que con la notación t[í] el compilador hace corresponder en realidad un cálculo de dirección del elemento de rango i de la matriz t; este cálculo hace intervenir a la vez a la dirección de principio de la matriz, la longitud de cada uno de sus elementos y el valor de *i* (observe que el compilador no puede conocer efectivamente esta dirección, la cual, por lo demás, puede ser diferente cada vez, según el valor efectivo de *i*).

Para los valores incorrectos de *i*, este cálculo devolverá siempre una dirección, pero simplemente esta última será «exterior» a la matriz. Las consecuencias pueden entonces ser diversas:

- valor de t[i] imprevisible (en el caso en que se quiera simplemente utilizar este valor f[*i*]);
- destrucción de un valor en un emplazamiento cualquiera, en el caso en que se quiera asignar un valor a t[i].

Se observa pues que es absolutamente necesario introducir uno mismo en el interior de un programa las instrucciones para verificar la coherencia de un índice, por cuanto existe el riesgo, por ejemplo si este índice resulta de valores proporcionados por teclado.

3. Ejemplo de utilización de una matriz de una dimensión

Veamos ahora el programa que permite resolver el problema presentado en el apartado 1.1:

```
main()
{ int numeros[5];      /* para conservar los 5 números proporcionados */
   int cuadrado;       /* para el cálculo del cuadrado de un número */
   int i;
        /* lectura de los 5 números en la matriz números */
  printf ("introduzca 5 números enteros (presentación libre):\n");
  for (i=0; i<5; i=i+l)
        scanf ("%d", &numeros[i]);
         /* visualización de los 5 números y de sus cuadrados */
printf ("NÚMERO CUADRADO\n");
for (i=0; i<5; i=i+l)
        {cuadrado = números[i] * numeros[i];
          printf ("%5d %6d\n", numeros[i], cuadrado);
        }
}
```

```
        introduzca 5 números enteros (presentación libre):
        11 9
        14 25 63
```

NÚMERO	CUADRADO
11	121
9	81
14	196
25	625

◇ Se podría sustituir el uso de la variable ***cuadrado*** reemplazando la última instrucción ***printf*** por:

```
printf ("%5d %5d", números[i], números[i] * números[i]);
```

- ***Ejercicio VII. 1***

Qué devolverá la ejecución de este programa:

```
main()
 {
int val[6];
int k;
val[0] = 1;
for (k=1; k<6; k=k+1)
val[k] = val[k-1] + 2;
for (k=1; k<6; k=k+1)
printf ("%d", val[k]);
```

- ***Ejercicio VII.2***

Qué devolverá la ejecución de este programa:

```
main()
   {       int serie[8];
           int i;
           serie[0]=1;
           serie[1]=1;
           for (i=2; i<8; i=i+1)
                   serie[i] = serie [i-1] + serie [i-2];
           for (i=0; i<8; i=i+1)
                   printf ("%d\n", serie[i]);
   }
```

4. Algunas técnicas clásicas aplicadas a las matrices de una dimensión

En el capítulo anterior, hemos aprendido a calcular la suma o el máximo de varios valores leídos por teclado. Las técnicas utilizadas pueden aplicarse sin dificultad al caso en que los valores en cuestión sean elementos de una matriz.

Siendo ***t*** una matriz de 200 enteros reservada por la declaración:

```
int t[200];
```

Las instrucciones siguientes calculan la **suma** en la variable entera llamada **suma** (la variable ***i*** se supone de tipo ***int):***

```
suma = 0;

for (i=0; i<200; i=i+1) suma = suma + t[i];
```

Las instrucciones siguientes permiten obtener, en la variable llamada ***max.*** supuesta de tipo ***int,*** **el mayor valor** de esta misma matriz ***t***:

```
max = t[0];

for (i=1; i<200; i=i+1)  /* atención se empieza por i=1 y no i=0 */

if (t[ i ] > max) max = t[ i ];
```

- ***Ejercicio VII.3***

  A partir de la matriz *t* anterior, escriba las instrucciones que permitan determinar la posición de su mayor elemento, es decir el valor del índice correspondiente.

5. Ordenación de una matriz de una dimensión

La utilización de una matriz permite resolver un problema bastante frecuente: ordenar, de manera creciente por ejemplo, una serie de valores.

En todos los casos, se empieza por colocar los valores en cuestión en una matriz. Después, se realiza una **ordenación** de los valores de esta matriz. Existen varias técnicas a este efecto: la más simple se denomina «ordenación por extracción simple»; se define así (representando *n* el número de elementos de la matriz):

• se busca el elemento más pequeño de los *n* elementos de la matriz;

• se intercambia este elemento con el primer elemento de la matriz;

• el elemento más pequeño se encuentra entonces en primera posición. Se puede entonces aplicar las dos operaciones anteriores a los *n* - 1 elementos restantes, después a los *n* - 2,... y ello hasta que no quede más que un elemento (el último) que es entonces el mayor.

Veamos un programa completo que aplica esta técnica a 15 valores enteros leídos por teclado:

```
main()
{ int t[15]; /* para los 15 valores a ordenar */
int i, j;
int temp; /* para proceder al intercambio de los dos valores */
     /* lectura de los valores a ordenar */
printf( "introduzca 15 valores enteros (presentación libre): \n");
for (i=0; i<15; i=i+1)
 scanf ("%d", &t[i]);
   /* ordenación de los valores */
for (i=0; i<14; i=i+1)        /* pasa por todos los elementos menos el último */
for (j=i+1; j<15; j=j+1)      /* para comparar t[ i ] con todos los siguientes */
if (t[  i ] > t[ j ] )
 {
    temp = t[i];       /* intercambio de t [ i ] */
     t [ i ] = t [ j ];    /* y t [ j ] si es */
     t [ j ] = temp;     /* necesario */
}
/* Visualización de los valores ordenados */
 printf ("Estos son sus valores por orden creciente:\n");
 for (i=0; i<15; i=i+1)
 printf("%d", t [ i ] );
}
```

```
Introduzca 15 valores enteros (presentación libre):
12 8 9 -5 0 15 3 21 8 8 4 -7 7 10 9
Estos son sus valores por orden creciente:
-7 -5 0 3 4 7 8 8 8 9 9 10 12 15 21
```

- ***Ejercicio VII.4***

  Escriba un programa que lea 25 notas y que indique cuántas son superiores a la media (¡es decir, a la media de dichas 25 notas!)

6. Inicialización de matrices de una dimensión

Hemos visto ya cómo era posible inicializar una variable en el momento de su declaración. Esta posibilidad se aplica también a las matrices de una dimensión. Por ejemplo, la declaración:

```
int tab[5] = {10, 20, 5, 0, 3};
```

Coloca los valores 10, 20, 5, 0 y 3 en cada uno de los 5 elementos de la matriz tab.

Del mismo modo:

```
char voc[5] = {'a', 'e', 'i', 'o', 'u');
```

Coloca los 6 caracteres correspondientes a nuestras vocales en la matriz del tipo char, llamada voc.

Es posible mencionar entre las llaves sólo ciertos valores, como en estos ejemplos:

int tab[5] = {10, 20 };

int tab[5] = { ,,5,,3};

Observe sin embargo que, en este caso, los valores que faltan serán imprevisibles.

Ejemplo: conteo de vocales de una serie de caracteres

Vamos a realizar un programa que lee una serie de caracteres (supuestamente terminada con una validación, y por tanto con un carácter de «fin de línea») y que contabiliza el número de veces en que cada una de las 5 vocales aparece en ella como en este ejemplo de ejecución:

Introduzca un texto de su elección:

Al programar, se puede encontrar con la necesidad de realizar recuentos distintos de un bucle

Su texto incluye:

8 veces la letra a

12 veces la letra e

4 veces la letra i

5 veces la letra o

4 veces la letra u

Para ello, vamos a utilizar dos matrices de la misma longitud:

• una que contendrá las diferentes vocales:

char voc[5] = { 'a', 'e', 'i', 'o', ‘u’};

• una segunda que servirá para contar el número de cada una de las vocales:

int cuenta [5];

Así, planearemos las cosas para obtener, en cuenta[0], el número de veces que aparece la vocal voc[0], en *cuenta*[1], el número de veces que aparece la vocal voc[1]...

En definitiva, observe que nos basta con repetir, para cada carácter leído (por ejemplo en la variable c) el tratamiento siguiente: comparar *c* con cada una de las 5 vocales de la matriz voc; si aparece una igualdad, se aumenta en uno el elemento correspondiente de la matriz *cuenta*.

Veamos el programa:

```
main()
 {
      char voc[5] = { ‘a’, ‘e’, ‘i’, ‘o’, ‘u’};          /* las 5 vocales */
      int cuenta [5];                /* los contadores correspondientes */
      int i;          /* contador de repetición */
      char c;       /* para un carácter leido por teclado */
```

```
for (i=0; i<5; i=i+l)
        cuenta[i] =0;           /* inicializa contadores */
printf ("introduzca un texto de su elección:\n");
do
 {
      scanf ("%c", &c);
       for (i=0; i<5; i=i+l)
                if (c == voc[i])
                cuentafi] = cuentafi] + 1;
}
while (c!='\n');
printf ("su texto incluye:\n");
for (i=0; i<5; i=i+1)
printf ("%d  veces la letra %c\n", cuentaf i ], voc[ i ]);
}
```

Incluso cuando se ha encontrado correspondiente a una vocal, se sigue el bucle de comparación con las demás vocales; sería posible evitarlo, pero a costa de una ligera complicación del programa.

Las cadenas de caracteres

Una cadena de caracteres es una serie de caracteres cualesquiera. Hasta ahora, hemos utilizado estas cadenas en los formatos de las instrucciones scanf o printf. Por ejemplo, la notación «n = %d» correspondía a una cadena formada por 6 caracteres (n, espacio, =, espacio, % y d); igualmente, la notación «buenos días\n» correspondía a una cadena formada por 12 caracteres (b, u, e, n, o, s, ,d, í, a, s y fin de línea).

Pero las cadenas que hemos utilizado bajo esta forma podrían ser calificadas de «constantes»; efectivamente, la serie de caracteres que las componen está perrectamente definida y no es susceptible de cambiar.

En ciertas situaciones, sería preferible disponer de variables capaces de acoger cadenas de caracteres susceptibles de evolucionar (a la vez en contenido y en número de caracteres) a lo largo de la ejecución del programa. Ciertos lenguajes disponen para ello de un «tipo cadena».

El lenguaje C, por su parte, no incluye un verdadero tipo de cadena; ciertamente, con lo que hemos estudiado hasta ahora, es posible colocar una serie de

caracteres en una matriz de caracteres (elementos de tipo char). Sin embargo, aparecen algunas restricciones:

- no sabemos cómo gestionar cadenas cuya longitud sea diferente de la dimensión de la matriz;
- la lectura o la escritura de los caracteres de nuestra matriz debería hacer carácter a carácter y no globalmente.

En realidad, el lenguaje C ofrece algunas posibilidades que facilitan la manipulación de tales cadenas, por poco que se respete una convención particular parí su representación, a saber, «marcar» el final con un carácter especial de código: \0. En este caso, se puede:

- leer o escribir globalmente una cadena, utilizando, bien las funciones scanf : printf y un código de formato apropiado, o bien con nuevas funciones específicas (gets y puts);
- realizar tratamientos propios de cadenas, tales como las comparaciones, k copia, la concatenación, utilizando funciones previstas a este efecto.

Veremos que la notación correspondiente a lo que hemos llamado una «constante de cadena» conduce al compilador a crear una serie de caracteres que respetan la convención citada; será así posible emplear como parámetro funciones de manipulación de cadenas y, por tanto, disponer en cierto modo de constantes de un «falso tipo cadena».

Finalmente, aprenderemos cómo manipular matrices de cadenas.

1 • Cómo leer o escribir cadenas

1.1 Con las funciones usuales *scanf* o *printf*

Dada una matriz de caracteres declarada como:

```
char nombre[30];
```

La instrucción:

```
scanf ("%s", nombre);        /* o scanf ("%s", &nombre[0]); */
```

Leerá una serie de caracteres del teclado para colocarlos en la matriz *nombre*, empezando a partir de *nombre* [0] y añadiendo automáticamente, a continuación, un carácter de fin de cadena (carácter de código nulo).

El código de formato *%s* funciona como los códigos de formato numéricos, y no como el código de formato *%c*. Es decir, empieza por «saltar» los delimitadores eventuales (espacio o fin de línea) y se interrumpe al encontrar uno de esos delimitadores; no permite por tanto leer cadenas que contengan espacios. Veremos que, para conseguirlo, hay que utilizar una función especializada llamada *gets*.

Del mismo modo, la instrucción:

```
printf("%s",nombre);        /* o printf ("%s", nombre[0]); */
```

Visualizará en la pantalla los caracteres encontrados en la matriz nombre, empezado por nombre[0] y siguiendo hasta encontrar un carácter de fin de cadena. Evidentemente, el código de formato *%s* puede utilizarse como cualquier otro

código de formato y por tanto, en particular, cohabitar (¡con mayor o menor armonía!) con otros códigos de formato (%d, %e, %c,...).

Cuando lea una cadena de n caracteres, hay que prever un espacio de al menos n + 1 caracteres, debido al carácter suplementario de fin de cadena.

Una cadena puede perfectamente no contener ningún carácter (aparte del de fin); se habla en ese caso de «cadena vacía».

Las observaciones hechas a propósito de los riesgos de desbordamiento de una matriz se aplican también aquí. En nuestro ejemplo de lectura, si el usuario proporciona una serie de más de 29 caracteres (lo que hace 30 con el carácter de fin de cadena) se destruirán emplazamientos situados más allá del fin de la matriz *nombre*.

De forma parecida, el código *%s* conduce a *printf* a colocar los caracteres a partir de cierta posición, interrumpiéndose sólo cuando encuentra un carácter de fin de cadena. Si este carácter no ha sido efectivamente introducido en la matriz, el examen seguirá más allá; también en este caso, no hay posibilidad de prevenirse: *printf* sólo recibe de hecho una «dirección de inicio» (la de *nombre* en nuestro ejemplo), y la dimensión de la matriz realmente reservada le resulta desconocida. En este caso, la visualización obtenida puede ser muy curiosa. No olvide, respecto a esto, que todo octeto de la memoria siempre puede considerarse como un carácter.

No confunda el carácter de fin de cadena que es introducido automáticamente por *scanf* para indicar, en memoria, el fin de una cadena, con el delimitador (espacio o fin de línea) que ha servido para detectar el fin del dato proporcionado por el usuario; este delimitador no es copiado en memoria.

1.2 Con las funciones especializadas *gets* y *puts*

Existen dos funciones que manipulan exclusivamente cadenas (una sola a la vez), que complementan de forma útil las posibilidades del código de formato *%s*.

Así, siempre con la declaración anterior, la instrucción:

```
gets (nombre);
```

Lee una serie de caracteres y la coloca en la matriz nombre, terminada por un carácter de fin de cadena; en esta ocasión:

- no se salta ningún delimitador antes de la lectura;
- los espacios son leídos como los demás caracteres;
- la lectura termina al encontrar un carácter de fin de línea.

Del mismo modo, la instrucción:

```
puts (nombre);
```

Visualiza los caracteres encontrados a partir de *&nombre* [0], interrumpiéndose al encontrar el carácter de fin de cadena, y realiza un **salto de línea** (se trata, en realidad, de la única diferencia de comportamiento con el código de formato *%s* de *printf*).

1.3 Ejemplo

Veamos un pequeño ejemplo recapitulativo de las posibilidades de lectura o de escritura de cadenas.

```
#include <stdio.h>
main()
{
   char apellido[20], nombre[20}, localidad[25];
   printf ("cuál es su localidad: ");
   gets(localidad);
   printf("escriba su nombre y apellido: ");
   scanf ("%s %s", nombre, apellido);
   printf ("buenos dias, %s %s que vive en ", nombre, apellido);
   puts(localidad);
}
```

Cuál es su localidad: Barcelona

Introduzca su nombre y apellido: Carlos Moreno

Buenos días, Carlos Moreno que vive en Barcelona

2. Para comparar cadenas: la función *strcmp*

Sabemos que es posible comparar dos caracteres basándose en el valor de su código. Hemos visto en el apartado 3.1 del capítulo IV que el resultado de esta comparación puede variar según el código utilizado pero que, en todos los casos, se tiene la seguridad de que el orden alfabético es respetado para las minúsculas por un lado, y para las mayúsculas por otro, y que los caracteres que representan a las cifras están colocados en su orden natural. Esta posibilidad se generaliza a las cadenas de caracteres. Pero, como se puede suponer, no es posible utilizar los operadores de comparación habituales; hay que recurrir a una función específica llamada strcmp (abreviatura de STRings CoMParison, comparación de cadenas), cuyo prototipo se encuentra en string.h. Esta función, cuya llamada se presenta así:

strcmp (cadena1, cadena2)

Compara las dos cadenas *cadena1* y *cadena2* y devuelve como resultado un valor entero que es:

- positivo si *cadena1* «va después» de *cadena2* y según el orden definido por el código de los caracteres;
- nulo si *cadena1* es igual a *cadena2*, es decir, si las dos cadenas contienen exactamente la misma serie de caracteres;
- negativo si *cadena1* va «antes» de *cadena2*, según el orden definido por el código de los caracteres.

Veamos un programa de ejemplo que lee dos palabras de menos de 30 letras, supone que sólo contienen letras minúsculas e indica si están o no en orden alfabético:

```
#define LONG_PALABRA 30
#include <stdio.h>
#include <string.h>
main ()
{ char palabra1[LONG__PALABRA+1];        /* +1 incluir el carácter de
                                            fin de cadena */
char palabra2[LONG_PALABRA+1];
int comp;              /* para el resultado de la comparación de las palabras */
printf ("introduzca dos palabras en minúsculas:\n");
 scanf ("%s%s", palabra1, palabra2);
 comp = strcmp(palabra1, palabra2);
if (comp<0) printf ("en orden alfabético\n");
 if (comp==0) printf ("idénticas\n");
if (comp>0) printf ("no están en orden alfabético\n");
Introduzca dos palabras en minúsculas: hola buenas
No están en orden alfabético
```

Hemos utilizado una instrucción *#define* para definir la longitud máxima de nuestras cadenas. Observe que ***LONG_PALABRA+1*** es una expresión constante (después de sustituir ***LONG_PALABRA*** por su valor, el compilador encontrará la expresión 30+1, que puede calcular), lo que nos ha permitido emplearla como dimensión.

Hemos utilizado tres instrucciones *if* sucesivas; *if* anidados también habrían podido servir.

Decimos que *strcmp* compara dos «cadenas»; se trata de hecho de un abuso del lenguaje porque, como hemos dicho ya, no existe un verdadero tipo cadena. Por otra parte, los parámetros recibidos por *strcmp* no son más que dos direcciones (más exactamente, dos punteros de tipo *char*)* y esta función se limita a comparar las dos series de caracteres que empiezan en dichas direcciones. En principio, estas direcciones son las de principie de matriz, pero esto no es obligatorio; volveremos sobre ello.

La cadena vacía es considerada siempre como anterior a cualquier otra cadena no vacía.

3. Para copiar cadenas: la función *strcpy*

Como es lógico, no es posible asignar directamente una cadena a otra, precisamente por el mismo motivo por el cual no se podía copiar globalmente el contenido de una matriz en otra.

Por el contrario, se dispone de una función que permite «copiar» una cadena de un lugar a otro. Se trata de *strcpy* (abreviatura de *STRings CoPY*, «copia de cadenas»):

strcpy (destino, fuente) *(string.h)*

Esta función copia la cadena fuente en la cadena destino.

No olvide que los parámetros de esta función no son de hecho más que dos punteros al principio de las cadenas correspondientes. En particular, es necesario que la longitud de la matriz reservada para la cadena *destino* sea suficiente para acoger la cadena a copiar, con riesgo de destrucción de datos si no es así.

Veamos un programa que, como el anterior, lee dos nombres (siempre escritos en minúsculas y de menos de 30 letras) y que los «coloca por orden alfabético», invirtiendo, si ha lugar, el contenido de las dos cadenas correspondientes. Como es de esperar, es necesario contar con una matriz suplementaria, destinada a recibir temporalmente el contenido de una de las matrices.

```
#define LONG_PALABRA 30
#include <stdio.h>
#include <string.h>
main ()
{ char palabra1[LONG_PALABRA+1];
char palabra2 [LONG__PALABRA+1 ];
char palabra[LONG_PALABRA+1];          /* para intercambiar
eventualmente                          palabra1 y
palabra2  */
printf ("introduzca dos palabras en minúsculas:\n");
 scanf ("%s%s", palabra1, palabra2);
if (strcmp(palabral,palabra2)>0)       /* si palabra1 va después de
                                       palabra2, se cambian */
      { strcpy (palabra, palabra1);    /*copia de palabra1 en palabra*/
        strcpy (palabra1, palabra2 ) ; /*copia de palabra2 en palabra1*/
        strcpy (palabra2, palabra);    /*copia de palabra en palabra2*/
      }
printf ("aquí están sus palabras ordenadas: %s %s", palabra1, palabra2);
}
introduzca dos palabras en minúsculas:
 hola buenas
Aquí están sus palabras ordenadas: buenas hola
```

◇ La función *strcpy* copia todos los caracteres que encuentra, hasta encontrar un carácter de fin de cadena, y esto sea cual sea el número de caracteres en cuestión. Existe otra función, llamada *strncpy*, análoga a *strcpy* que permite, gracias a un parámetro suplementario, limitar el número de caracteres a tomar en cuenta. Por ejemplo, con:

strncpy (palabra, palabra1, 5); se copiarán hasta 5 caracteres de palabra1 en palabra.

4. Para obtener la longitud de una cadena: la función strlen

Hasta ahora, hemos manipulado cadenas, sin necesidad de conocer su «longitud», es decir, el número de caracteres que contenían en un momento dado. En efecto, las funciones especializadas que hemos utilizado (,*strcmp, strcpy*,...) tenían en cuenta este aspecto automáticamente (basándose simplemente en el carácter de fin de cadena).
En ciertos casos, sin embargo, tendremos que conocer este número de caracteres. Cierto, es posible contar uno por uno los caracteres que van antes del carácter de fin de cadena. Pero existe una función llamada strlen (abreviatura de STRing LENgth, longitud de cadena) que nos devuelve directamente esta longitud.
Veamos un programa que cuenta el número de letras e que hay en una palabra de menos de 30 caracteres leído del teclado; se utiliza la función *strlen* para conocer el número de caracteres a examinar.

```
#define LONG_PALABRA 30
#include <stdio.h>
#include <string.h>
main ()
{  char palabra[LONG_PALABRA];
   int i;
   int ne;                  /* contador del número de e */
   printf ("introduzca una palabra:\n");
   scanf (" % s ", palabra);
   ne = 0;
   for (i=0; i<strlen(palabra); i=i+1)
   if (palabra[ i ] == 'e')
         ne = ne + 1;
   printf ("su palabra de %d letras contiene %d veces la letra e", strlen(palabra), ne);
}
```

```
Introduzca una palabra: anticonstitucionalmente
Su palabra de 23 letras contiene 2 veces la letra e
```

◇ Aquí, sólo se trata de un ejemplo de ilustración del empleo de *strlen*. En efecto, habríamos podido limitarnos a leer uno a uno los caracteres de nuestra palabra, sin necesidad de guardarlos en una matriz.
La longitud de una cadena vacía es, obviamente, cero.

- ***Ejercicio X.1***

  Escriba un programa que lea una palabra de como máximo 26 caracteres y que lo visualice a razón de un carácter por línea; se utilizará la función *gets* para leer la palabra.