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

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

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

Abstracción

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

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

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

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

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

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

Tipos de datos estructurados: Tablas

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

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

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

Vectores

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

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

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

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

donde:

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

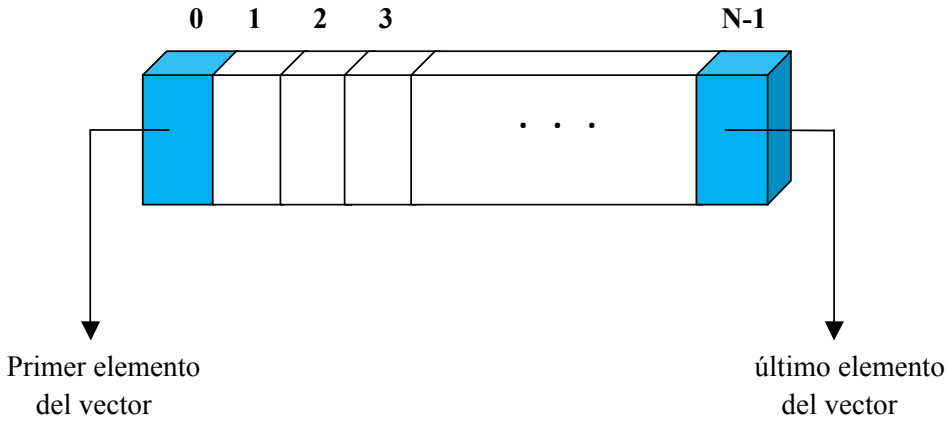


Figura 7.1: Representación gráfica de un vector de N *elementos*

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

Consulta

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

```
nombre_vector[índice];
```

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

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

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

Asignación

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

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

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

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

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

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

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

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

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

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

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

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

Ejemplos

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

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

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

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

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

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

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

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

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

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

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

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

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

Matrices

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

La declaración de una variable matriz tiene la forma siguiente:

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

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

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

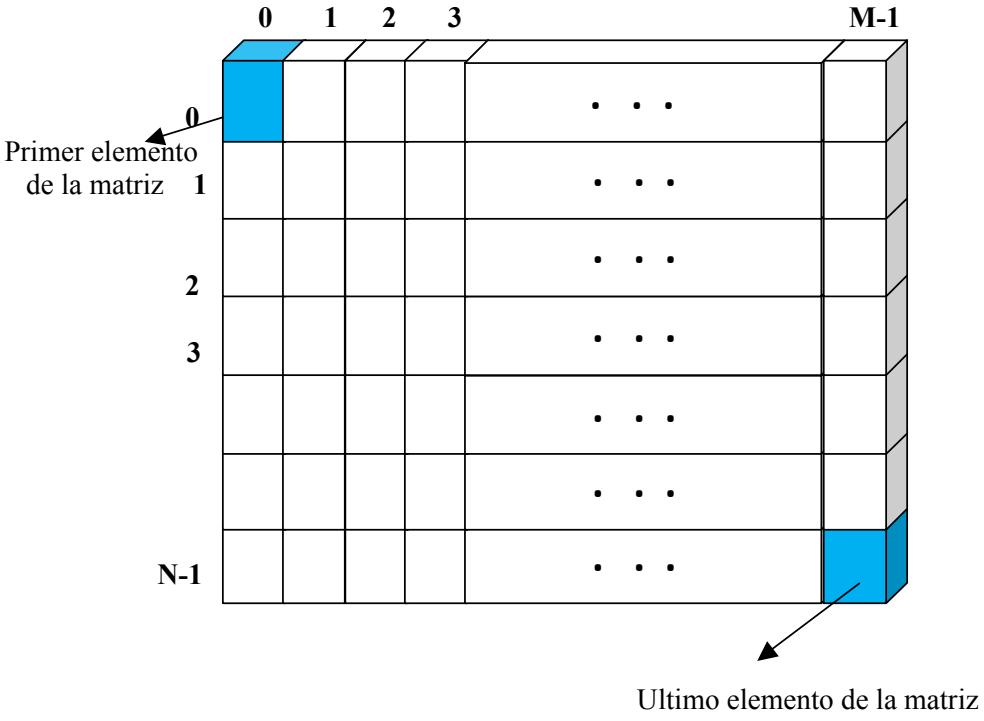


Figura 7.2: Representación gráfica de una matriz de N *filas y* M *columnas*

Consulta

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

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

Asignación

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

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

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

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

Ejemplo

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

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

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

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

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

```
}
```

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

Tablas multidimensionales

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

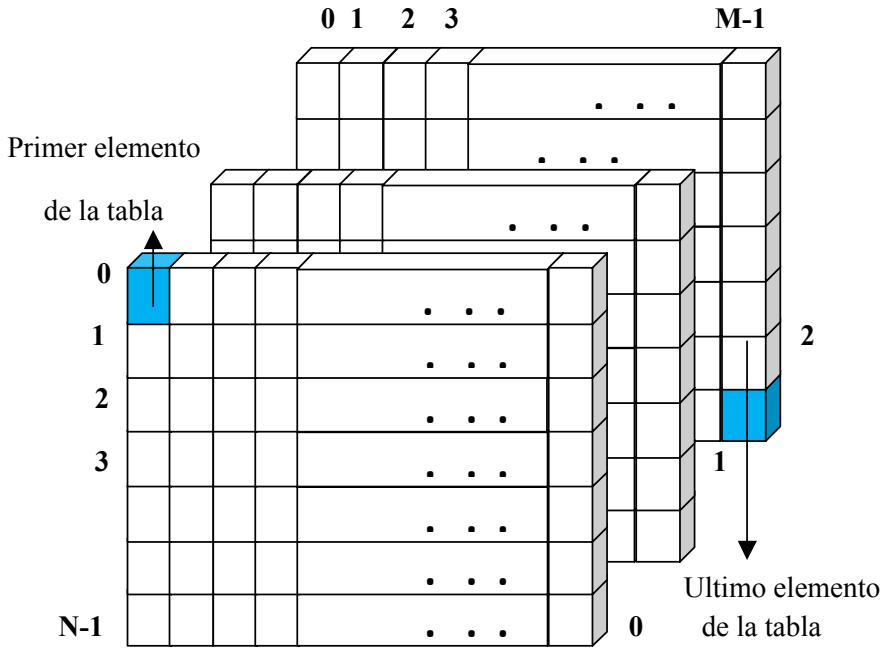


Figura 7.3: Representación gráfica de una tabla de tres dimensiones: N x M x 3

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

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

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

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

Ejemplo

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

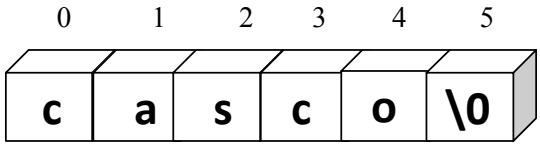
```
#include <stdio.h>
#include <stdlib.h>
#define DIM 10
void main()
{
  int tabla_random [DIM][DIM][DIM], a, b, c;

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

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

Cadenas de caracteres

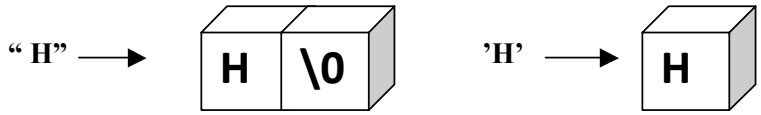
Las cadenas de caracteres son vectores de tipo carácter (char) que reciben un tratamiento especial para simular el tipo de datos "*string*", presente en otros lenguajes de programación.

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

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



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



Asignación

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

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

```
char cadena[10];

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

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

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

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

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

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

puede hacerse también de forma equivalente como:

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

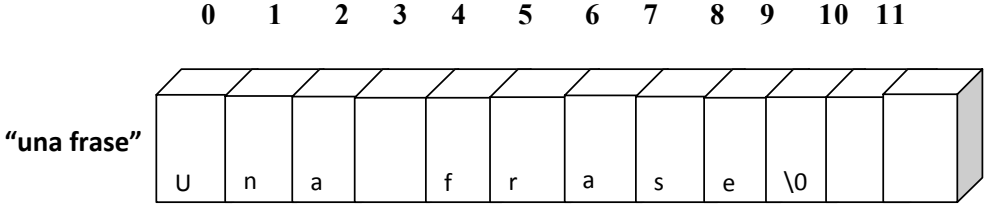
Finalmente, la inicialización anterior puede hacerse sin necesidad de especificar el tamaño del vector correspondiente. En este caso, el compilador se encarga de calcularlo automáticamente, reservando espacio de memoria suficiente para almacenar la cadena, incluyendo el carácter nulo al final. Así pues,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

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

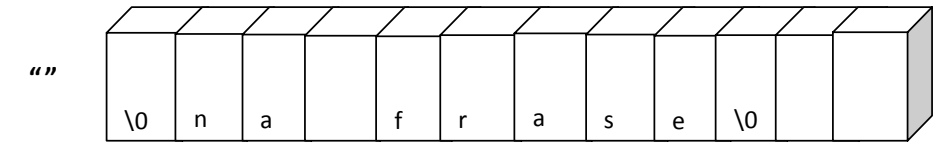
La cadena vacía

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

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



Cadena [0]='\0';



Manejo de cadenas de caracteres

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

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

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

Entrada y salida

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

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

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

Ejemplos

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

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

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

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

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

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

  strcat( cad3, cad2 );
```

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

}
```

Otros tipos de datos

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

Estructuras

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

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

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