6

Arreglos

Objetivos

- Presentar el concepto de la estructura de arreglos de datos.
- Comprender el uso de los arreglos para almacenar, ordenar y buscar listas y tablas de valores.
- Comprender como declarar un arreglo, como incializarlo y como referirse a los elementos individuales de un arreglo.
- Ser capaz de pasar arreglos a funciones.
- Comprender técnicas básicas de clasificación.
- Ser capaz de declarar y de manipular arreglos de varios subíndices.

Con sollozos y lágrimas escogió
Aquellos de mayor tamaño ...
Lewis Carroll

Intente lo último, y nunca acepte la duda;
Nada es tan duro, y la búsqueda lo encontrará.
Robert Herrick

Ahora ve, escríbelo delante de ellos en una mesa,
y anótalo en un libro.
Isaías 30:8

'Está bajo llave en mi memoria'
Y tu mismo conservarás la llave.
William Shakespeare.

Sinopsis

6.1 Introducción

Este capítulo sirve como una introducción al tema tan importante de las estructuras de datos. Los *arreglos* son estructuras de datos consistentes en elementos de datos relacionados del mismo tipo. En el capítulo 10, analizamos la idea o noción en C de **`struct`** (estructuras) —una estructura de datos consistente de elementos relacionados de datos, de tipos posiblemente distintos. Los arreglos y las estructuras son entidades "estáticas", debido a que se conservan del mismo tamaño a todo lo largo de la ejecución del programa (pudieran, naturalmente, ser de la clase de almacenamiento automático y, por lo tanto, creadas y destruidas cada vez que los bloques en los cuales se definen entren o salgan). En el capítulo 12, introducirémos estructuras dinámicas de datos como son listas, colas de espera, pilas y árboles, que pueden crecer o encogerse conforme los programas se ejecutan.

6.2 Arreglos

Un arreglo es un grupo de posiciones en memoria relacionadas entre sí, por el hecho de que todas tienen el mismo nombre y son del mismo tipo. Para referirse a una posición en particular o elemento dentro del arreglo, especificamos el nombre del arreglo y el número de posición del elemento particular dentro del mismo.

En la figura 6.1 se muestra un arreglo de enteros llamado **c**. Este arreglo contiene doce *elementos*. Cualquiera de estos elementos puede ser referenciado dándole el nombre del arreglo seguido por el número de posición de dicho elemento en particular en paréntesis cuadrados o corchetes (**`[]`**). El primer elemento de cualquier arreglo es el *elemento cero*. Entonces, el primer elemento de un arreglo **c** se conoce como **`c[0]`**, el segundo como **`c[1]`**, el séptimo como **`c[6]`** y en general, el elemento de orden *i* del arreglo **c** se conoce como **`c[i-1]`**. Los nombres de los arreglos siguen las mismas reglas convencionales que los demás nombres de variables.

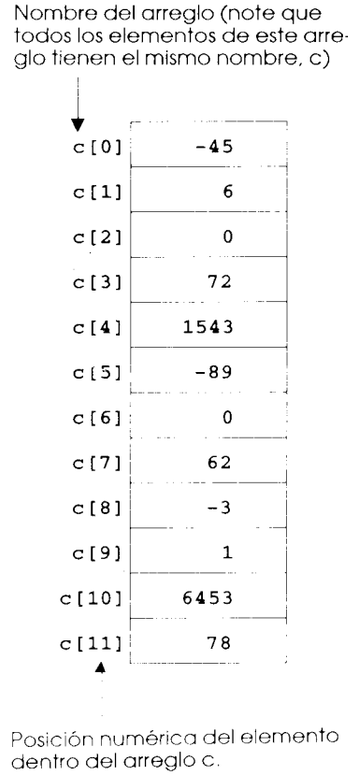

Fig. 6.1 Un arreglo de 12 elementos.

El número de posición que aparece dentro de los corchetes se conoce más formalmente como *subíndice*. Un subíndice debe ser un entero o una expresión entera. Si un programa utiliza una expresión como subíndice, entonces la expresión se evalúa para determinar el subíndice. Por ejemplo, si **`a = 5`** y **`b = 6`**, entonces el enunciado

```
c[a + b] += 2;
```

añade 2 al elemento del arreglo **`c[11]`**. Note que un nombre de arreglo con subíndice es un lvalue que puede ser utilizado en el lado izquierdo de una asignación.

Examinemos más de cerca el arreglo **c** de la figura 6.1. El *nombre* del arreglo es **c**. Sus doce elementos se conocen como **`c[0]`**, **`c[1]`**, **`c[2]`**, ..., **`c[11]`**. El *valor* de **`c[0]`** es **`-45`**, el valor de **`c[1]`** es **`6`**, el valor de **`c[2]`** es **`0`**, el valor de **`c[7]`** es **`62`** y el valor de **`c[11]`** es **`78`**. Para imprimir la suma de los valores contenidos en los primeros tres elementos del arreglo **c**, escribiríamos

```
printf("%d", c[0] + c[1] + c[2]);
```

Para dividir el valor del séptimo elemento del arreglo **c** entre **2** y asignar el resultado a la variable **x**, escribiríamos

```
x = c[6] / 2;
```

Error común de programación 6.1

Es importante notar la diferencia entre el séptimo elemento del arreglo "y el elemento siete del arreglo". Dado que los subíndices de los arreglos empiezan en 0, "el séptimo elemento del arreglo" tiene un subíndice de 6, en tanto que "el elemento 7 del arreglo" tiene un subíndice de siete y de hecho es el octavo elemento del arreglo. Este es una fuente de error por "diferencia de uno".

Los corchetes utilizados para cerrar el subíndice de un arreglo, son de hecho en C considerados como un operador. Tienen el mismo nivel de precedencia que los paréntesis. La gráfica de la figura 6.2 muestra la precedencia y asociatividad de los operadores introducidos hasta este momento dentro del texto. Se muestran de arriba a abajo en orden decreciente de precedencia.

6.3 Cómo declarar los arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada elemento y el número de elementos requerido por cada arreglo, de tal forma que la computadora pueda reservar la cantidad apropiada de memoria. Para indicarle a la computadora que reserve 12 elementos para el arreglo entero *c*, la declaración

```
int c[12];
```

es utilizada. La memoria puede ser reservada para varios arreglos dentro de una sola declaración. Para reservar 100 elementos para el arreglo entero **b** y 27 elementos para el arreglo entero **x**, se utiliza la siguiente declaración:

Operador	Asociatividad	Tipo
`()  []`	de izquierda a derecha	máximo
`++  -  !  (tipo)`	de derecha a izquierda	unario
`*  /  %`	de izquierda a derecha	multiplicativo
`+`	de izquierda a derecha	aditivo
`<  <=  >  >=`	de izquierda a derecha	relacional
`==  !=`	de izquierda a derecha	igualdad
`&&`	de izquierda a derecha	lógica y
`||`	de izquierda a derecha	lógica o
`?:`	de derecha a izquierda	condicional
`=  +=  -=  *=  /=  %=`	de derecha a izquierda	asignación
`,`	de izquierda a derecha	coma

Fig. 6.2 Precedencia de operadores.


```
int b[100], x[27];
```

Los arreglos pueden ser declarados para que contengan otros tipos de datos. Por ejemplo, un arreglo del tipo **char** puede ser utilizado para almacenar una cadena de caracteres. Las cadenas de caracteres y su similitud con arreglos son analizadas en el capítulo 8. La relación entre apuntadores y arreglos es analizada en el capítulo 7.

6.4 Ejemplos utilizando arreglos

El programa de la figura 6.3 utiliza la estructura de repetición **for** para inicializar los elementos de un arreglo entero de diez elementos **n** a ceros, e imprime el arreglo en formato tabular.

Advierta que decidimos no colocar una línea en blanco entre el primer enunciado **printf** y la estructura **for** en la figura. 6.3, dado que están relacionados de forma íntima. En este caso, el enunciado **printf** muestra los encabezados de columnas de las dos columnas impresas en la estructura **for**. Los programadores omiten la línea en blanco entre la estructura **for** y algún enunciado **printf** que esté muy relacionado.

```
/* initializing an array */
#include <stdio.h>

main()
{
   int n[10], i;

   for (i = 0; i <= 9; i++)        /* initialize array */
      n[i] = 0;

   printf("%s%13s\n", "Element", "Value");
   for(i = 0; i <= 9; i++)         /* print array */
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Fig. 6.3 Cómo inicializar los elementos de un arreglo a ceros.

Los elementos de un arreglo también pueden ser inicializados en la declaración del arreglo mismo, haciendo seguir a la declaración con el signo igual y una lista separada por comas (encerrada entre llaves) de *inicializadores*. El programa de la figura 6.4 inicializa un arreglo entero con diez valores e imprime el arreglo en formato tabular.

Si dentro del arreglo existe un número menor de inicializadores que de elementos, los elementos restantes son inicializados a cero de forma automática. Por ejemplo, los elementos del arreglo **n** de la figura 6.3 podrían haber sido inicializados a cero con la declaración

```
int n[10] = {0};
```

con lo que de manera explícita se inicializa el primer elemento a cero y, por lo demás, los nueve elementos restantes se inicializan automáticamente a cero, porque existen menos inicializadores que elementos del arreglo. Es importante recordar que los arreglos no son de forma automática inicializados a cero. El programador debe por lo menos inicializar el primer elemento a cero, para que los demás queden automáticamente inicializados a cero. Este método de inicializar los elementos del arreglo a `0` se ejecuta en tiempo de compilación. El método utilizado en la figura 6.3, puede ser llevado a cabo en forma repetida, conforme se ejecuta el programa.

```
/* Initializing an array with a declaration */
#include <stdio.h>

main()
{
   int n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
   int i;

   printf("%s%13s\n", "Element", "Value");

   for(i = 0; i <= 9; i++)
      printf("%7d%13d\n", i, n[i]);

   return 0;
}
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Fig. 6.4 Cómo inicializar los elementos de un arreglo mediante una declaración.

Error común de programación 6.2

Olvidar inicializar los elementos de un arreglo cuyos elementos deban de estar inicializados.

La siguiente declaración de arreglo

```
int n[5] = {32, 27, 64, 18, 95, 14};
```

generaría un error de sintaxis, porque en el arreglo existen 6 inicializadores y únicamente 5 elementos.

Error común de programación 6.3

El proporcionar más inicializadores en una lista inicializadora de arreglo que elementos existan dentro del mismo constituye un error de sintaxis.

Si de una declaración con una lista inicializadora se omite el tamaño del arreglo, el número de elementos en el arreglo será el número de elementos incluidos en la lista inicializadora. Por ejemplo,

```
int n[] = {1, 2, 3, 4, 5};
```

crearía un arreglo de cinco elementos.

El programa de la figura 6.5 incializa los elementos de un arreglo **s** de diez elementos a los valores `2, 4, 6, ... 20`, e imprime el arreglo en formato tabular. Los valores se generan multiplicando el contador del ciclo por `2` y añadiendo `2`.

La directiva de preprocesador **`#define`** se introduce en este programa. La línea

```
#define SIZE 10
```

define una *constante simbólica* **`SIZE`** cuyo valor es `10`. Una constante simbólica es un identificador que se *remplaza con texto* de remplazo en el preprocesador C, antes de que el programa sea compilado. Cuando el programa es preprocesado, todas las instancias de la constante simbólica **`SIZE`** serán remplazadas por el texto de remplazo `10`. El uso de constantes simbólicas para especificar tamaños de arreglo hacen que los programas sean más *dimensionables*. En la figura 6.5, el primer ciclo **`for`** podría llenar un arreglo de 1000 elementos sólo modificando el valor de **`SIZE`** de la directiva **`#define`** de `10` a `1000`. Si no se hubiera utilizado la constante simbólica **`SIZE`**, tendríamos que modificar el programa en tres lugares distintos para dimensionar el programa, a fin de que el arreglo pudiera manejar 1000 elementos. Conforme los programas se hacen más grandes, esta técnica se hace más útil para la escritura de programas claros.

Error común de programación 6.4

Terminar una directiva de preprocesador **`#define`**, *o bien* **`#include`** *con un punto y coma. Recuerde que las directivas de preprocesador no son enunciados C.*

En la directiva de preprocesador **`#define`** anterior, si ésta estuviera terminada con un punto y coma, todas las ocurrencias de aparición de la constante simbólica **`SIZE`** en el programa serían remplazadas por el texto `10;` por dicho preprocesador. Esto pudiera llevar a errores de sintaxis en tiempo de compilación, o errores lógicos en tiempo de ejecución. Recuerde que el preprocesador no es C, es sólo un manipulador de texto.

Error común de programación 6.5

Asignar un valor a una constante simbólica en un enunciado ejecutable es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio para ella como hace con las variables que contienen valores durante la ejecución.

```
/* Initialize the elements of array s to
   the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10

main()
{
   int s[SIZE], j;

   for (j = 0; j <= SIZE - 1; j++)   /* set the values */
      s[j] = 2 + 2 * j;

   printf("%s%13s\n", "Element", "Value");

   for (j = 0; j <= SIZE - 1; j++)   /* print the values */
      printf("%7d%13d\n", j, s[j]);

   return 0;
}
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Fig. 6.5 Cómo generar los valores a colocarse en los elementos de un arreglo.

Observación de ingeniería de software 6.1

Definir el tamaño de cada arreglo como una constante simbólica hace a los programas más dimensionables.

Práctica sana de programación 6.1

Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y le recuerden al programador que las constantes simbólicas no son variables.

El programa de la figura 6.6 suma los valores contenidos en un arreglo entero **a** de doce elementos. El enunciado en el cuerpo del ciclo **for** se ocupa de la totalización.

Nuestro siguiente ejemplo utiliza arreglos para resumir los resultados de datos recopilados en una investigación. Considere el enunciado del problema.

A cuarenta alumnos se les preguntó el nivel de calidad de los alimentos de la cafetería para alumnos en una escala de 1 a 10 (1 significa terrible y 10 significa excelente). Coloque las cuarenta respuestas en un arreglo entero y resuma los resultados de la encuesta.


```
/* Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

main()
{
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45},
       i, total = 0;

   for (i = 0; i <= SIZE - 1; i++)
      total += a[i];

   printf("Total of array element values is %d\n", total);
   return 0;
}
```

```
Total of array element values is 383
```

Fig. 6.6 Cómo calcular la suma de los elementos de un arreglo.

Esta es una aplicación típica de arreglo (vea la figura 6.7). Deseamos resumir el número de respuestas de cada tipo (es decir, de 1 hasta 10). El arreglo **responses** es un arreglo de 40 elementos correspondiente a las respuestas de los alumnos. Utilizamos un arreglo de once elementos, **frequency** para contar el número de ocurrencias de cada respuesta. Ignoramos el primer elemento **frequency[0]**, porque es más lógico tener un incremento de respuesta 1 **frequency[1]** que **frequency[0]**. Esto nos permite utilizar cada respuesta directa como subíndice en el arreglo **frequency**.

Práctica sana de programación 6.2

Busque claridad en el programa. Algunas veces será preferible sacrificar una utilización más eficiente de memoria o de tiempo de procesador en aras de escribir programas más claros.

Sugerencia de rendimiento 6.1

Algunas veces las consideraciones de rendimiento tienen mayor importancia que las consideraciones de claridad.

El primer ciclo **for** toma las respuestas del arreglo **response** una por una e incrementa uno de los diez contadores (**frequency[1]** hasta **frequency[10]**) en el arreglo **frequency**. El enunciado clave del ciclo es

```
++frequency[responses[answer]];
```

Este enunciado incrementa el contador **frequency** apropiado, dependiendo del valor de **responses[answer]**. Por ejemplo, cuando la variable del contador **answer** es 0, **responses[answer]** es 1 y, por lo tanto, **++frequency[responses[answer]];** se interpreta en realidad como

```
++frequency[1];
```

```
/* Student poll program */
#include <stdio.h>
#define RESPONSE_SIZE 40
#define FREQUENCY_SIZE 11

main()
{
   int answer, rating;
   int responses[RESPONSE_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10};
   int frequency[FREQUENCY_SIZE] = {0};

   for(answer = 0; answer <= RESPONSE_SIZE - 1; answer++)
      ++frequency[responses[answer]];

   printf("%s%17s\n", "Rating", "Frequency");

   for(rating = 1; rating <= FREQUENCY_SIZE - 1; rating++)
      printf("%6d%17d\n", rating, frequency[rating]);

   return 0;
}
```

```
Rating          Frequency
     1                  2
     2                  2
     3                  2
     4                  2
     5                  5
     6                 11
     7                  5
     8                  7
     9                  1
    10                  3
```

Fig. 6.7 Un programa sencillo de análisis de una encuesta de alumnos.

lo que incrementa el elemento uno del arreglo. Cuando **answer** es **1**, **responses[answer]** es **2** y, por lo que, **++frequency [responses[answer]];** se interpreta como

```
++frequency[2];
```

lo que incrementa el elemento dos del arreglo. Cuando **answer** es **2**, **responses[answer]** es **6**, por lo que **++frequency [responses[answer]];** se interpreta como

```
++frequency[6];
```

lo que incrementa al elemento seis del arreglo, y así en lo sucesivo. Note que, independiente del número de respuestas procesadas en la encuesta, sólo se requiere un arreglo de once elementos

(ignorando al elemento cero) para resumir los resultados. Si los datos tuvieran valores inválidos como el 13, el programa intentaría añadir **1** a **frequency[13]**. Esto quedaría fuera de los límites del arreglo. *C no tiene verificación de límites de arreglo, para impedir que la computadora se refiera a un elemento no existente*. Entonces, un programa en ejecución podría salirse sin aviso del final de un arreglo. El programador debe asegurarse que todas las referencias al arreglo se conservan dentro de los límites del mismo.

Error común de programación 6.6

Referirse a un elemento exterior a los límites del arreglo.

Práctica sana de programación 6.3

Al ciclar a través de un arreglo, el subíndice de un arreglo no debe de pasar nunca por debajo de 0 y siempre tiene que ser menor que el número total de elementos del arreglo (tamaño -1). Asegúrese que la condición de terminación del ciclo impide el acceso a elementos fuera de este rango.

Práctica sana de programación 6.4

Mencione el subíndice alto del arreglo en una estructura `for`, a fin de ayudar a eliminar los errores por diferencia de uno.

Práctica sana de programación 6.5

Los programas deberían verificar la corrección de todos los valores de entrada, para impedir que información errónea afecte los cálculos del programa.

Sugerencia de rendimiento 6.2

Los efectos (normalmente serios) de referirse a elementos fuera de los límites del arreglo, son dependientes del sistema.

Nuestro siguiente ejemplo (figura 6.8) lee números de un arreglo y representa la información en forma de una gráfica de barras o histograma —cada número es impreso, y a continuación al lado del número se imprime una barra, formada por muchos asteriscos. De hecho el que dibuja las barras es el ciclo anidado **for**. Note el uso de **printf ("\n")** para dar por terminada la barra del histograma.

En el capítulo 5 indicamos que mostraríamos un método más elegante de escribir el programa de tirada de dados de la figura 5.10. El problema era tirar un dado de seis caras 6000 veces, para probar si el generador de números aleatorios de verdad produce números al azar. Una versión en arreglo de este programa, se muestra en la figura 6.9

Hasta este punto, sólo nos hemos ocupado de arreglos enteros. Sin embargo, los arreglos son capaces de contener datos de cualquier tipo. Ahora analizarémos el almacenamiento de cadenas en arreglos de caracteres. Hasta ahora, la única capacidad que tenemos de procesamiento de cadenas es la salida de una cadena mediante **printf**. En C, una cadena como "**hello**", de hecho es un arreglo de caracteres individuales.

Los arreglos de caracteres tienen varias características únicas. Un arreglo de caracteres puede ser inicializado utilizando una literal de cadena. Por ejemplo, la declaración

```
char string1[] = "first";
```

inicializa los elementos de la cadena **string1** a los caracteres individuales de la literal de cadena **"first"**. El tamaño del arreglo **string1** en la declaración anterior queda determinada por el compilador, basado en la longitud de la cadena. Es importante hacer notar que la cadena **"first"**

```
/* Histogram printing program */
#include <stdio.h>
#define SIZE 10

main()
{
   int n[SIZE] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
   int i, j;

   printf("%s%13s%17s\n", "Element", "Value", "Histogram");

   for (i = 0; i <= SIZE - 1; i++) {
      printf("%7d%13d        ", i, n[i]);

      for(j = 1; j <= n[i]; j++)    /* print one bar */
         printf("%c", '*');

      printf("\n");
   }

   return 0;
}
```

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

Fig. 6.8 Un programa que imprime histogramas.

contiene cinco caracteres, *más* un carácter especial de terminación de cadena, conocido como *carácter nulo*. Entonces, el arreglo **string1**, de hecho contiene seis elementos. La representación de la constante de caracteres del carácter nulo es **'\0'**. En C todas las cadenas terminan con este carácter. Un arreglo de caracteres, representando una cadena, debería declararse siempre lo suficiente grande para contener el número de caracteres de la cadena, incluyendo el carácter nulo de terminación.

Los arreglos de caracteres también pueden ser inicializados con constantes individuales de caracteres en una lista de incialización. La declaración anterior es equivalente a

```
char string1[] = {'f', 'i', 'r', 's', 't', '\0'};
```

Dado que la cadena de hecho es un arreglo de caracteres, podemos tener acceso directo a los caracteres individuales de una cadena, utilizando la notación de subíndices de arreglos. Por ejemplo, **string1[0]**, es el carácter **'f'** y **string1[3]** es el carácter **'s'**.


```
/* Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

main()
{
   int face, roll, frequency[SIZE] = {0};

   srand(time(NULL));

   for (roll = 1; roll <= 6000; roll++) {
      face = rand() % 6 + 1;
      ++frequency[face];        /* replaces 20-line switch */
   }                            /* of Fig. 5.10 */

   printf("%s%17s\n", "Face", "Frequency");

   for (face = 1; face <= SIZE - 1; face++)
      printf("%4d%17d\n", face, frequency[face]);

   return 0;
}
```

```
Face        Frequency
   1             1037
   2              987
   3             1013
   4             1028
   5              952
   6              983
```

Fig. 6.9 Programa de tirada de dados utilizando arreglos en vez de **switch**.

También podemos introducir desde el teclado directamente una cadena en un arreglo de caracteres, utilizando **scanf** y la especificación de conversión **%s**. Por ejemplo, la declaración.

```
char string2[20];
```

crea un arreglo de caracteres capaz de almacenar una cadena de 19 caracteres y un carácter nulo de terminación. El enunciado

```
scanf("%s", string2);
```

lee una cadena del teclado y la coloca en **string2**. Note que el nombre del arreglo se pasa a **scanf** sin colocar el **&** precedente, que en otras variables se utiliza. El **&** es utilizado por lo regular para darle a **scanf** una localización de variable en memoria, a fin de que se pueda almacenar un valor ahí. En la sección 6.5 analizarémos cómo pasar arreglos a funciones. Veremos que el nombre de un arreglo es la dirección del inicio del arreglo y, por lo tanto, **&** no es necesario.

Es la responsabilidad del programador asegurarse que el arreglo al cual se lee la cadena, es capaz de contener cualquier cadena que el usuario escriba en el teclado. La función **scanf** lee caracteres del teclado hasta que se encuentra con el primer carácter de espacio en blanco sin importarle qué tan grande es el arreglo. Por lo tanto, **scanf** podría escribir más allá del final del arreglo.

Error común de programación 6.7

No proporcionar, en un programa, a scanf un arreglo de caracteres lo suficiente grande para almacenar una cadena escrita en el teclado, puede dar como resultado una pérdida de datos, así como otros errores en tiempo de ejecución.

Un arreglo de caracteres que represente a una cadena puede ser sacado utilizando **printf** y el especificador de conversión **%s**. El arreglo **string2** se imprime utilizando el enunciado

```
printf("%s\n", string2);
```

Note que a **printf**, al igual que a **scanf**, no le importa qué tan grande es el arreglo de caracteres. Los caracteres de la cadena serán impresos, hasta que un carácter de terminación nulo sea encontrado.

En la figura 6.10 se demuestra la inicialización de un arreglo de caracteres con una literal de caracteres, la lectura de una cadena a un arreglo de caracteres, la impresión de un arreglo de caracteres como una cadena, y el acceso a caracteres individuales de una cadena.

```
/* Treating character arrays as strings */
#include <stdio.h>

main()
{
   char string1[20], string2[] = "string literal";
   int i;

   printf("Enter a string: ");
   scanf("%s", string1);
   printf("string1 is: %s\nstring2: is %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2);

   for (i = 0; string1[i] != '\0'; i++)
      printf("%c ", string1[i]);

   printf("\n");
   return 0;
}
```

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Fig. 6.10 Cómo tratar arreglos de caracteres como cadenas.

En la figura 6.10 se utiliza la estructura **for** para ciclar a través del arreglo **string1** e imprimir los caracteres individuales separados por espacios mediante la especificación de conversión **%c**. La condición en la estructura **for, string1[i] != '\0'**, es verdadera en tanto, dentro de la cadena, el carácter nulo de terminación no sea encontrado.

En el capítulo 5, se analizó el especificador de clase de almacenamiento **static**. Una variable local **static**, en una definición de función, existe durante la duración del programa, pero resulta sólo visible en el cuerpo de la función. Podemos aplicar **static** a una declaración de arreglo local, de tal forma que el arreglo no sea creado e inicializado cada vez que se llame a la función, y el arreglo no se destruya cada vez que en el programa la función se termine. Esto reduce el tiempo de ejecución del programa, en particular en el caso de programas con funciones de llamado frecuente y que contengan arreglos extensos.

Sugerencia de rendimiento 6.3

En funciones que contengan arreglos automáticos donde la función entra y sale de alcance con frecuencia, haga **static** *dicho arreglo, de tal forma que no sea necesario crearlo cada vez que la función sea llamada.*

Los arreglos que se declaran **static** son inicializados de forma automática una vez en tiempo de compilación. Si un arreglo **static** no se inicializa de manera explícita por el programador, éste quedará inicializado a cero por el compilador.

En la figura 6.11 se demuestra la función **staticArrayInit**, con un arreglo local declarado **static** y una función **automaticArrayInit** con un arreglo local automático. La función **staticArrayInit** es llamada dos veces. El arreglo local **static** en la función es inicializado a cero por el compilador. La función imprime el arreglo, añade 5 a cada uno de los elementos, y lo vuelve a imprimir. La segunda vez que la función es llamada, el arreglo **static** contiene los valores almacenados durante la primera llamada de la función. La función **automaticArrayInit** también es llamada dos veces. Se inicializan los elementos del arreglo local automático en la función con los valores 1, 2 y 3. La función imprime el arreglo, añade 5 a cada elemento y lo vuelve a imprimir. La segunda vez que se llama a la función, los elementos del arreglo están otra vez inicializados a 1, 2, 3, porque el arreglo tiene una duración de almacenamiento automático.

Error común de programación 6.8

Suponer que los elementos de un arreglo local, que está declarado como **static**, *están inicializados a cero, cada vez que la función sea llamada donde se declara el arreglo.*

6.5 Cómo pasar arreglos a funciones

Para pasar cualquier argumento de arreglo a una función, especifique el nombre del arreglo, sin corchete alguno. Por ejemplo, si el arreglo **hourlyTemperatures** ha sido declarado como

```
in hourlyTemperatures[24];
```

el enunciado de llamada a la función

```
modifyArray(hourlyTemperatures, 24);
```

pasa el arreglo **hourlyTemperatures** y su tamaño, a la función **modifyArray**. Al pasar un arreglo a una función, el tamaño del arreglo a menudo se pasa a la función, de tal forma que pueda procesar el número específico de elementos incluidos en dicho arreglo.

```
/* Static arrays are initialized to zero */
#include <stdio.h>

void staticArrayInit(void);
void automaticArrayInit(void);

main()
{
   printf("First call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   printf("\n\nSecond call to each function:\n");
   staticArrayInit();
   automaticArrayInit();
   return 0;
}

/* function to demonstrate a static local array */
void staticArrayInit(void)
{
   static int a[3];
   int i;

   printf("\nValues on entering staticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf("\nValues on exiting staticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}

/* function to demonstrate an automatic local array */
void automaticArrayInit(void)
{
   int a[3] = {1, 2, 3};
   int i;

   printf("\n\nValues on entering automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i]);

   printf("\nValues on exiting automaticArrayInit:\n");

   for (i = 0; i <= 2; i++)
      printf("array1[%d] = %d  ", i, a[i] += 5);
}
```

Fig. 6.11 Los arreglos estáticos son de forma automática inicializados a cero, si no han sido de manera explícita inicializados por el programador (parte 1 de 2).


```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array[2] = 5

Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array1[0] = 1  array1[1] = 2  array1[2] = 3
Values on exiting automaticArrayInit:
array1[0] = 6  array1[1] = 7  array1[2] = 8
```

Fig. 6.11 Los arreglos estáticos son de forma automática inicializados a cero, si no han sido explícitamente inicializados por el programador (parte 2 de 2).

C pasa de forma automática los arreglos a las funciones utilizando simulación de llamadas por referencia —las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de los llamadores. ¡El nombre del arreglo de hecho es la dirección del primer elemento de dicho arreglo! Dado que ha sido pasada la dirección inicial del arreglo, la función llamada sabe precisamente dónde está el arreglo almacenado. Por lo tanto, cuando en su cuerpo de función, la función llamada modifica los elementos del arreglo, está modificando los elementos reales del arreglo, en sus localizaciones de memoria originales.

En la figura 6.12 puede demostrarse que un nombre de un arreglo es en realidad la dirección del primer elemento de dicho arreglo, imprimiendo **`array`** y **`&array[0]`**, utilizando la especificación de conversión %p —especificación especial de conversión para impresión de direcciones. La especificación de conversión %p da por lo regular salida a direcciones como de números hexadecimales. Los números hexadecimales (base 16) se forman de los dígitos 0 a 9 y de las letras A hasta la F. Se utilizan a menudo como una escritura abreviada de valores de enteros grandes. En el Apéndice E: en los Sistemas numéricos se proporciona un análisis en profundidad de las relaciones entre enteros binarios (de base 2), octales (en base 8), decimales (en base diez; enteros estándar), y hexadecimales. La salida muestra que tanto **`array`** como **`&array[0]`** tienen el mismo valor, es decir **`FFF0`.** La salida de este programa es dependiente del sistema, pero las direcciones siempre resultarán idénticas.

Sugerencia de rendimiento 6.4

Tiene sentido pasar arreglos simulando llamadas por referencia por razones de rendimiento. Si los arreglos fueran pasados en llamadas por valor, se pasaría una copia de cada uno de los elementos. En el caso de arreglos grandes pasados con frecuencia, esto tomaría mucho tiempo y consumiría gran cantidad de espacio de almacenamiento para las copias de los arreglos.

```
/* The name of an array is the same as &array[0] */
#include <stdio.h>

main()
{
   char array[5];

   printf("    array = %p\n&array[0] = %p\n",
          array, &array[0]);
   return 0;
}
```

```
    array = FFF0
&array[0] = FFF0
```

Fig. 6.12 El nombre de un arreglo es el mismo que la dirección del primer elemento de dicho arreglo.

Observación de ingeniería de software 6.2

Es posible pasar un arreglo por valor utilizando una triquiñuela sencilla que explicaremos en el capítulo 10.

A pesar de que se pasan arreglos completos simulando llamadas por referencia, los elementos individuales del arreglo se pasan en llamadas por valor, de la misma forma que se pasan las variables simples. Estas simples y únicas porciones de datos son conocidas como *escalares o cantidades escalares.* Para pasar un elemento de un arreglo a una función, utilice como argumento en la llamada a la función el nombre con subíndice del elemento del arreglo. En el capítulo 7, mostramos cómo simular llamadas por referencia para escalares (es decir, para elementos del arreglo y variables individuales).

Para que una función reciba un arreglo a través de una llamada de función, la lista de parámetros de la función debe especificar que se va a recibir un arreglo. Por ejemplo, el encabezado de función para la función **modifyArray** pudiera ser escrito como

```
void modifyArray(int b[], int size)
```

indicando que **modifyArray** espera recibir un arreglo de enteros en el parámetro **b** y un cierto número de elementos de arreglo en el parámetro **size.** No es necesario indicar el tamaño del arreglo dentro de los corchetes del arreglo. Si se incluye, el compilador lo ignorará. Dado que los arreglos son pasados de forma automática por simulación de llamadas por referencia, cuando la función llamada utiliza el nombre del arreglo **b**, de hecho se estará refiriendo al arreglo real en el llamador (arreglo **hourlyTemperatures** de la llamada anterior). En el capítulo 7, presentamos otras notaciones para indicar que un arreglo está siendo recibido por una función. Como veremos, en el lenguaje C estas formas de notación se basan en las relaciones íntimas existentes entre arreglos y apuntadores.

Advierta la apariencia extraña del prototipo de función correspondiente a **modifyArray**

```
void modifyArray(int [], int);
```

Este prototipo podría haber sido escrito

```
void modifyArray(int anyArrayName[], int anyVariableName)
```


pero como aprendimos en el capítulo 5, el compilador de C ignora los nombres de variables dentro de los prototipos.

Práctica sana de programación 6.6

Algunos programadores incluyen nombres de variables en los prototipos de función, para que los programas resulten más claros. El compilador ignorará estos nombres.

Recuerde, el prototipo le indica al compilador el número de argumentos y los tipos en que aparecerán cada uno de los argumentos (en el orden en que se ejecutarán estos).

El programa de la figura 6.13 demuestra la diferencia entre pasar un arreglo completo y pasar un elemento de un arreglo. El programa primero imprime los cinco elementos de un arreglo entero **a**. A continuación, **a** y su tamaño son pasados a la función **modifyArray,** donde cada uno de los elementos de **a** es multiplicado por 2. A continuación **a** se vuelve a imprimir en **main**. Como se muestra en la salida, los elementos de **a** en realidad han sido modificados por **modifyArray**. Ahora el programa imprime el valor de **a[3]** y lo pasa a la función **modifyElement**. La función **modifyElement** multiplica su argumento por 2 e imprime el nuevo valor. Note que cuando **a[3]** es vuelto a imprimir en **main**, no ha sido modificado porque los elementos individuales del arreglo han sido pasados en llamada por valor.

```
/* Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

void modifyArray(int [], int);  /* appears strange */
void modifyElement(int);

main()
{
   int a[SIZE] = {0, 1, 2, 3, 4};
   int i;

   printf("Effects of passing entire array call "
          "by reference:\n\nThe values of the "
          "original array are:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);

   printf("\n");
   modifyArray(a, SIZE);  /* array a passed call by reference */
   printf("The values of the modified array are:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d", a[i]);

   printf("\n\n\nEffects of passing array element call "
          "by value:\n\nThe value of a[3] is %d\n", a[3]);
   modifyElement(a[3]);
   printf("The value of a[3] is %d\n", a[3]);
   return 0;
}
```

Fig. 6.13 Cómo pasar arreglos y elementos individuales de arreglo a funciones (parte 1 de 2).

```
void modifyArray(int b[], int size)
{
   int j;

   for (j = 0; j <= size - 1; j++)
      b[j] *= 2;
}

void modifyElement(int e)
{
   printf("Value in modifyElement is %d\n", e *= 2);
}
```

```
Effects of passing entire array call by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call by avalue:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Fig. 6.13 Cómo pasar arreglos y elementos individuales de arreglo a funciones (parte 2 de 2).

Pudieran existir en sus programas situaciones en las cuales a una función no se le deberá permitir que modifique los elementos de un arreglo. Dado que los arreglos son siempre pasados simulando llamadas por referencia, resulta difícil de controlar la modificación de los valores de un arreglo. C proporciona un calificador especial de tipo **const**, para evitar la modificación de los valores de arreglo en una función. Cuando un parámetro de arreglo es antecedido por el calificador **const**, los elementos del arreglo se convierten en constantes en el cuerpo de la función, y cualquier intento para modificar un elemento en el cuerpo de la función da como resultado un error en tiempo de compilación. Esto le permite al programador corregir un programa, de forma tal que no intente modificar los elementos del arreglo. A pesar de que el calificador **const** está bien definido en el estándar ANSI, los diferentes sistemas C tienen variantes en sus capacidades para ponerlo en práctica.

En la figura 6.14 se demuestra el calificador **const**. La función **tryToModifyArray** está definida con el parámetro **const int b[]** que especifica que el arreglo **b** es constante, y no puede ser modificado. La salida muestra los mensajes de error producidos por el compilador Borland C++. Cada uno de los tres intentos de la función para modificar elementos del arreglo resultan en el error de compilación **"Cannot modify a const object"**. El calificador **const** será vuelto a analizar en el capítulo 7.

Observación de ingeniería de software 6.3

El calificador de tipo **const** *puede ser aplicado a un parámetro de arreglo en una definición de función, para impedir que en el cuerpo de la función el arreglo original sea modificado. Este es otro ejemplo del principio de mínimo privilegio. Las funciones no deben tener, ni se les dará, capacidad de modificar un arreglo, a menos de que sea en lo absoluto necesario.*


```
/* Demonstrating the const type qualifier */
#include <stdio.h>

void tryToModifyArray(const int []);

main()
{
   int a[] = {10, 20, 30};

   tryToModifyArray(a);
   printf("%d %d %d\n", a[0], a[1], a[2]);
   return 0;
}

void tryToModifyArray(const int b[])
{
   b[0] /= 2;     /* error */
   b[1] /= 2;     /* error */
   b[2] /= 2;     /* error */
}
```

```
Compiling FIG6_14.C:
Error FIG6_14-C 16: Cannot modify a const object
Error FIG6_14.C 17: Cannot modify a const object
Error FIG6_14.C 18: Cannot modify a const object
Warning FIG6_14.C 19: Parameter 'b' is never used
```

Fig. 6.14 Demostración del calificador de tipo **const**.

6.6 Cómo ordenar arreglos

La *ordenación* de datos (es decir, colocar los datos en un orden particular, como orden ascendente o descendente) es una de las aplicaciones más importantes de la computación. Un banco clasifica todos los cheques por número de cuenta, de tal forma que al final de cada mes pueda preparar estados bancarios individuales. Para facilitar la búsqueda de números telefónicos, las empresas telefónicas clasifican sus listas de cuentas por apellido y dentro de ello, por nombre. Virtualmente todas las organizaciones deben clasificar algún dato y en muchos casos, cantidades masivas de información. La ordenación de datos es un problema intrigante que ha atraído alguno de los esfuerzos más intensos de investigación en el campo de la ciencia de la computación. En este capítulo analizamos lo que quizás es el esquema más simple de ordenación conocido. En los ejercicios y en el capítulo 12, investigamos esquemas más complejos, que consiguen rendimientos muy superiores.

Sugerencia de rendimiento 6.5

A menudo, los algoritmos más sencillos tienen rendimientos pobres. Su virtud estriba en que son fáciles de escribir, de probar y de depurar. Sin embargo, para obtener rendimiento máximo con frecuencia se requiere de algoritmos más complejos.

El programa en la figura 6.15 ordena en orden ascendente los valores de los elementos de un arreglo **a** de diez elementos. La técnica que utilizamos se conoce como *ordenación tipo burbuja*

u ordenación por hundimiento, porque los valores más pequeños de forma gradual "flotan" hacia la parte superior del arreglo, como suben las burbujas de aire en el agua, en tanto que los valores más grandes se hunden hacia el fondo del arreglo. La técnica consiste en llevar a cabo varias pasadas a través del arreglo. En cada pasada, se comparan pares sucesivos de elementos. Si un par está en orden creciente (o son idénticos sus valores), dejamos los valores tal y como están. Si un par aparece en orden decreciente, sus valores en el arreglo se intercambian de lugar.

```
/* This program sorts an array's values into
   ascending order */
#include <stdio.h>
#define SIZE 10

main()
{
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
   int i, pass, hold;

   printf("Data items in original order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   for (pass = 1; pass <= SIZE - 1; pass++)  /* passes */

      for (i = 0; i <= SIZE - 2; i++)       /* one pass */

         if (a[i] > a[i + 1]) {       /* one comparison */
            hold = a[i];                 /* one swap */
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }

   printf("\nData items in ascending order\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%4d", a[i]);

   printf("\n");

   return 0;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 6.15 Cómo ordenar un arreglo utilizando la ordenación tipo burbuja.

Primero el programa compara **a[0]** con **a[1]**, después **a[1]** con **a[2]**, a continuación **a[2]** con **a[3]**, y así en lo sucesivo, hasta completar la pasada, comparando **a[8]** con **a[9]**. Note que aunque existen 10 elementos, sólo se ejecutan nueve comparaciones. En razón de la forma en que se llevan a cabo las sucesivas comparaciones, en una pasada un valor grande puede pasar hacia abajo en el arreglo muchas posiciones, pero un valor pequeño pudiera moverse hacia arriba en una posición. En la primera pasada, el valor más grande está garantizado que se hundirá hasta el elemento más inferior del arreglo, **a[9]**. En la segunda pasada el segundo valor más grande está garantizado que se hundirá a **a[8]**. En la novena pasada, el valor noveno en tamaño se hundirá a **a[1]**. Esto dejará el tamaño más pequeño en **a[0]**, por lo que sólo se necesitan de nueve pasadas del arreglo para ordenarlo, aun cuando existan diez elementos.

La ordenación se ejecuta mediante el ciclo **for** anidado. Si un intercambio es necesario, es ejecutado mediante las tres asignaciones

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

donde la variable adicional **hold** almacena en forma temporal uno de los dos valores en intercambio. El intercambio no se puede ejecutar con sólo dos asignaciones

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Si, por ejemplo, **a[i]** es 7 y **a[i + 1]** es 5, después de la primera asignación ambos valores serán 5 y se perderá el valor 7. De ahí la necesidad de la variable adicional **hold**.

La virtud más importante de la ordenación tipo burbuja, es que es fácil de programar. Sin embargo, la ordenación tipo burbujas es de lenta ejecución. Esto se hace aparente al ordenar arreglos extensos. En los ejercicios, desarrollaremos versiones más eficientes de la ordenación tipo burbuja. Se han desarrollado ordenaciones mucho más eficientes que la clasificación tipo burbuja. Más adelante en el texto investigaremos unas cuantas de éstas. Cursos más avanzados investigan con mayor profundidad la ordenación y búsqueda.

6.7 Estudio de caso: cómo calcular el promedio, la mediana y el modo utilizando arreglos

Consideremos ahora un ejemplo mayor. Las computadoras se utilizan por lo común para compilar y analizar los resultados de encuestas y de encuestas de opinión. El programa de la figura 6.16 utiliza el arreglo **response** inicializado con 99 respuestas (representadas por la constante simbólica **SIZE**) de una encuesta. Cada una de las respuestas es un número del 1 al 9. El programa calcula el promedio, la mediana y el modo de los 99 valores.

El promedio es el promedio aritmético de los 99 valores. La función **mean** calcula el promedio totalizando los 99 elementos y dividiendo el resultado entre 99.

La mediana es el "valor medio". La función **median** determina la mediana llamando a la función **bubbleSort** para ordenar el arreglo de respuestas en orden ascendente, y seleccionando el elemento medio, **answer [SIZE / 2]**, del arreglo ya ordenado. Note que cuando existe un número par de elementos, la mediana deberá ser calculada como el promedio de los dos elementos de en medio. Esta versión de la función **median** no proporciona esta capacidad. La función **printArray** es llamada para la salida del arreglo **response**.

El modo es el valor que ocurre con mayor frecuencia entre las 99 respuestas. La función **mode** determina el modo, contando el número de respuestas de cada tipo, y a continuación escogiendo

el valor que tenga el conteo mayor. Esta versión de la función **mode** no maneja un empate (vea el ejercicio 6.14). La función **mode** también produce un histograma, para auxiliar en la determinación gráfica del modo. En la figura 6.17 se da una ejecución de muestra de este programa. Este ejemplo incluye la mayor parte de las manipulaciones normales requeridas con frecuencia en los problemas de arreglos, incluyendo el pasar arreglos a funciones.

```
/* This program introduces the topic of survey data analysis.
   It computes the mean, median, and  mode of the data */
#include <stdio.h>
#define SIZE 99

void mean(int []);
void median(int []);
void mode(int [], int []);
void bubbleSort(int[]);
void printArray(int[]);

main()
{
   int frequency[10] = {0},
       response[SIZE] = {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
                         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
                         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
                         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
                         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
                         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
                         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
                         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
                         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
                         4, 5, 6, 1, 6, 5, 7, 8, 7};
   mean(response);
   median(response);
   mode(frequency, response);
   return 0;
}

void mean(int answer[])
{
   int j, total = 0;

   printf("%s\n%s\n%s\n", "*********", "  Mean", "*********");

   for (j = 0; j <= SIZE - 1; j++)
      total += answer[j];

   printf("The mean is the average value of the data\n"
          "items. The mean is equal to the total of\n"
          "all the data items divided by the number\n"
          "of data items (%d). The mean value for\n"
          "this run is: %d / %d = %.4f\n\n",
          SIZE, total, SIZE, (float) total / SIZE);
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 1 de 3).


```
void median(int answer[])
{
   printf("\n%s\n%s\n%s\n%s",
          "*********", " Median", "*********",
          "The unsorted array of responses is");

   printArray(answer);
   bubbleSort(answer);
   printf("\n\nThe sorted array is");
   printArray(answer);
   printf("\n\nThe median is element %d of\n"
          "the sorted %d element array.\n"
          "For this run the median is %d\n\n",
          SIZE / 2, SIZE, answer[SIZE / 2]);
}

void mode(int freq[], int answer[])
{
   int rating, j, h, largest = 0, modeValue = 0;

   printf("\n%s\n%s\n%s\n",
          "*********", "  Mode", "*********");

   for (rating = 1; rating <= 9; rating++)
      freq[rating] = 0;

   for (j = 0; j <= SIZE - 1; j++)
      ++freq[answer[j]];

   printf("%s%11s%19s\n\n%54s\n%54s\n\n",
          "Response", "Frequency", "Histogram",
          "1    1    2    2", "5    0    5    0    5");

   for (rating = 1; rating <= 9; rating++) {
      printf("%8d%11d          ", rating, freq[rating]);

      if (freq[rating] > largest) {
         largest = freq[rating];
         modeValue = rating;
      }

      for (h = 1; h <= freq[rating]; h++)
         printf("*");

      printf("\n");
   }

   printf("The mode is the most frequent value.\n"
          "For this run the mode is %d which occurred"
          " %d times.\n", modeValue, largest);
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 2 de 3).

```
void bubbleSort(int a[])
{
   int pass, j, hold;

   for (pass = 1; pass <= SIZE - 1; pass++)

      for (j = 0; j <= SIZE - 2; j++)

         if (a[j] > a[j+1]) {
            hold = a[j];
            a[j] = a[j+1];
            a[j+1] = hold;
         }
}

void printArray(int a[])
{
   int j;

   for (j = 0; j <= SIZE - 1; j++) {

      if (j % 20 == 0)
         printf("\n");

      printf("%2d", a[j]);
   }
}
```

Fig. 6.16 Programa de análisis de datos de encuesta (parte 3 de 3).

6.8 Búsqueda en arreglos

A menudo, un programador estará trabajando con grandes cantidades de datos almacenados en arreglos. Pudiera resultar necesario determinar si un arreglo contiene un valor que coincide con algún *valor clave o buscado*. El proceso de encontrar en un arreglo un elemento en particular, se llama *búsqueda*. En esta sección se analizan dos técnicas de búsqueda —la técnica simple de *búsqueda lineal* y la técnica más eficiente de *búsqueda binaria*. Los ejercicios 6.34 y 6.35 al final de este capítulo le piden que diseñe versiones recursivas, tanto de la búsqueda lineal como de la búsqueda binaria.

La búsqueda lineal (figura 6.18) compara cada uno de los elementos del arreglo con la *el valor buscado*. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentre, ya sea en el primer elemento como en el último. Por lo tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.

El método de búsqueda lineal funciona bien para arreglos pequeños o para arreglos no ordenados. Sin embargo, para la búsqueda de arreglos extensos, el sistema lineal es ineficiente. Si el arreglo está ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria.

El algoritmo de búsqueda binaria, después de cada una de las comparaciones, elimina la mitad de los elementos en el arreglo bajo búsqueda. El algoritmo localiza el elemento medio del arreglo y lo compara con el valor buscado. Si son iguales, la clave de búsqueda ha sido encontrada y se


```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788

********
  Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response   Frequency              Histogram

                                    1  1  2  2
                                 5  0  5  0  5

       1          1        *
       2          3        ***
       3          4        ****
       4          5        *****
       5          8        ********
       6          9        *********
       7         23        ***********************
       8         27        ***************************
       9         19        *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 6.17 Ejecución de muestra del programa de análisis de datos de encuesta.

```
/* Linear search of an array */
#include <stdio.h>
#define SIZE 100

int linearSearch(int [], int, int);

main()
{
   int a[SIZE], x, searchKey, element;

   for (x = 0; x <= SIZE - 1; x++)  /* create some data */
      a[x] = 2 * x;

   printf("Enter integer search key:\n");
   scanf("%d", &searchKey);
   element = linearSearch(a, searchKey, SIZE);

   if (element != -1)
      printf("Found value in element %d\n", element);
   else
      printf("Value not found\n");

   return 0;
}

int linearSearch(int array[], int key, int size)
{
   int n;

   for (n = 0; n <= size - 1; ++n)
      if (array[n] == key)
         return n;

   return -1;
}
```

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

Fig. 6.18 Búsqueda lineal en un arreglo.

regresa el subíndice del arreglo correspondiente a dicho elemento. Si no son iguales, el problema se reduce a buscar en una mitad del arreglo. Si el valor buscado es menor que el elemento medio del arreglo, se seguirá buscando en la primera parte del arreglo, de lo contrario se buscará en la

segunda parte. Si el valor buscado no se encuentra en el subarreglo especificado (es la porción del arreglo original), el algoritmo se repite en una cuarta parte del arreglo original. La búsqueda continúa, hasta que el valor buscado es igual al elemento del medio del subarreglo, o hasta que el subarreglo ha quedado reducido a un elemento diferente a el valor buscado (es decir, el valor buscado no ha sido encontrado).

En el peor escenario, utilizando la búsqueda binaria, la búsqueda de un arreglo de 1024 elementos sólo tomará 10 comparaciones. La división repetida de 1024 entre 2, da los valores 512, 256, 128, 64, 32, 16, 8, 4, 2 y 1. El número 1024 (2^{10}) se divide entre 2 sólo diez veces para obtener el valor de 1. En el algoritmo de búsqueda binaria la división por 2 es equivalente a una comparación. Un arreglo de 1048576 (2^{20}) elementos toma un máximo de 20 comparaciones, para encontrar el valor buscado. Un arreglo de mil millones de elementos, toma un máximo de 30 comparaciones para encontrar el valor buscado. Esto es un incremento tremendo en rendimiento en comparación con la búsqueda lineal, que en promedio requería la comparación de el valor buscado en la mitad de los elementos del arreglo. ¡En el caso de un arreglo de mil millones de elementos, es una diferencia entre un promedio de 500 millones de comparaciones y un máximo de 30! Las comparaciones máximas para cualquier arreglo pueden ser determinadas encontrando la primera potencia de 2 mayor que el número de elementos en el arreglo.

En la figura 6.19 se presenta la versión iterativa de la función **binarySearch**. La función recibe cuatro argumentos —un arreglo entero **b**, un entero **searchKey**, el subíndice de arreglo **low** y el subíndice de arreglo **high**. Si el valor buscado no coincide con el elemento medio de un subarreglo, el subíndice **low** o el subíndice **high** es modificado, de tal forma que pueda buscarse en un subarreglo más pequeño. Si el valor buscado es menor que el elemento medio, el subíndice **high** se define como **middle - 1**, y la búsqueda se continúa sobre los elementos desde **low** hasta **middle - 1**. Si el valor buscado es mayor que el elemento medio, el subíndice **low** se define como **middle + 1** y la búsqueda se continúa sobre los elementos desde **middle + 1** hasta **high**. El programa utiliza un arreglo de 15 elementos. La primera potencia de 2 mayor que el número de elementos en este arreglo es 16 (2^4), por lo que se requerirán un máximo de 4 comparaciones para encontrar el valor buscado. El programa utiliza la función **printHeader** para sacar los subíndices del arreglo y la función **printRow** para sacar cada subarreglo durante el proceso de búsqueda binaria. El elemento medio de cada subarreglo queda marcado con un asterisco (*) para indicar cuál es el elemento con el cual el valor buscado se compara.

6.9 Arreglos con múltiples subíndices

En C los arreglos pueden tener múltiples subíndices. Una utilización común de los arreglos con múltiples subíndices es la representación de *tablas* de valores, consistiendo de información arreglada en *renglones* y *columnas*. Para identificar un elemento particular de la tabla, deberemos especificar dos subíndices; el primero (por regla convencional) identifica el renglón del elemento, y el segundo (también por regla convencional) identifica la columna del elemento. Tablas o arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como *arreglos de doble subíndice*. Note que los arreglos de múltiples subíndices pueden tener más de dos subíndices. El estándar ANSI indica que un sistema ANSI C debe soportar por lo menos 12 subíndices de arreglo.

En la figura 6.20 se ilustra un arreglo de doble subíndice, **a**. El arreglo contiene tres renglones y cuatro columnas, por lo que se dice que se trata de un arreglo de 3 por 4. En general, un arreglo con *m* renglones y *n* columnas se llama un *arreglo de m por n*.

```
/* Binary search of an array */

#include <stdio.h>
#define SIZE 15

int binarySearch(int [], int, int, int);
void printHeader(void);
void printRow(int [], int, int, int);


main()
{
   int a[SIZE], i, key, result;

   for (i = 0; i <= SIZE - 1; i++)
      a[i] = 2 * i;

   printf("Enter a number between 0 and 28: ");
   scanf("%d", &key);

   printHeader();
   result = binarySearch(a, key, 0, SIZE - 1);

   if (result != -1)
      printf("\n%d found in array element %d\n", key, result);
   else
      printf("\n%d not found\n", key);

   return 0;
}


int binarySearch(int b[], int searchKey, int low, int high)
{
   int middle;

   while (low <= high) {
      middle = (low + high) / 2;

      printRow(b, low, middle, high);

      if (searchKey == b[middle])
         return middle;
      else if (searchKey < b[middle])
         high = middle - 1;
      else
         low = middle + 1;
   }

   return -1;    /* searchKey not found */
}
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 1 de 3).


```
/* Print a header for the output */
void printHeader(void)
{
   int i;

   printf("\nSubscripts:\n");

   for (i = 0; i <= SIZE - 1; i++)
      printf("%3d ", i);

   printf("\n");

   for (i = 1; i <= 4 * SIZE; i++)
      printf("-");

   printf("\n");
}

/* Print one row of output showing the current
   part of the array being processed. */
void printRow(int b[], int low, int mid, int high)
{
   int i;

   for (i = 0; i <= SIZE - 1; i++)
      if (i < low || i > high)
         printf("    ");
      else if (i == mid)
         printf("%3d*", b[i]);  /* mark middle value */
      else
         printf("%3d ", b[i]);

   printf("\n");
}
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 2 de 3).

Cada uno de los elementos en el arreglo **a** está identificado en la figura 6.20 por un nombre de elemento de la forma **a[i][j]**; **a** es el nombre del arreglo, e **i** y **j** son los subíndices que identifican en forma única a cada elemento dentro de **a**. Note que los nombres de los elementos en el primer renglón, todos tienen un primer subíndice de **0**; los nombres de los elementos en la cuarta columna, todos tienen un segundo subíndice de **3**.

Error común de programación 6.9

Referenciar un elemento de arreglo de doble subíndice como **a[x][y]** *en vez de* **a[x, y]**.

Un arreglo de múltiple subíndice puede ser inicializado en su declaración en forma similar a un arreglo de un subíndice. Por ejemplo, un arreglo de doble subíndice **b[2][2]** podría ser declarado e inicializado con

```
int b[2][2] = {{1, 2}, {3, 4}};
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                              16  18  20  22* 24  26  28
                                          24  26* 28
                                          24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6

Suscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Fig. 6.19 Búsqueda binaria de un arreglo ordenado (parte 3 de 3).

Los valores se agrupan por renglones entre llaves. Por lo tanto, `1` y `2` inicializan `b[0][0]` y `b[0][1]`, `3` y `4` inicializan `b[1][0]` y `b[1][1]`. Si para un renglón dado no se proporcionan suficientes inicializadores, los elementos restantes de dicho renglón se inicializarán a `0`. Por lo tanto, la declaración

```
int b[2][2] = {{1}, {3, 4}};
```

inicializaría `b[0][0]` a `1`, `b[0][1]` a `0`, `b[1][0]` a `3` y `b[1][1]` a `4`.

En la figura 6.21 se demuestra la inicialización de arreglos de doble subíndice en declaraciones. El programa declara tres arreglos de dos renglones y tres columnas (de seis elementos cada

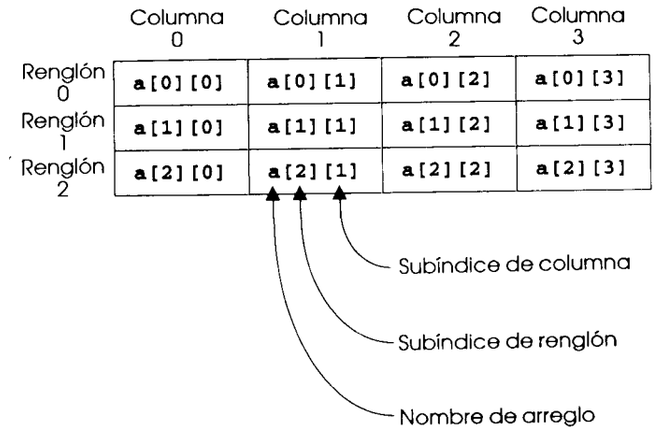

Fig. 6.20 Un arreglo de doble subíndice con tres renglones y cuatro columnas.

uno de ellos). La declaración de `array1` proporciona seis inicializadores en dos sublistas. La primera sublista incializa el primer renglón del arreglo a los valores 1, 2 y 3; y la segunda sublista inicializa el segundo renglón del arreglo a los valores 4, 5, y 6. Si de la lista de inicialización `array1` se retiran los parentesis ( ) de alrededor de cada sublista, el compilador de forma automática inicializa los elementos del primer renglón seguido por los elementos del segundo. La declaración de `array1` proporciona cinco inicializadores. Los inicializadores son asignados al primer renglón y a continuación al segundo. Cualquier elemento que no tenga un inicializador explícito es inicializado en forma automática a cero, por lo que `array2[1][2]` es inicializado a 0. La declaración de `array3` proporciona tres inicializadores en dos sublistas. La sublista para el primer renglón inicializa de manera explícita los dos primeros elementos del primer renglón a 1 y a 2. El tercer elemento automáticamente queda inicializado a cero. La sublista para el segundo renglón inicializa explícitamente el primer elemento a 4. Los dos elementos siguientes automáticamente se inicializan a cero.

El programa llama a la función `printArray` para que se impriman los elementos de cada arreglo. Note que la definición de la función especifica el parámetro de arreglo como `int a[][3]`. Cuando recibimos un arreglo de un subíndice como argumento a una función, los corchetes del arreglo están vacíos en la lista de parámetros de la función. El primer subíndice de un arreglo de múltiples subíndices tampoco se requiere, pero todos los demás subíndices son requeridos. El compilador utiliza estos subíndices para determinar las localizaciones en memoria de los elementos en arreglos de múltiples subíndices. Todos los elementos del arreglo son almacenados en memoria de forma consecutiva, independiente del número de subíndices. En un arreglo de doble subíndice, el primer renglón se almacena en memoria y a continuación se almacena el segundo.

Proporcionar los valores de subíndices en una declaración de parámetros le permite al compilador indicarle a la función cómo localizar un elemento del arreglo. En un arreglo de doble subíndice, cada renglón es básicamente un arreglo de un solo subíndice. Para localizar un elemento en un renglón en particular, el compilador debe saber con exactitud cuántos elementos existen en cada renglón, de tal forma que pueda saltar la cantidad apropiada de posiciones de memoria para llegar al arreglo. Por lo tanto, al tener acceso a `a[1][2]` en nuestro ejemplo, el compilador sabe

```
/* Initializing multidimensional arrays */
#include <stdio.h>

void printArray(int [][3]);

main()
{
   int array1[2][3] = { {1, 2, 3}, {4, 5, 6} },
       array2[2][3] = { 1, 2, 3, 4, 5 },
       array3[2][3] = { {1, 2}, {4} };

   printf("Values in array1 by row are:\n");
   printArray(array1);

   printf("Values in array2 by row are:\n");
   printArray(array2);

   printf("Values in array3 by row are:\n");
   printArray(array3);

   return 0;
}

void printArray(int a[][3])
{
   int i, j;

   for (i = 0; i <= 1; i++) {

      for (j = 0; j <= 2; j++)
         printf("%d ", a[i][j]);;

      printf("\n");
   }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 6.21 Cómo inicializar arreglos multidimensionales.

que debe de saltarse los tres elementos del primer renglón en la memoria para obtener el segundo renglón (renglón 1). A continuación, el compilador llega al tercer elemento de dicho renglón (elemento 2).

Muchas manipulaciones comunes con arreglos utilizan estructuras de repetición **for**. Por ejemplo, la siguiente estructura define todos los elementos en el tercer renglón del arreglo **a** en la figura 6.20 a cero:

```
for (column = 0; column  3; column++)
   a[2][column] = 0;
```

Especificamos el *tercer* renglón, por lo tanto, sabemos que el primer subíndice será siempre **2** (**0** es el primer renglón y **1** el segundo). El ciclo **for** varía sólo en el segundo subíndice (es decir, el subíndice de columnas). La estructura **for** anterior es equivalente a los enunciados de asignación siguientes:

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

En el arreglo **a**, la siguiente estructura **for** anidada determina el total de todos los elementos.

```
total = 0;

for (row = 0; row <= 2; row++)

for (column = 0; column <= 3; column++)
   total += a [row][column];
```

La estructura **for** totaliza los elementos del arreglo, renglón por renglón. La estructura externa **for** empieza estableciendo **row** (es decir, el subíndice de renglón) a **0**, de tal forma que los elementos del primer renglón puedan ser totalizados por la estructura interna **for**. La estructura externa **for** incrementa **row** a **1**, de tal forma que los elementos del segundo renglón sean totalizados. Por último, la estructura externa **for** incrementa **row** a **2**, para que se totalicen los elementos del tercer renglón. El resultado se imprime cuando la estructura **for** anidada termina.

El programa de la figura 6.22 ejecuta otras manipulaciones comunes de arreglos en un arreglo de 3 por 4 **studentGrades**, utilizando estructuras **for**. Cada renglón del arreglo representa un alumno y cada columna representa una calificación, en uno de los cuatro exámenes que los alumnos pasaron durante el semestre. Las manipulaciones de arreglo se ejecutan mediante cuatro funciones. La función **minimum** determina el grado más bajo de cualquier alumno para el semestre. La función **maximum** determina la calificación más alta de cualquier alumno en el semestre. La función **average** determina el promedio para el semestre de un alumno en particular. La función **printArray** extrae la salida del arreglo de doble subíndice en un formato tabular nítido.

Las funciones **minimum**, **maximum** y **printArray** cada una de ellas recibe tres argumentos —el arreglo **studentGrades** (denominadas **grades** en cada función), el número de alumnos (renglones del arreglo), y el número de exámenes (columnas del arreglo). Cada función cicla a través del arreglo **grades**, utilizando estructuras **for** anidadas. La siguiente estructura **for** anidada corresponde a la función de definición **minimum**:

```
for (i = 0; i <= pupils - 1; i++)
   for (j = 0; j <= tests - 1; j++)
      if (grades[i][j] < lowGrade)
         lowGrade = grades [i][j];
```

```
/* Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

int minimum(int [][EXAMS], int, int);
int maximum(int [][EXAMS], int, int);
float average(int [], int);
void printArray(int [][EXAMS], int, int);

main()
{
   int student,
       studentGrades[STUDENTS][EXAMS] = {{77, 68, 86, 73},
                                         {96, 87, 89, 78},
                                         {70, 90, 86, 81}};

   printf("The array is:\n");
   printArray(studentGrades, STUDENTS, EXAMS);
   printf("\n\nLowest grade: %d\nHighest grade: %d\n",
          minimum(studentGrades, STUDENTS, EXAMS),
          maximum(studentGrades, STUDENTS, EXAMS));

   for (student = 0; student <= STUDENTS - 1; student++)
      printf("The average grade for student %d is %.2f\n",
             student, average(studentGrades[student], EXAMS));

   return 0;
}

/* Find the minimum grade */
int minimum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, lowGrade = 100;

   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] < lowGrade)
            lowGrade = grades[i][j];

   return lowGrade;
}

/* Find the maximum grade */
int maximum(int grades[][EXAMS], int pupils, int tests)
{
   int i, j, highGrade = 0;

   for (i = 0; i <= pupils - 1; i++)
      for (j = 0; j <= tests - 1; j++)
         if (grades[i][j] > highGrade)
            highGrade = grades[i][j];

   return highGrade;
}
```

Fig. 6.22 Ejemplo del uso de arreglos con doble subíndice (parte 1 de 2).


```
/* Determine the average grade for a particular exam */
float average(int setOfGrades[], int tests)
{
   int i, total = 0;

   for (i = 0; i <= tests - 1; i++)
      total += setOfGrades[i];

   return (float) total / tests;
}

/* Print the array */
void printArray(int grades[][EXAMS], int pupils, int tests)
{
   int i, j;

   printf("                  [0]  [1]  [2]  [3]");

   for (i = 0; i <= pupils - 1; i++) {
      printf("\nstudentGrades[%d] ", i);

      for (j = 0; j <= tests - 1; j++)
         printf("%-5d", grades[i][j]);
   }
}
```

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Fig. 6.22 Ejemplo del uso de arreglos con doble subíndice (parte 2 de 2).

La estructura externa **for** empieza definiendo a **i** (es decir, el subíndice de renglón) en **0**, de tal forma que los elementos del primer renglón puedan ser comparados a la variable **lowGrade** en el cuerpo de la estructura interna **for**. La estructura interna **for** cicla a través de las cuatro calificaciones de un renglón en particular y compara cada una de las calificaciones con **lowGrade**. Si una calificación es menor que **lowGrade**, **lowGrade** se define igual a tal calificación. A continuación la estructura externa **for** incrementa el subíndice de renglón a **1**. El elemento del segundo renglón se compara a la variable **lowGrade**. Después la estructura **for** externa incrementa el subíndice de renglón a **2**. Los elementos del tercer renglón son comparados con la variable **lowGrade**. Cuando la ejecución de la estructura anidada está completa, **lowGrade** contiene la calificación más pequeña del arreglo de doble índice. La función **maximum** funciona de manera similar a la función **minimum**.

La función **average** toma dos argumentos —un arreglo de un solo índice de los resultados de las pruebas para un alumno en particular, llamada **setOfGrades** y el número de resultados de pruebas del arreglo. Cuando se llama a **average**, el primer argumento que se pasa es **studentGrades[student]**. Esto hace que la dirección de un renglón del arreglo de doble subíndice sea pasado a **average**. El argumento **studentGrades[1]** es la dirección inicial del segundo renglón del arreglo. Recuerde que un arreglo de doble subíndice es básicamente un arreglo de arreglos de un índice, y que el nombre de un arreglo de un solo índice es la dirección del arreglo en memoria. La función **average** calcula la suma de los elementos del arreglo, divide el total por el número de resultados de pruebas, y regresa un resultado en punto flotante.

Resumen

- C almacena listas de valores en arreglos. Un arreglo es un grupo de posiciones relacionadas en memoria. Estas posiciones están relacionadas por el hecho de que todas tienen el mismo nombre y son del mismo tipo. Para referirse a una posición particular o algún elemento dentro del arreglo, especificamos el nombre del arreglo y el subíndice.
- Un subíndice puede ser un entero o una expresión de enteros. Si un programa utiliza como subíndice una expresión, entonces la expresión se evalúa para determinar el elemento particular del arreglo.
- Es importante notar la diferencia entre referirse al séptimo elemento del arreglo y referirse al elemento siete del arreglo. El séptimo elemento tiene un subíndice de **6**, en tanto que el elemento siete del arreglo tiene un subíndice de **7** (y de hecho dentro del arreglo es el octavo elemento). Esto es una fuente de errores "por diferencia de uno".
- Los arreglos ocupan espacio en memoria. Para reservar 100 elementos para el arreglo entero **b** y 27 elementos para el arreglo entero **x**, el programador escribe

```
int b[100], x[27];
```

- Un arreglo del tipo **char** puede ser utilizado para almacenar una cadena de caracteres.
- Los elementos de un arreglo pueden ser inicializados de tres formas distintas: por declaración, por asignación y por entrada.
- Si existen menos inicializadores que elementos en el arreglo, C inicializará de forma automática a cero los elementos restantes.
- C no evita la referenciación de elementos más allá de los límites de un arreglo.
- Un arreglo de caracteres puede ser inicializado utilizando una literal de cadena.
- Todas las cadenas en C terminan con un carácter nulo. La representación de la constante de carácter nulo, es **'\0'**.
- Los arreglos de caracteres pueden ser inicializados en la lista de inicialización con constantes de caracteres.
- Los caracteres individuales en una cadena almacenada en un arreglo pueden ser accesibles de forma directa utilizando la notación de subíndice de arreglos.
- Se puede introducir de manera directa una cadena en un arreglo de caracteres desde el teclado utilizando **scanf** y la especificación de conversión **%s**.

- Un arreglo de caracteres que represente una cadena puede ser extraído utilizando **printf** y el especificador de conversión **%s**.
- Aplique **static** a una declaración de arreglo local a fin de que el arreglo no tenga que ser creado cada vez que se llame a la función y cada vez que la función termine no sea destruido el arreglo.
- Los arreglos declarados **static** se inicializan de forma automática una vez en tiempo de compilación. Si el programador no inicializa explícitamente un arreglo **static**, será inicializado a cero por el compilador.
- Para pasar un arreglo a una función, se pasa el nombre del arreglo. Para pasar un elemento de un arreglo a una función, sólo pase el nombre del arreglo seguido por el subíndice (contenido entre corchetes) del elemento particular.
- C pasa arreglo a las funciones utilizando simulación de llamada por referencias —las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de los llamadores. ¡El nombre del arreglo es de hecho la dirección del primer elemento del arreglo! Dado que es pasada la dirección de inicio del arreglo, la función llamada sabe precisamente donde dicho arreglo está almacenado.
- Para recibir un argumento de arreglo, la lista de parámetros de la función debe especificar que un arreglo será recibido. En los corchetes del arreglo no es requerido el tamaño del mismo.
- La especificación de conversión **%p** por lo regular extrae direcciones en números hexadecimales.
- C proporciona un calificador especial de tipo **const** para impedir la modificación de los valores de arreglos en una función. Cuando un parámetro de arreglo está precedido de un calificador **const**, los elementos del arreglo se convierten en constantes dentro del cuerpo de la función, y cualquier intento para modificar un elemento del arreglo en el cuerpo de la función, dan como resultado un error en tiempo de compilación.
- Un arreglo puede ser ordenado utilizando la técnica de oredenación tipo burbuja. Se hacen varias pasadas del arreglo. En cada una de las pasadas, se comparan pares sucesivos de elementos. Si un par está en orden (o si sus valores son idénticos), se deja tal como está. Si el par está fuera de orden, los valores son intercambiados. Para pequeños arreglos, la ordenación tipo burbuja es aceptable, pero es ineficiente en caso de arreglos mayores, y en comparación con otros algoritmos de ordenación más avanzados.
- La búsqueda lineal compara cada elemento del arreglo con el valor buscado. Dado que el arreglo no está en ningún orden en particular, existe igual probabilidad de que el valor sea encontrado tanto en el primer elemento como en el último. En promedio, por lo tanto, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo. El método de búsqueda lineal funciona bien para arreglos pequeños o para arreglos no clasificados.
- El algoritmo de búsqueda binaria elimina la mitad de los elementos del arreglo bajo búsqueda después de cada comparación. El algoritmo localiza el elemento medio del arreglo y lo compara con el valor buscado. Si son iguales, se ha encontrado el valor buscado y el subíndice del arreglo de dicho elemento se regresa. Si no son iguales, el problema queda reducido a volver a buscar en una de las mitades del arreglo.
- En el peor escenario, la búsqueda mediante la búsqueda binaria de un arreglo de 1024 elementos tomará sólo 10 comparaciones. Un arreglo de 1048576(2^{20}) elementos toma un máximo de 20 comparaciones para encontrar el valor buscado. Un arreglo de mil millones de elementos toma un máximo de 30 comparaciones para encontrar el valor buscado.

- Los arreglos pueden ser utilizados para representar tablas de valores, consistiendo de información arreglada en renglones y columnas. Para identificar un elemento particular de una tabla, se especifican dos subíndices: el primero (por regla convencional) identifica el renglón en el cual se contiene el elemento, y el segundo (por regla convencional) identifica la columna en el cual el elemento está contenido. Las tablas de arreglo que requieran de dos subíndices para identificar un elemento en particular, se conocen como arreglos de doble subíndice.
- El estándar indica que un sistema ANSI C debe de soportar por lo menos 12 subíndices de arreglo.
- Un arreglo de múltiples subíndices puede ser inicializado en su declaración utilizando una lista inicializadora.
- Cuando recibimos un arreglo de un subíndice como argumento de una función, los corchetes del arreglo están vacíos en la lista de parámetros de la función. Tampoco se requiere el primer subíndice de un arreglo de múltiples subíndices, pero todos los subíndices subsecuentes son requeridos. El compilador utilizará estos subíndices para determinar las posiciones en memoria de los elementos en los arreglos de múltiples subíndices.
- Para pasar un renglón de un arreglo de doble subíndice a una función que recibe un arreglo de un subíndice, sólo pase el nombre del arreglo, seguido por el primer subíndice.

Terminología

a[i]
a[i][j]
arreglo
lista de inicialización de arreglo
gráfica de barras
verificación de límites
ordenación tipo burbuja
subíndice de columna
declarar un arreglo
directiva de preprocesador `# define`
doble precisión
arreglo de doble subíndice
elemento de un arreglo
expresión como un subíndice
histograma
inicializar un arreglo
búsqueda lineal
arreglo m por n
promedio
mediana
modo
arreglo de múltiples subíndices
nombre de un arreglo
carácter nulo `'\0'`
error por diferencia de uno
pasar por referencia
pasar arreglos a funciones
especificación de conversión %p
número de posición
texto de remplazo
subíndice de renglón
dimensionabilidad
escalar
cantidad escalar
valor buscado
búsqueda en un arreglo
arreglo de un subíndice
ordenación por hundimiento
ordenación
pasada de ordenación
ordenación de los elementos de un arreglo
corchetes
cadena
subíndice
análisis de datos de encuesta
constante simbólica
tabla de valores
formato tabular
área temporal para intercambio de valores
totalizar los elementos de un arreglo
arreglo de triple subíndice
valor de un elemento
salirse de un arreglo
elemento de orden cero

Errores comunes de programación

6.1 Es importante notar la diferencia entre "el séptimo elemento del arreglo" "y el elemento siete del arreglo". Dado que los subíndices de los arreglos empiezan en 0, "el séptimo elemento del arreglo" tiene un subíndice de 6, en tanto que el elemento 7 del arreglo "tiene un subíndice de siete" y de hecho es el octavo elemento del arreglo. Este es una fuente de un error por "diferencia de uno".
6.2 Olvidar inicializar los elementos de un arreglo cuyos elementos deban de estar inicializados.
6.3 El proporcionar más inicializadores en una lista inicializadora de arreglo que elementos existan dentro del mismo constituye un error de sintaxis.
6.4 Terminar una directiva de preprocesador `#define`, o bien `#include` con un punto y coma. Recuerde que las directivas de preprocesador no son enunciados C.
6.5 Asignar un valor a una constante simbólica en un enunciado ejecutable es un error de sintaxis. Una constante simbólica no es una variable. El compilador no reserva espacio para ella por como hace con las variables que contienen valores durante la ejecución.
6.6 Referirse a un elemento exterior a los límites del arreglo.
6.7 No proporcionar, en un programa, a `scanf` un arreglo de caracteres lo suficiente grande para almacenar una cadena escrita en el teclado, puede dar como resultado una pérdida de datos, así como otros errores en tiempo de ejecución.
6.8 Suponer que los elementos de un arreglo local, que está declarado como `static`, están inicializados a cero, cada vez que la función sea llamada donde se declara el arreglo.
6.9 Referenciar un elemento de arreglo de doble subíndice como `a[x][y]` en vez de `a[x, y]`.

Prácticas sanas de programación

6.1 Utilice sólo letras mayúsculas para los nombres de constantes simbólicas. Esto hace que estas constantes resalten en un programa y le recuerden al programador que las constantes simbólicas no son variables.
6.2 Busque claridad en el programa. Algunas veces será preferible sacrificar la utilización más eficiente de memoria o de tiempo de procesador en aras de escribir programas más claros.
6.3 Al ciclar a través de un arreglo, el subíndice de un arreglo no debe de pasar nunca por debajo de 0 y siempre tiene que ser menor que el número total de elementos del arreglo (tamaño -1). Asegúrese que la condición de terminación del ciclo impide el acceso a elementos fuera de este rango.
6.4 Mencione el subíndice alto del arreglo en una estructura `for`, a fin de ayudar a eliminar los errores por diferencia de uno.
6.5 Los programas deberían verificar la corrección de todos los valores de entrada, para impedir que información errónea afecte los cálculos del programa.
6.6 Algunos programadores incluyen nombres de variables en las funciones prototipo, para que los programas resulten más claros. El compilador ignorará estos nombres.

Sugerencias de rendimiento

6.1 Algunas veces las consideraciones de rendimiento tienen mucho mayor importancia que las consideraciones de claridad.
6.2 Los efectos (por lo regular serios) de referirse a elementos fuera de los límites del arreglo, son dependientes del sistema.
6.3 En funciones que contengan arreglos automáticos donde la función entra y sale de alcance con frecuencia, haga `static` dicho arreglo, de tal forma que no sea necesario crearlo cada vez que la función sea llamada.
6.4 Tiene sentido pasar arreglos simulando llamadas por referencia por razones de rendimiento. Si los arreglos fueran pasados en llamadas por valor, se pasaría una copia de cada uno de los elementos. En el caso de arreglos grandes pasados con frecuencia, esto tomaría mucho tiempo y consumiría gran cantidad de espacio de almacenamiento para las copias de los arreglos.

6.5 A menudo, los algoritmos más sencillos tienen rendimientos pobres. Su virtud estriba en que son fáciles de escribir, de probar y de depurar. Sin embargo, para obtener rendimiento máximo a menudo se requiere de algoritmos más complejos.

Observaciones de ingeniería de software

6.1 Definir el tamaño de cada arreglo como una constante simbólica hace a los programas más dimensionables.

6.2 Es posible pasar un arreglo por valor utilizando una triquiñuela sencilla que explicaremos en el capítulo 10.

6.3 El calificador de tipo `const` puede ser aplicado a un parámetro de arreglo en una definición de función, para impedir que en el cuerpo de la función el arreglo original sea modificado. Este es otro ejemplo del principio de mínimo privilegio. Las funciones no deben tener, ni se les dará, capacidad de modificar un arreglo, a menos de que sea en lo absoluto necesario.

Ejercicios de autoevaluación

6.1 Llene cada uno de los siguientes espacios vacíos:

a) Las listas y las tablas de valores se almacenan en _____.
b) Los elementos de un arreglo están relacionados entre sí por el hecho de que tienen el mismo _____ y _____.
c) El número utilizado para referirnos a un elemento particular de un arreglo se llama su _____.
d) Debe utilizarse un _____ para declarar el tamaño de un arreglo porque el programa hace más dimensionable.
e) El proceso de colocar en orden los elementos de un arreglo se conoce como _____ el arreglo.
f) El proceso de determinar si un arreglo contiene un cierto valor buscado, se conoce como _____ el arreglo
g) Un arreglo que utilice dos subíndices se conoce como un arreglo de _____.

6.2 Declare si lo siguiente es verdadero o falso. Si la respuesta es falsa, explique por qué.

a) Un arreglo puede almacenar muchos tipos diferentes de valores.
b) Un subíndice de arreglo puede ser del tipo de datos `float`.
c) Si existen menos inicializadores en una lista inicializadora que el número de elementos dentro del arreglo, se inicializarán automáticamente C los elementos faltantes con el último valor en la lista de inicializadores.
d) Es un error si la lista de inicializadores contiene más inicializadores que existan elementos dentro del arreglo.
e) Un elemento individual de un arreglo que se pasa a una función y que fue modificado en la función llamada contendrá el valor modificado en la función llamada.

6.3 Conteste las preguntas siguientes relacionadas con un arreglo conocido como `fractions`.

a) Defina una constante simbólica `SIZE`, misma que será remplazada con el texto de remplazo 10.
b) Declare un arreglo con elementos `SIZE` del tipo `float`, e inicialice los elementos a `0`.
c) Dele nombre al cuarto elemento a partir del principio del arreglo.
d) Refiérase al elemento del arreglo 4.
e) Asigne el valor de `1.667` al elemento nueve del arreglo.
f) Asigne el valor `3.333` al séptimo elemento del arreglo.
g) Imprima los elementos 6 y 9 del arreglo con dos dígitos de precisión a la derecha del punto decimal, y muestre la salida que en realidad se despliega en pantalla.
h) Imprima todos los elementos del arreglo utilizando una estructura de repetición `for`. Suponga que la variable entera `x` ha sido definida como variable de control para el ciclo. Muestre la salida.

6.4 Conteste las siguientes preguntas en relación con un arreglo denominado `table`.

a) Declare el arreglo como un arreglo entero con 3 renglones y 3 columnas. Suponga que se ha definido la constante simbólica `SIZE` como de valor 3.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una estructura de repetición `for` para inicializar cada elemento del arreglo a la suma de sus subíndices. Suponga que las variables enteras `x` e `y` han sido declaradas como variables de control.
d) Imprima los valores de cada elemento del arreglo `table`. Suponga que el arreglo fue inicializado con la declaración,

```
int table[SIZE][SIZE] = {{1, 8}, {2, 4, 6}, {5}};
```

y las variables enteras `x` e `y` han sido declaradas como variables de control. Muestre la salida.

6.5 Encuentre el error en cada uno de los siguientes segmentos de programa y corrija el error.

a)
```
#define SIZE 100;
```
b)
```
SIZE = 10
```
c) Suponga
```
int b[10] = {0}, i;
for (i = 0; i <= 10; i++)
  b[i] = 1;
```
d)
```
#include <stdio.h>;
```
e) Suponga
```
int a[2][2] = {{1, 2}, {3, 4}};
a[1, 1] = 5;
```

Respuestas a los ejercicios de autoevaluación

6.1 a) Arreglos. b) Nombre, tipo. c) Subíndice. d) Constante simbólica. e) Ordenación. f) Búsqueda. g) Doble subíndice.

6.2 a) Falso. Un arreglo puede sólo almacenar valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Falso. C inicializa a cero en forma automática los elementos restantes.
d) Verdadero.
e) Falso. Los elementos individuales de un arreglo son pasados en llamada por valor. Si todo el arreglo se pasa a una función, entonces cualesquiera modificaciones se verán reflejadas en el original.

6.3 a)
```
#define SIZE 10
```
b)
```
Float fractions[SIZE] = {0};
```
c)
```
fractions[3]
```
d)
```
fractions[4]
```
e)
```
fractions[9] = 1.667;
```
f)
```
fractions[6] = 3.333;
```
g)
```
printf("%2f %.2f\n", fractions[6], fractions[9]);
```
Salida: `3.33 1.67.`

h)
```
for (x = 0; x <= SIZE - 1; x++)
  printf("fractions[%d] = %f\n", x, fractions[x]);
```
Salida:
```
fractions[0] = 0.000000
fractions[1] = 0.000000
fractions[2] = 0.000000
fractions[3] = 0.000000
fractions[4] = 0.000000
fractions[5] = 0.000000
fractions[6] = 3.333000
fractions[7] = 0.000000
```

```
fractions[8] = 0.000000
fractions[9] = 1.667000
```

6.4 a) `int table[SIZE][SIZE];`

b) Nueve elementos

c)
```
for (x = 0; x <= SIZE - 1; x++);
   for (y = 0; y <= SIZE - 1; y++)
      table[x][y] = x + y;
```

d)
```
for (x = 0; x <= SIZE - 1; x++)
   for (y = 0; y <= SIZE - 1; y++)
      printf("table[%d][%d] = %d\n", x, y, table[x][y]);0
```

Salida:
```
table[0][0] = 1
table[0][1] = 8
table[0][2] = 0
table[1][0] = 2
table[1][1] = 4
table[1][2] = 6
table[2][0] = 5
table[2][1] = 0
table[2][2] = 0
```

6.5 a) Error: punto y coma al final de la directiva de preprocesador `#define`.
Corrección: Eliminar el punto y coma.

b) Error: asignar un valor a una constante simbólica mediante un enunciado de asignación.
Corrección: asignar un valor a una constante simbólica en una directiva de preprocesador `#define` sin utilizar el operador de asignación como en `#define SIZE 10`.

c) Error: referenciar cualquier elemento del arreglo fuera de los límites del mismo (`b[10]`).
Corrección: cambie el valor final de la variable de control a `9`.

d) Error: punto y coma al final de la directiva de preprocesador `#include`.
Corrección: eliminar el punto y coma.

e) Error: la subindexación del arreglo se ha hecho en forma incorrecta.
Corrección: cambie el enunciado a `a[1][1] = 5;`

Ejercicios

6.6 Llene cada uno de los siguientes espacios vacíos:

a) C almacena listas de valores en _____.

b) Los elementos de un arreglo están relacionados por el hecho de que son _____.

c) Al referirse a un elemento de un arreglo, el número de posición contenido entre paréntesis se conoce como _____.

d) Los nombres de los cinco elementos de un arreglo `p` son _____, _____, _____, _____, y _____.

e) El contenido de un elemento particular de un arreglo se conoce como el _____ de dicho elemento.

f) El ponerle nombre a un arreglo, indicar su tipo, y especificar el número de elementos dentro del mismo, se conoce como _____ el arreglo.

g) El proceso de colocar los elementos de un arreglo en orden ascendente o descendente, se conoce como _____.

h) En un arreglo de doble subíndice, el primer subíndice (por regla convencional) identifica el _____ del elemento, y el segundo subíndice (por regla convencional), identifica el _____ de un elemento.

i) Un arreglo m por n contiene _____ renglones, _____ columnas y _____ elementos.

j) El nombre del elemento en el renglón 3 y en la columna 5 del arreglo `d` es _____.

6.7 Indique cuál de los siguientes es verdadero y cuál es falso; para aquellos que son falsos, explique por qué lo son.

a) Para referirse a una posición o elemento particular dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento particular.

b) Una declaración de arreglo reserva espacio para el mismo.

c) Para indicar que 100 localizaciones deberán de ser reservadas para el arreglo entero `p`, el programador escribe la declaración

```
p[100];
```

d) Un programa C que inicializa a cero los elemento de un arreglo de 15 elementos debe contener un enunciado `for`.

e) Un programa en C que totaliza los elementos de un arreglo de doble subíndice debe contener enunciados `for` anidados.

f) El promedio, media y modo del siguiente conjunto de valores son 5, 6 y 7 respectivamente: 1, 2, 5, 6, 7, 7, 7.

6.8 Escriba enunciados en C que ejecuten cada uno de los siguientes:

a) Muestre el valor del séptimo elemento del arreglo de caracteres `f`.

b) Introduzca un valor en el elemento 4 de un arreglo de punto flotante de un solo subíndice `b`.

c) Inicialice a `8` cada uno de los 5 elementos de un arreglo entero de un solo subíndice `g`.

d) Totalice los elementos del arreglo de punto flotante `c` de 100 elementos.

e) Copie el arreglo `a` en la primera porción del arreglo `b`. Suponga `float a[11], b[34];`

f) Determine e imprima los valores más pequeños y más grandes contenidos en un arreglo de punto flotante `w` de 99 elementos.

6.9 Suponga un arreglo entero de 2 por 5 de nombre `t`.

a) Escriba una declaración para `t`.

b) ¿Cuántos renglones tiene `t`?

c) ¿Cuántas columnas tiene `t`?

d) ¿Cuántos elementos tiene `t`?

e) Escriba los nombres de todos los elementos del segundo renglón de `t`.

f) Escriba los nombres de todos los elementos de la tercera columna de `t`.

g) Escriba un solo enunciado en C que defina a cero el elemento de `t` en el renglón 1 y columna 2.

h) Escriba una serie de enunciados en C que inicialicen a cero cada elemento de `t`. No utilice una estructura de repetición.

i) Escriba una estructura `for` anidada que inicialice a cero cada elemento de `t`.

j) Escriba un enunciado de C que introduzca los valores para los elementos de `t` desde la terminal.

k) Escriba una serie de enunciados en C que determinen e impriman el valor más pequeño en el arreglo `t`.

l) Escriba un enunciado en C que muestre los elementos del primer renglón de `t`.

m) Escriba un enunciado en C que totalice los elementos de la cuarta columna de `t`.

n) Escriba una serie de enunciados en C que impriman el arreglo `t` en formato tabular nítido. Enliste los subíndices de columna como encabezados en la parte superior, y enliste los subíndices de renglones en la parte izquierda de cada renglón.

6.10 Utilice un arreglo de un subíndice para resolver el siguiente problema. Una empresa le paga a su personal de ventas en base a comisión. Los vendedores reciben $200 por semana más 9 % de sus ventas brutas de dicha semana. Por ejemplo, un vendedor que vende $3000 en ventas brutas en una semana recibe $200 más 9% de 3000, o sea un total de $470. Escriba un programa en C (utilizando un arreglo de contadores) que determine cuántos de los vendedores ganaron salarios en cada uno de los rangos siguientes (suponiendo que el salario de cada vendedor se trunca a una cantidad entera):

1. $200-$299
2. $300-$399
3. $400-$499
4. $500-$599
5. $600-$699
6. $700-$799
7. $800-$899
8. $900-$999
9. $1000 o superior

6.11 La ordenación por el método de burbuja presentada en la figura 6.15, es ineficiente en el caso de arreglos grandes. Haga las siguientes modificaciones simples para mejorar el rendimiento de este tipo de ordenación.

a) Después de la primera pasada el número más alto está garantizado que deberá aparecer en el elemento numerado más alto dentro del arreglo; después de la segunda pasada, los dos números más altos estarán "en su lugar", y así en lo sucesivo. En vez de hacer nueve comparaciones en cada pasada, modifique la ordenación tipo burbuja para llevar a cabo ocho comparaciones en la segunda pasada, 7 en la tercera, y así sucesivamente.

b) Los datos en el arreglo pudieran estar ya en el orden apropiado o en un orden casi apropiado, por lo tanto, ¿por qué hacer nueve pasadas si menos pudieran ser suficientes? Modifique la ordenación para verificar, al final de cada pasada, si se han hecho intercambios. Si no ha habido intercambios, entonces los datos deben de estar ya en el orden apropiado y, por lo tanto, el programa debe darse por terminado. Si ha habido intercambios, entonces por lo menos se requiere de una pasada adicional.

6.12 Escriba enunciados individuales que ejecuten cada una de las operaciones siguientes, de arreglos de un subíndice:

a) Inicialice a ceros los 10 elementos del arreglo entero `counts`.

b) Añada uno a cada uno de los 15 elementos del arreglo entero `bonus`.

c) Lea los 12 valores del arreglo de punto flotante `monthlyTemperatures` desde el teclado.

d) Imprima los 5 valores del arreglo entero `bestScores` en formato de columna.

6.13 Encuentre el error o los errores en cada uno de los enunciados siguientes:

a) Suponga: `char str[5];`
```
scanf("%s", str);      /* User types hello */
```

b) Suponga: `int a [3];`
```
printf("$d %d %d\n", a[1], a[2], a[3]);
```

c)
```
float f[3] = {1.1, 10.01, 100.001, 1000.0001};
```

d) Suponga: `double d[2][10];`
```
d[1, 9] = 2.345;
```

6.14 Modifique el programa de la figura 6.16 para que la función `mode` sea capaz de manejar un empate para el valor modo. También modifique la función `median` de tal forma que en un arreglo que tenga un número par de elementos, los dos elementos del medio sean promediados.

6.15 Utilice un arreglo de un subíndice para resolver el problema siguiente. Lea 20 números, cada uno de los cuales este entre 10 y 100 inclusive. Conforme cada número es leído, imprímalo sólo si no es un duplicado de un número ya leído. Provea para un "caso peor" en el cual los 20 números resulten diferentes. Utilice el arreglo más pequeño posible para resolver este problema.

6.16 Etiquete los elementos de un arreglo de doble subíndice de 3 por 5, de nombre `sales`, para indicar el orden en el cual se definen a cero, mediante el siguiente segmento de programa:

```
for (row = 0; row <= 2; row++)
   for (column = 0; column <= 4; column++)
      sales[row][column] = 0;
```


6.17 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>
#define SIZE 10

int whatIsThis(int [], int);

main()
{
   int total, a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   total = whatIsThis(a, SIZE);
   printf("Total of array element values is %d\n", total);
   return 0;
}

int whatIsThis(int b[], int size)
{
   if (size == 1)
      return b[0];
   else
      return b[size - 1] + whatIsThis(b, size - 1);
}
```

6.18 ¿Qué es lo que lleva a cabo el siguiente programa?

```
#include <stdio.h>
#define SIZE 10

void someFunction(int [], int);

main()
{
   int a[SIZE] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

   printf("The values in the array are:\n");
   someFunction(a, SIZE);
   printf("\n");
   return 0;
}

void someFunction(int b[], int size)
{
   if (size > 0) {
      someFunction(&b[1], size - 1);
      printf("%d ",b[0]);
   }
}
```

6.19 Escriba un programa en C que simule el tirar dos dados. El programa deberá utilizar `rand` para tirar el primer dado, y después volverá a utilizar `rand` para tirar el segundo. La suma de los dos valores deberá entonces ser calculada. *Nota:* en vista de que cada dado puede mostrar un valor entero de 1 a 6, entonces la suma de los dos valores variará desde 2 hasta 12, siendo 7 la suma más frecuente y 2 y 12 las menos frecuentes. En la figura 6.23 se muestran las 36 combinaciones posibles de los dos dados. Su programa deberá tirar 36,000 veces los dos dados. Utilice un arreglo de un subíndice para llevar cuenta del número de veces que aparece cada suma posible. Imprima los resultados en un formato tabular. También, determine si los totales son razonables, es decir, existen seis formas de llegar a un 7, por lo que aproximadamente una sexta parte de todas las tiradas deberán ser 7.

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 6.23 Las 36 posibles combinaciones de tirar dos dados.

6.20 Escriba un programa que ejecute 1000 juegos de craps y que responda a cada una de las preguntas siguientes:

a) ¿Cuántos juegos son ganados en la primera tirada, segunda tirada, ..., tirada número veinte y después de la tirada número veinte?.
b) ¿Cuántos juegos han sido perdidos en la primera tirada, segunda tirada, ..., tirada número veinte y después de la tirada número veinte?.
c) ¿Cuales son las probabilidades de ganar en el craps?. (*Nota:* deberá llegar a la conclusión que el craps es uno de los juegos de casino más justos). ¿Qué es lo que supone que esto significa?
d) ¿Cuál es la longitud o duración promedio de un juego de craps?.
e) ¿Al aumentar la duración del juego aumentan las probabilidades de ganar?

6.21 (*Sistema de reservaciones de aerolínea*). Una pequeña aerolínea acaba de adquirir una computadora para su sistema automatizado de reservaciones. El presidente le ha solicitado a usted que programe el nuevo sistema en C. Usted debe escribir un programa que asigne asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su programa deberá mostrar el siguiente menú de alternativas:

```
Please type 1 for "smoking"
Please type 2 for "nonsmoking"
```

Si la persona escribe 1, entonces su programa deberá asignar un asiento en la sección de fumar (asientos 1 al 5) si la persona escribe 2, entonces su programa deberá de asignar un asiento en la sección de no fumar (asientos 6 al 10). Su programa a continuación deberá imprimir un pase de abordaje, indicando el número de asiento de la persona y si está en la sección o de no fumar del aeroplano.

Utilice un arreglo de un subíndice para representar el diagrama de asientos del avión. Inicialice todos los elementos del arreglo a cero para indicar que todos los asientos están vacíos. Conforme se asigne cada asiento, defina los elementos correspondientes del arreglo a 1 para indicar que dicho asiento ya no está disponible.

Su programa no deberá, naturalmente, asignar nunca un asiento que ya haya sido asignado. Cuando esté llena la sección de fumar, su programa deberá solicitar a la persona, si le parece aceptable ser colocada en la sección de no fumar (o viceversa). Si dice que sí, entonces efectúe la asignación apropiada de asiento. Si dice que no, entonces imprima el mensaje `"Next flight leaves in 3 hours".`

6.22 Utilice una arreglo de doble subíndice para resolver el problema siguiente. Una empresa tiene cuatro vendedores (1 a 4) que venden cinco productos diferentes (1 a 5). Una vez al día, cada vendedor emite un volante para cada tipo distinto de producto vendido. Cada volante contiene:

1. El número del vendedor.
2. El número del producto.
3. El valor total en dólares del producto vendido ese día.

Por lo tanto, cada vendedor entrega por día entre 0 y 5 volantes de ventas. Suponga que está disponible la información de todos los volantes correspondientes al mes anterior. Escriba un programa que lea toda esta información correspondiente a las ventas del mes anterior, y que resuma las ventas totales por vendedor y por producto. Todos los totales deberán almacenarse en un arreglo de doble subíndice `sales`. Después de procesar toda la información correspondiente al mes anterior, imprima los resultados en forma tabular, con cada una de las columnas representando a un vendedor en particular y cada uno de los renglones representando un producto en particular. Totalice en forma cruzada cada renglón, para obtener las ventas totales de cada producto del mes pasado; totalice cada columna para obtener las ventas totales por vendedor correspondiente al mes pasado. Su impresión en forma tabular deberá incluir estos totales a la derecha de los renglones totalizados y en la parte inferior de las columnas totalizadas.

6.23 (*Gráficos tipo tortuga*). El lenguaje Logo, que es en particular popular entre usuarios de computadoras personales, hizo famoso el concepto de los *gráficos tipo tortuga*. Imagine una tortuga mecánica, que camina por la habitación bajo el control de un programa de C. La tortuga sujeta una pluma en dos posiciones posibles, arriba o abajo. Cuando la pluma está abajo, la tortuga traza formas conforme se mueve; cuando la pluma está arriba, la tortuga se mueve a su antojo libremente, sin escribir nada. En este problema simularemos la operación de la tortuga y además crearemos un bloque de notas computarizado.

Utilice un arreglo de 50 por 50 de nombre `floor`, que se inicializa a ceros. Lea los comandos partiendo de un arreglo que los contenga. Lleve control en todo momento de la posición actual de la tortuga, así como de si la pluma en ese momento está arriba o abajo. Suponga que la tortuga siempre empieza a partir de la posición 0,0 en el piso, con su pluma arriba. El conjunto de comandos de la tortuga que su programa debe procesar, son como sigue:

Comando	Significado
1	Pluma arriba
2	Pluma abajo
3	Giro a la derecha
4	Giro a la izquierda
5,10	Moverse hacia adelante 10 espacios (o un número distinto que 10)
6	Imprima el arreglo de 50 por 50
9	Fin de los datos (valor centinela)

Suponga que la tortuga está en algún lugar cerca del centro del piso. El siguiente "programa" dibujaría e imprimiría un cuadrado de 12 por 12:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Conforme la tortuga se mueve con la pluma abajo, defina los elementos apropiados del arreglo `floor` al valor **1**. Cuando se da el comando **6** (imprimir), siempre que exista en el arreglo un **1**, despliegue un asterisco, o cualquier otro carácter que seleccione. Siempre que aparezca un **0**, despliegue un espacio vacío. Escriba un programa en C para poner en operación las capacidades gráficas de la tortuga discutidas aquí. Escriba varios programas gráficos de tortuga para dibujar formas interesantes. Añada otros comandos para incrementar el poder del lenguaje gráfico de su tortuga.

6.24 (*Recorrido del caballo*). Uno de los acertijos más interesantes para los aficionados al ajedrez, es el problema del recorrido del caballo, propuesto originalmente por el matemático Euler. La pregunta es la siguiente: ¿puede la pieza de ajedrez conocida como el caballo moverse en el interior de un tablero de ajedrez vacío y entrar en contacto con cada una de las 64 casillas, una vez y sólo una vez? Estudiamos aquí este problema interesante en profundidad.

El caballo hace movimientos en forma de L (dos en una dirección y a continuación uno en una dirección perpendicular). Entonces, a partir de una casilla en la mitad de un tablero de ajedrez vacío, el caballo puede efectuar ocho movimientos diferentes (numerados desde 0 hasta 7) tal y como se muestra en la figura 6.24.

a) Dibuje un tablero de ajedrez de 8 por 8 en una hoja de papel e intente a mano un recorrido de caballo. Coloque un **1** en la primera casilla a la cual se mueva, un **2** sobre la segunda, un **3** sobre la tercera, etcétera. Antes de iniciar el recorrido, estime qué tan lejos piensa que pueda llegar, recordando que un recorrido completo consistiría de 64 movimientos. ¿Qué tan lejos llegó? ¿Se aproximó en su estimación?

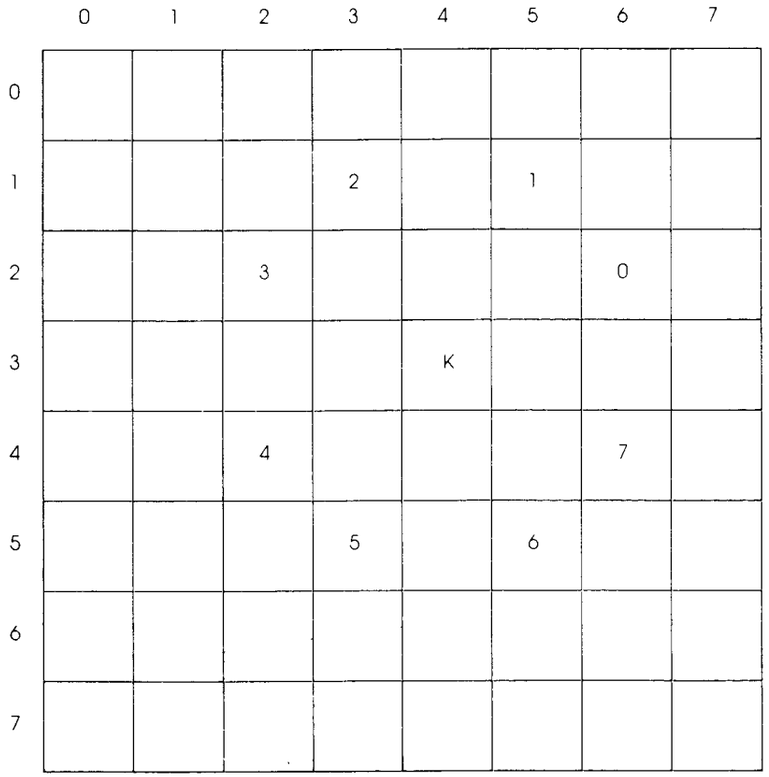

Fig. 6.24 Los ocho movimientos posibles del caballo.

b) Ahora desarrollemos un programa que mueva al caballo sobre el tablero. El tablero mismo se representa por un arreglo de doble subíndice de 8 por 8 de nombre `board`. Cada una de las casillas se inicializa a cero. Describimos cada uno de los ocho movimientos posibles en términos tanto de sus componentes horizontales como verticales. Por ejemplo, un movimiento del tipo 0, tal y como se muestra en la figura 6.24, consiste en mover horizontalmente dos casillas a la derecha y una casilla de forma vertical hacia arriba. El movimiento 2 consiste en mover de manera horizontal una casilla hacia la derecha y dos casillas verticalmente hacia arriba. Los movimientos horizontales a la izquierda y los movimientos verticales hacia abajo se indicarán con números negativos. Los ocho movimientos pueden ser descritos en dos arreglos de un solo subíndice, `horizontal` y `vertical`, como sigue:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

Hagamos las variables `currentRow` y `currentColumn` indicar el renglón y la columna de la posición actual del caballo. Para hacer un movimiento del tipo `moveNumber`, donde `moveNumber` quede entre 0 y 7, su programa utiliza los enunciados

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Lleve un contador que varíe desde **1** hasta **64**. Registre la última cuenta en cada casilla movida por el caballo. Recuerde en probar cada movimiento potencial, para asegurarse que el caballo no haya visitado ya esa casilla. Y, naturalmente, pruebe cada movimiento potencial, para asegurarse que el caballo no se ha salido del tablero de ajedrez. Ahora escriba un programa para mover el caballo por el teclado. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar un programa del recorrido del caballo, probablemente usted ha desarrollado algunos conocimientos valiosos. Los utilizaremos para desarrollar una *heurística* (o estrategia) para el movimiento del caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora en forma importante la oportunidad de éxito. Quizás haya observado que las casillas externas son en cierta forma más difíciles que las casillas más cercanas al centro del tablero. De hecho, las más difíciles o inaccesibles de las casillas son las de las cuatro esquinas.

La intuición le puede sugerir que debería intentar primero mover el caballo a las casillas más problemáticas, y dejar abiertas aquellas que son más fáciles de llegar, a fin de que conforme se vaya congestionando el tablero cerca del fin del recorrido, existan mayores oportunidades de éxito.

Pudiéramos desarrollar una "heurística de accesibilidad" ordenando cada una de las casillas de acuerdo a su accesibilidad, y a partir de eso, moviendo siempre el caballo a la casilla (dentro de los movimientos aceptados del caballo, naturalmente) que sea más inaccesible.

Etiquetamos un arreglo de doble subíndice `accessibility`, con números que indiquen desde cuántas casillas es accesible una casilla en particular. En un tablero en blanco o vacío, las casillas centrales serán, por lo tanto, tasadas o valuadas como `8`, las casillas de esquina como `2`, y las demás casillas tendrán números de accesibilidad de `3`, `4`, o `6` como sigue:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 6 3
2 3 4 4 4 4 3 2
```

Ahora escriba una versión del programa del recorrido del caballo utilizando la heurística de accesibilidad. En todo momento, el caballo deberá moverse a la casilla que tenga el número de accesibilidad más bajo. En caso de empate, el caballo deberá moverse a cualquiera de las casillas de empate. Por lo tanto, el recorrido pudiera empezar en cualquiera de las cuatro esquinas. (*Nota:* conforme el caballo se mueva por el tablero, su programa debería de reducir los números de accesibilidad conforme más y más casillas se van ocupando. De esta forma, en cualquier momento dado durante el recorrido, cada número de accesibilidad de las casillas disponibles se conservará igual precisamente el número de casillas que pueden ser alcanzadas). Ejecute esta versión de su programa. ¿Obtuvo un recorrido completo? Ahora modifique el programa para ejecutar 64 recorridos, uno partiendo de cada una de las casillas del tablero. ¿Cuántos recorridos completos obtuvo?

d) Escriba una versión del programa del recorrido del caballo en la cual, al encontrar un empate entre dos o más casillas, el programa decida cual casilla seleccionar mirando hacia adelante cuáles son las casillas alcanzables a partir de las casillas de "empate". Su programa deberá moverse a la casilla a partir de la cual el siguiente movimiento llegaría a una casilla con el número de accesibilidad menor.

6.25 (*Recorrido del caballo: enfoques de fuerza bruta*). En el ejercicio 6.24 desarrollamos una solución al programa del recorrido del caballo. El enfoque utilizado, conocido como "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

Conforme las computadoras siguen aumentando en potencia, seremos capaces de resolver muchos problemas a base del simple poder de la computadora y utilizando algoritmos relativamente poco complicados. Llamémosle a este enfoque la solución de problemas de "fuerza bruta".

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace sobre el tablero de ajedrez (en movimientos legítimos en forma de L, naturalmente) en forma aleatoria. Su programa debería ejecutar un recorrido e imprimir el tablero de ajedrez final. ¿Qué tan lejos llegó el caballo?

b) Lo más probable es que el programa anterior produjo un recorrido relativamente corto. Ahora modifique su programa para intentar 1000 recorridos. Utilice un arreglo de un solo subíndice para llevar control del número de recorridos de cada longitud. Cuando su programa termine el intento de 1000 recorridos, deberá imprimir esta información en un formato tabular nítido. ¿Cuál fue el resultado mejor?

c) Lo más probable, es que el programa anterior le dió algunos recorridos "respetables" pero ningún recorrido completo. Ahora "quite todas las amarras" y deje sólo que su programa se ejecute hasta que produzca un recorrido completo. (*Precaución.* Esta versión del programa podría ejecutarse durante horas en una computadora poderosa). Aquí otra vez, lleve una tabla del número de recorridos de cada longitud, e imprima esta tabla cuando se encuentre el primer recorrido completo. ¿Cuántos recorridos tuvo que intentar su programa antes de producir un recorrido completo? ¿Cuánto tiempo le tomó?

d) Compare la versión de fuerza bruta del recorrido del caballo con la versión de la heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió de más potencia de la computadora? ¿Podríamos estar seguros (por anticipado) de poder llegar a obtener un recorrido completo con el enfoque heurístico de accesibilidad? ¿Podíamos estar seguros (con anticipación) de obtener un recorrido completo con el enfoque de fuerza bruta? Discuta los pros y los contras de la resolución en general de problemas por fuerza bruta.

6.26 (*Las ocho reinas*). Otro acertijo para aficionados al ajedrez es el problema de las ocho reinas. Dicho simplemente: es o no posible colocar ocho reinas en un tablero de ajedrez vacío, de tal forma que ninguna reina esté atacando a ninguna otra, esto es de tal forma que ¿ningún par de reinas estén en el mismo renglón, la misma columna o a lo largo de la misma diagonal? Utilice el tipo de proceso mental desarrollado en el ejercicio 6.24 para formular una heurística para resolver el problema de las ocho reinas. Ejecute su programa. (*Sugerencia:* es posible asignar un valor numérico a cada casilla del tablero, indicando cuántas casillas son "eliminadas" de un tablero vacío, una vez que se ha colocado una reina en dicha casilla. Por ejemplo, a cada una de las cuatro esquinas deberá asignársele el valor 22, como en la figura 6.25).

Una vez que esos "números de eliminación" son colocados en las 64 casillas, la heurística apropiada podría ser: coloque la reina siguiente en la casilla con el número de eliminación más pequeño. ¿Por qué es de forma intuitiva atrayente esta estrategia?

6.27 (*Ocho reinas: enfoques de fuerza bruta*). En este problema desarrollará varios enfoques de fuerza bruta para resolver el problema de las ocho reinas introducido en el ejercicio 6.26.

a) Resuelva el problema de las ocho reinas, utilizando la técnica de fuerza bruta al azar desarrollada en el problema 6.25.

b) Utilice una técnica exhaustiva, es decir, pruebe todas las combinaciones posibles de las ocho reinas sobre el tablero.

c) ¿Por qué supone que el enfoque exhaustivo de fuerza bruta pudiera no ser apropiado para la resolución del problema del recorrido del caballo?

d) Compare y resalte las diferencias en general de los enfoques de fuerza bruta aleatoria y fuerza bruta exhaustiva.

6.28 (*Eliminación de duplicados*). En el capítulo 12 exploramos la estructura de datos del árbol de búsqueda binaria de alta velocidad. Una característica del árbol de búsqueda binaria, es que los valores duplicados son descartados al hacer inserciones en el árbol. Esto se conoce como eliminación de duplicados. Escriba un programa que produzca 20 números al azar entre 1 y 20. El programa deberá almacenar en un arreglo todos los valores no duplicados. Utilice para esta tarea el arreglo más pequeño posible.

6.29 (*Recorrido del caballo: prueba del recorrido cerrado*). En el recorrido del caballo, un recorrido completo es aquel en que el caballo efectúa 64 movimientos tocando cada casilla del tablero de ajedrez una vez y sólo una vez. Un recorrido cerrado ocurre cuando el 64º movimiento queda a un movimiento de distancia de la posición en la cual el caballo inició el recorrido. Modifique el programa del recorrido del caballo, que escribió en el ejercicio 6.24, para probar para un recorrido cerrado, cuando haya ocurrido un recorrido completo.

Fig. 6.25 Las 22 casillas eliminadas al colocar una reina en la esquina superior izquierda.

6.30 (*La criba de Erastostenes*). Un entero primo es cualquier entero que puede ser sólo dividido entre sí mismo y entre 1. La criba de Erastostenes es un método para encontrar los números primos. Funciona como sigue:

1) Instituya un arreglo con todos los elementos inicializados a 1 (verdadero). Los elementos del arreglo con subíndices primos se conservarán en 1. Todos los otros elementos del arreglo eventualmente se quedarán en cero.
2) Empezando con el subíndice 2 del arreglo (el subíndice 1 debe ser primo), cada vez que se encuentre un elemento de arreglo cuyo valor sea 1, cicle a través del resto del arreglo y defina como cero cualquier elemento cuyo subíndice resulte un múltiplo del subíndice correspondiente al elemento con valor 1. Para el subíndice 2 del arreglo, todos los elementos más allá de 2 dentro del arreglo que sean múltiplos de 2, serán valuados en cero (subíndices 4, 6, 8, 10, etcétera). Para el subíndice del arreglo 3, todos los elementos más allá de 3 en el arreglo que sean múltiples de 3, serán valuados en cero (subíndices 6, 9, 12, 15, etcétera).

Cuando se haya terminado este proceso, los elementos del arreglo que aún estén definidos como 1, indicarán que el subíndice es un número primo. Entonces esos subíndices pueden ser impresos. Escriba un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 1 y 999. Ignore el elemento cero del arreglo.

6.31 (*Ordenación tipo cubeta*). Una ordenación tipo cubeta empieza con un arreglo de un índice de enteros positivos a ordenar, un arreglo de doble subíndice de enteros con renglones con subíndices desde 0 hasta 9 y con columnas con subíndices desde 0 hasta *n* - 1, donde *n* es el número de valores dentro del arreglo a ordenar. Cada renglón del arreglo de doble subíndice se conoce como una cubeta. Escriba una función `bucketSort`, que tome un arreglo de enteros y el tamaño del arreglo como argumentos.

El algoritmo es como sigue:

1) Cicle a través del arreglo de un subíndice y coloque cada uno de sus valores en un renglón del arreglo de cubeta basado en sus dígitos uno. Por ejemplo, 97 se coloca en el renglón 7, 3 se coloca en el renglón 3, y 100 se coloca en el renglón 0.
2) Cicle a través del arreglo de cubeta y copie los valores de regreso al arreglo original. El nuevo orden de los valores anteriores en el arreglo de un solo subíndice es 100, 3 y 97.
3) Repita este proceso para cada posición digital subsecuente (decenas, centenas, miles, etcétera), y deténgase cuando se haya procesado el dígito más a la izquierda del número mayor.

En la segunda pasada del arreglo, 100 se coloca en el renglón 0, 3 se coloca en el renglón 0 (sólo tenía un dígito), y 97 se coloca en el renglón 9. El orden de los valores en el arreglo de un solo subíndice es 100, 3 y 97. En la tercera pasada, 100 se coloca en el renglón 1, 3 se coloca en el renglón 0 y 97 se coloca en el renglón 0 (después de 3). La ordenación por cubeta garantiza tener todos los valores de forma correcta clasificados, una vez procesado el dígito más a la izquierda del número más grande. La ordenación por cubeta sabe que ha terminado, cuando todos los valores se copian en el renglón cero del arreglo de doble subíndice.

Advierta que el arreglo de doble subíndice de cubetas es diez veces del tamaño del arreglo entero ordenándose. Esta técnica de ordenación permite un mayor rendimiento que una ordenación tipo burbuja, pero requiere de una capacidad de almacenamiento mucho mayor. La ordenación tipo burbuja sólo requiere de una localización de memoria adicional para el tipo de datos ordenándose. La ordenación por cubeta es un ejemplo de intercambio, espacio-tiempo. Utiliza más memoria, pero da un mejor rendimiento. Esta versión de la ordenación de cubeta requiere el copiado de todos los datos de regreso al arreglo original en cada una de las pasadas. Otra posibilidad es la creación de un segundo arreglo de cubeta de doble subíndice, y mover repetidamente los datos entre los dos arreglos de cubeta, hasta que todos los datos quedan copiados en el renglón cero de uno de ellos. El renglón cero, contendrá entonces el arreglo ordenado.

Ejercicios de recursión

6.32 (*Ordenación de selección*). Una ordenación de selección recorre un arreglo buscando el elemento más pequeño del mismo. Cuando encuentra el más pequeño, es intercambiado con el primer elemento del

arreglo. El proceso a continuación se repite para el subarreglo que empieza con el segundo elemento del arreglo. Cada pasada del arreglo resulta en un elemento colocado en su posición correcta. Esta ordenación requiere de capacidades de procesamientos similares a la ordenación tipo burbuja para un arreglo de *n* elementos, deberán de hacerse *n* - 1 pasadas, y para cada subarreglo, se harán *n* - 1 comparaciones para encontrar el valor más pequeño. Cuando el subarreglo bajo proceso contenga un solo elemento, el arreglo habrá quedado terminado y ordenado. Escriba una función recursiva `selectionSort` para ejecutar este algoritmo.

6.33 (*Palíndromos*). Un palíndromo es una cadena que se escribe de la misma forma hacia adelante y hacia atrás. Algunos ejemplos de palíndromos son "radar", "able was i ere i saw elba", y "a man a plan a canal panama". Escriba una función recursiva `testPalindrome`, que devuelva uno si la cadena almacenada en el arreglo es un palíndromo y 0 de lo contrario. La función deberá ignorar espacios y puntuaciones incluidas en la cadena.

6.34 (*Búsqueda lineal*). Modifique el programa de la figura 6.18 para utilizar una función recursiva `linearSearch` para ejecutar la búsqueda lineal del arreglo. La función deberá recibir como argumentos un arreglo entero y el tamaño del arreglo. Si se encuentra el valor buscado, devuelva el subíndice del arreglo; de lo contrario devuelva -1.

6.35 (*Búsqueda binaria*). Modifique el programa de la figura 6.19 para utilizar una función recursiva `binarySearch` para ejecutar la búsqueda binaria del arreglo. La función deberá recibir como argumentos un arreglo entero y el subíndice inicial y final. Si el valor buscado es hallado, devuelva el subíndice del arreglo; de lo contrario, devuelva -1.

6.36 (*Ocho reinas*). Modifique el programa ocho reinas creado en el ejercicio 6.26 para resolver el problema en forma recursiva.

6.37 (*Imprima un arreglo*). Escriba una función recursiva `printArray` que toma un arreglo y el tamaño del arreglo como argumentos y que no devuelva nada. La función deberá dejar de procesar y regresar cuando reciba un arreglo de tamaño cero.

6.38 (*Imprimir una cadena de atrás para adelante*). Escriba una función recursiva `stringReverse` que tome un arreglo de caracteres como argumento y que no regrese nada. La función deberá dejar de procesar y regresar cuando se encuentre el carácter nulo de terminación de la cadena.

6.39 (*Encontrar el valor mínimo en un arreglo*). Escriba la función recursiva `recursiveMinimum`, que toma un arreglo entero y el tamaño del arreglo como argumentos y regresa el elemento más pequeño del mismo. La función deberá detener su proceso y regresar cuando reciba un arreglo de un solo elemento.