CAPÍTULO 11
Arreglos

TEMAS

Todas las variables que se han usado hasta ahora han tenido una característica común: cada variable sólo podía utilizarse para almacenar un solo valor a la vez. Por ejemplo, aunque las variables `clave`*, cuenta y calificacion declaradas en las instrucciones*

```
char clave;
int cuenta;
double calificacion;
```

son todas de tipos de datos diferentes, cada variable sólo puede almacenar un valor del tipo de datos declarado. Estos tipos de variables se llaman ***variables atómicas****. Una variable atómica, la cual también se conoce como* ***variable escalar****, es una variable cuyo valor no puede subdividirse o separarse más en un tipo de datos legítimo.*

Con frecuencia se puede tener un conjunto de valores, todos del mismo tipo de datos, que forman un grupo lógico. Por ejemplo, la figura 11.1 ilustra tres grupos de elementos. El primer grupo es una lista de cinco temperaturas en número de precisión doble, el segundo grupo es una lista de cuatro códigos de caracteres y el último grupo es una lista de seis voltajes en número entero.

Temperaturas	Códigos	Voltajes
95.75	Z	98
83.0	C	87
97.625	K	92
72.5	L	79
86.25		85
		72

Figura 11.1 Tres listas de elementos.

Una lista simple que contiene elementos individuales del mismo tipo de datos se llama arreglo unidimensional. En este capítulo se describe cómo se declaran, inicializan, almacenan dentro de una computadora y usan los arreglos unidimensionales. Además, se explorará el uso de arreglos unidimensionales con ejemplos de programas y se presentan los procedimientos para declarar y usar arreglos multidimensionales.

11.1 ARREGLOS UNIDIMENSIONALES

Un **arreglo unidimensional**, el cual también se conoce como **arreglo de dimensión única**, es una lista de valores relacionados con el mismo tipo de datos que se almacena usando un nombre de grupo único.[1] En C++, como en otros lenguajes de computadora, el nombre del grupo se conoce como el **nombre del arreglo**. Por ejemplo, considérese la lista de temperaturas ilustrada en la figura 11.2.

Temperaturas
95.75
83.0
97.625
72.5
86.25

Figura 11.2 Una lista de temperaturas.

[1]Hay que observar que las listas pueden implementarse en una variedad de formas. Un arreglo tan sólo es una implementación de una lista en la que todos los elementos de la lista son del mismo tipo y cada elemento es almacenado de manera consecutiva en un conjunto de ubicaciones de memoria contiguas.

Todas las temperaturas en la lista son números en punto flotante y deben declararse como tales. Sin embargo, los elementos individuales en la lista no tienen que declararse por separado. Los elementos en la lista pueden ser declarados como una sola unidad y almacenarse bajo un nombre de variable común llamado nombre del arreglo. Por comodidad, se elegirá `temp` como el nombre para la lista mostrada en la figura 11.2. Especificar que `temp` va a almacenar cinco valores en punto flotante individuales requiere la instrucción de declaración `double temp[5]`. Hay que observar que esta instrucción de declaración proporciona el nombre del arreglo (o lista), el tipo de datos de los elementos en el arreglo y el número de elementos en el arreglo. Es un ejemplo específico de la instrucción de declaración de arreglo general que tiene la sintaxis:

tipo-de-datos nombreArreglo[número-de-elementos]

La buena práctica de programación requiere definir el número de elementos en el arreglo como una constante antes de declarar el arreglo. Por tanto, la declaración del arreglo previa para `temp` se declararía, en la práctica, usando dos elementos, como:

```
const int NUMELS = 5;  // define una constante para el número de
                          elementos
double temp[NUMELS];   // declara el arreglo
```

Otros ejemplos de declaraciones de arreglo usando esta sintaxis de dos líneas son:

```
const int NUMELS = 6;
int voltios[NUMELS];

const int TAMARREGLO = 4;
char codigo[TAMARREGLO];

const int TAMANHO = 100;
double cantidad[TAMANHO];
```

En estas instrucciones de declaración, a cada arreglo se le asigna suficiente memoria para contener el número de elementos de datos dado en la instrucción de declaración. Por tanto, el arreglo nombrado `voltios` tiene almacenamiento reservado para seis números enteros, el arreglo llamado `codigo` tiene almacenamiento reservado para cuatro caracteres y el arreglo llamado `cantidad` tiene almacenamiento reservado para 100 números de precisión doble. Los identificadores de la constante, `NUMELS, TAMARREGLO` y `TAMANHO` son nombres seleccionados por el programador.

La figura 11.3 ilustra el almacenamiento reservado para los arreglos `voltios` y `codigo`.

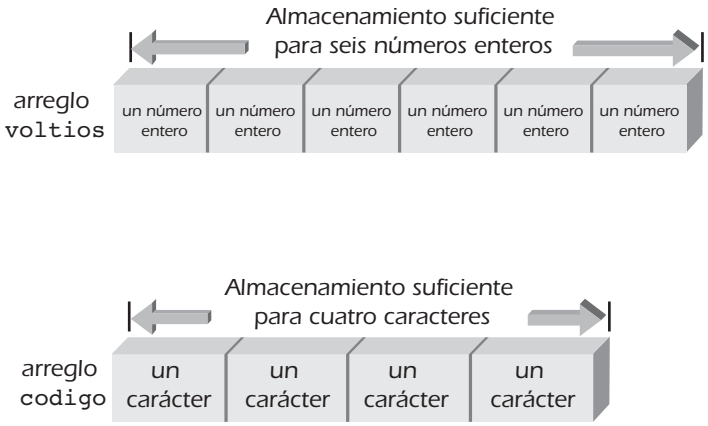


Figura 11.3 *Los arreglos* `voltios` *y* `codigo` *en la memoria.*

Cada elemento en un arreglo se llama **elemento** o **componente** del arreglo. Los elementos individuales almacenados en los arreglos ilustrados en la figura 11.3 se almacenan de manera secuencial, con el primer elemento del arreglo almacenado en la primera ubicación reservada, el segundo elemento almacenado en la segunda ubicación reservada, y así en forma sucesiva hasta que el último elemento es almacenado en la última ubicación reservada. Esta asignación de almacenamiento contiguo para la lista es una característica clave de los arreglos porque proporciona un mecanismo simple para localizar con facilidad cualquier elemento individual en la lista.

Dado que los elementos en el arreglo se almacenan de manera secuencial, puede tenerse acceso a cualquier elemento individual dando el nombre del arreglo y la posición del elemento. Esta posición se llama valor ***índice*** o ***subíndice*** del elemento (los dos términos son sinónimos). Para un arreglo unidimensional, el primer elemento tiene un índice de 0, el segundo elemento tiene un índice de 1, etc. En C++, el nombre del arreglo y el índice del elemento deseado se combinan enlistando el índice entre corchetes después del nombre del arreglo. Por ejemplo, dada la declaración `double temp[5]`.

`temp[0]` se refiere a la primera temperatura almacenada en el arreglo `temp`

`temp[1]` se refiere a la segunda temperatura almacenada en el arreglo `temp`

`temp[2]` se refiere a la tercera temperatura almacenada en el arreglo `temp`

`temp[3]` se refiere a la cuarta temperatura almacenada en el arreglo `temp`

`temp[4]` se refiere a la quinta temperatura almacenada en el arreglo `temp`

La figura 11.4 ilustra el arreglo `temp` en la memoria con la designación correcta para cada elemento del arreglo. Se dice que cada elemento individual es una **variable indexada** o una **variable subindexada**, ya que deben usarse tanto un nombre de variable como un valor índice o subíndice para hacer referencia al elemento. Recuérdese que el valor índice o de subíndice da *la posición* del elemento en el arreglo.

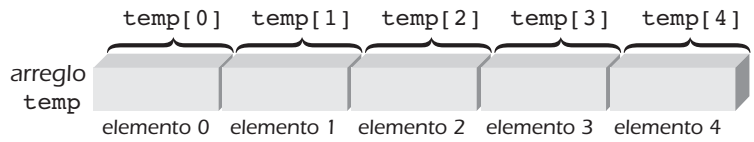


Figura 11.4 *Identificación de elementos individuales del arreglo.*

La variable subindexada, `temp[0]`, se lee como `"temp subíndice cero"`. Ésta es una forma abreviada de decir `"el arreglo temp con subíndice cero"`. Del mismo modo, `temp[1]` se lee como `"temp subíndice uno"`, `temp[2]` como `"temp subíndice dos"`, etcétera.

Aunque puede parecer inusual hacer referencia al primer elemento con un índice de cero, hacerlo así incrementa la velocidad de la computadora cuando tiene acceso a los elementos del arreglo. De manera interna, invisible para el programador, la computadora usa el índice como una compensación de la posición inicial del arreglo. Como se ilustra en la figura 11.5, el índice le indica a la computadora cuántos elementos saltar, empezando desde el principio del arreglo, para obtener el elemento deseado.

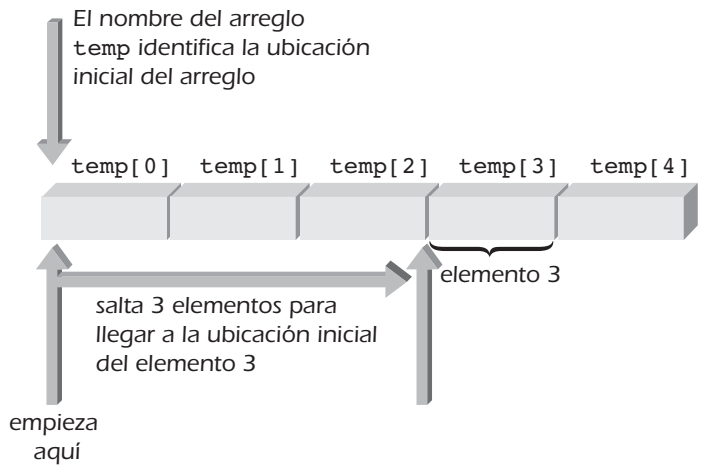


Figura 11.5 *Acceso a un elemento individual del arreglo: el elemento 3.*

Las variables subindexadas pueden usarse en cualquier parte en que sean válidas las variables escalares. Son ejemplos que utilizan los elementos del arreglo temp:

```
temp[0] = 95.75;
temp[1] = temp[0] - 11.0;
temp[2] = 5.0 * temp[0];
temp[3] = 79.0;
temp[4] = (temp[1] + temp[2] - 3.1) / 2.2;
suma = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

El subíndice contenido dentro de los corchetes no necesita ser una constante en número entero; cualquier expresión que evalúe a un número entero puede usarse como un subíndice.[2] En cada caso, por supuesto, el valor de la expresión debe estar dentro del rango de subíndices válidos definidos cuando se declaró el arreglo. Por ejemplo, suponiendo que `i` y `j` son variables `int`, las siguientes variables subindexadas son válidas:

```
temp[i]
temp[2*i]
temp[j-i]
```

Una ventaja importante en extremo de usar expresiones en número entero como subíndices es que permite secuenciar a través de un arreglo usando un ciclo. Esto hace innecesarias instrucciones como

```
suma = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

Los valores de subíndice en esta instrucción pueden reemplazarse por un contador de ciclo `for` para tener acceso a cada elemento en el arreglo en forma secuencial. Por ejemplo, el código

```
suma = 0;                  // inicializa la suma en cero
for (i = 0; i < 5; i++)
  suma = suma + temp[i];   // agrega un valor
```

recupera de manera secuencial cada elemento del arreglo y agrega el elemento a `suma`. Aquí la variable `i` se usa como el contador en el ciclo `for` y como un subíndice. Conforme `i` se incrementa en uno cada vez a través del ciclo, el siguiente elemento en el arreglo es referenciado. El procedimiento para agregar los elementos del arreglo dentro del ciclo `for` es similar al procedimiento de acumulación que se ha usado muchas veces antes.

La ventaja de usar un ciclo `for` para procesar en secuencia a través de un arreglo se hace evidente cuando se trabaja con arreglos grandes. Por ejemplo, si el arreglo `temp` contuviera 100 valores en lugar de sólo cinco, cambiar el número 5 a 100 en la instrucción `for` es suficiente para procesar en secuencia los 100 elementos y agregar cada temperatura a la suma.

Como otro ejemplo del uso de un ciclo `for` para secuenciar un arreglo, suponga que se desea localizar el valor máximo en un arreglo de 1000 elementos llamado `voltios`. El procedimiento que se usará para localizar el valor máximo es suponer inicialmente que el primer elemento en el arreglo es el número más grande. Luego, conforme avanzamos en secuencia a través del arreglo, el máximo se compara con cada elemento. Cuando se localiza un elemento con un valor mayor, ese elemento se convierte en el nuevo máximo. El siguiente código hace el trabajo.

```
const int NUMELS = 1000;

maximo = voltios[0];       // establece el máximo en el elemento cero
for (i = 1; i < NUMELS; i++) // ciclo a traves del resto del arreglo
  if (voltios[i] > maximo)   // compara cada elemento con el máximo
    maximo = voltios[i];     // captura el nuevo valor alto
```

[2]Nota: Algunos compiladores permiten variables de punto flotante como subíndices; en estos casos el valor en punto flotante es truncado a un valor entero.

En este código la instrucción `for` consiste en una instrucción `if`. La búsqueda de un nuevo valor máximo comienza con el elemento 1 del arreglo y continúa hasta el último elemento. Cada elemento es comparado con el máximo actual, y cuando se encuentra un valor más alto éste se convierte en el nuevo máximo.

Entrada y salida de valores de arreglo

A los elementos individuales del arreglo se les pueden asignar valores de manera interactiva usando un objeto de corriente `cin`. Son ejemplos de instrucciones de introducción de datos individuales:

```
cin >> temp[0];
cin >> temp[1] >> temp[2] >> temp[3];
cin >> temp[4] >> voltios[6];
```

En la primera instrucción se leerá y almacenará un solo valor en la variable nombrada `temp[0]`. La segunda instrucción causará que se lean y almacenen tres valores en las variables `temp[1], temp[2]` y `temp[3]`, respectivamente. Por último, puede usarse la última instrucción `cin` para leer valores en las variables `temp[4]` y `voltios[6]`.

De manera alternativa, puede utilizarse un ciclo `for` para recorrer en forma cíclica el arreglo para la introducción interactiva de datos. Por ejemplo, el código

```
const int NUMELS = 5;

for (i = 0; i < NUMELS; i++)
{
  cout << "Introduzca una temperatura: ";
  cin  >> temp[i];
}
```

insta al usuario a introducir cinco temperaturas. La primera temperatura introducida se almacena en `temp[0]`, la segunda temperatura introducida se almacena en `temp[1]`, y así de manera sucesiva hasta que se han introducido cinco temperaturas.

Hay que hacer una advertencia acerca de almacenar datos en un arreglo. C++ no comprueba el valor del índice que se está usando (llamado ***verificación de límites***). Si se ha declarado que un arreglo consta de 10 elementos, por ejemplo, y se usa un índice de 12, el cual está fuera de los límites del arreglo, C++ no notificará el error cuando se compile el programa. El programa intentará tener acceso al elemento 12 saltándose el número apropiado de bytes desde el inicio del arreglo. Por lo general esto produce una caída del programa, pero no siempre. Si la ubicación referenciada en sí contiene un valor del tipo de datos correcto, el nuevo valor tan sólo sobrescribirá el valor en las ubicaciones de memoria referenciadas. Esto conduce a más errores, los cuales son difíciles de localizar, en particular cuando la variable asignada de manera legítima a la ubicación de almacenamiento se usa en un punto diferente del programa.

Durante la salida, pueden desplegarse elementos individuales del arreglo usando el objeto `cout` o secciones completas del arreglo incluyendo una instrucción `cout` dentro de un ciclo `for`. Son ejemplos de esto

```
cout << voltios[6];
```

y

```
cout << "El valor del elemento " << i << " es " << temp[i];
```

Punto de información

Tipos de datos agregados

En contraste con los tipos atómicos, como datos en número entero y de punto flotante, hay tipos agregados. Un tipo agregado, el cual también se conoce como *tipo estructurado* y estructura de datos, es cualquier tipo cuyos valores puedan descomponerse y están relacionados por alguna estructura definida. Además, debe haber operaciones disponibles para recuperar y actualizar valores individuales en la estructura de datos.

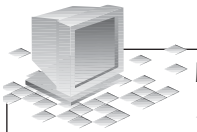
Los arreglos unidimensionales son ejemplos de un tipo estructurado. En un arreglo unidimensional, como un arreglo de números enteros, el arreglo está compuesto por valores enteros individuales, donde los números enteros están relacionados por su posición en la lista. Las variables indexadas proporcionan los medios para tener acceso a los valores en el arreglo y modificarlos.

y

```
const int NUMELS = 20;

for (k = 5; k < NUMELS; k++)
   cout << k << "   " << cantidad[k] << endl;
```

La primera instrucción despliega el valor de la variable subindexada `voltios[6]`. La segunda instrucción despliega el valor del subíndice `i` y el valor de `temp[i]`. Antes que pueda ejecutarse esta instrucción, `i` tendría que tener un valor asignado. Por último, el ejemplo final incluye un objeto `cout` dentro de un ciclo `for`. Se despliegan tanto el valor del índice como el valor de los elementos de 5 a 20.

El programa 11.1 ilustra estas técnicas de entrada y salida usando un arreglo llamado `temp` que se define para almacenar cinco números enteros. Se incluyen en el programa dos ciclos `for`. El primer ciclo `for` se usa para pasar por cada elemento del arreglo y permite al usuario introducir valores individuales del arreglo. Después que se han introducido cinco valores, se usa el segundo ciclo `for` para desplegar los valores almacenados.



Program 11.1

```
#include <iostream>
using namespace std;

int main()
{
  const int TEMPMAX = 5;
  int i, temp[TEMPMAX];

  for (i = 0; i < TEMPMAX; i++)      // Se introducen las temperaturas
  {
    cout << "Introduzca una temperatura: ";
    cin  >> temp[i];
  }

  cout << endl;

  for (i = 0; i << TEMPMAX; i++)      // Se imprimen las temperaturas
    cout << "temperatura " << i << " es " << temp[i] << endl;

  return 0;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.1:

```
Introduzca una temperatura: 85
Introduzca una temperatura: 90
Introduzca una temperatura: 78
Introduzca una temperatura: 75
Introduzca una temperatura: 92

temperatura 0 es 85
temperatura 1 es 90
temperatura 2 es 78
temperatura 3 es 75
temperatura 4 es 92
```

Al revisar la salida producida por el programa 11.1, debe ponerse particular atención a la diferencia entre el valor del índice desplegado y el valor numérico almacenado en el correspondiente elemento del arreglo. El valor del índice se refiere a la ubicación del elemento en el arreglo, mientras la variable subindexada se refiere al valor almacenado en la ubicación designada.

Además de tan sólo desplegar los valores almacenados en cada elemento del arreglo, los elementos también pueden procesarse al referenciar de manera apropiada el elemento deseado. Por ejemplo, en el programa 11.2, el valor de cada elemento se acumula en un total, el cual se despliega al completar el despliegue individual de cada elemento del arreglo.

Programa 11.2

```
#include <iostream>
using namespace std;

int main()
{
  const int TEMPMAX = 5;
  int i, temp[TEMPMAX], total = 0;

  for (i = 0; i < TEMPMAX; i++)      // Se introducen las temperaturas
  {
    cout << "Introduzca una temperatura: ";
    cin  >> temp[i];
  }

  cout << "\nEl total de las temperaturas";

  for (i = 0; i < TEMPMAX; i++)     // Despliega y calcula el total de
                                       las temperaturas
  {
    cout << "   " << temp[i];
    total =  total + temp[i];
  }

  cout << " es " << total << endl;

  return 0;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.2:

```
Introduzca una temperatura: 85
Introduzca una temperatura: 90
Introduzca una temperatura: 78
Introduzca una temperatura: 75
Introduzca una temperatura: 92

El total de las temperaturas 85   90   78   75   92   es   420
```

Hay que observar que en el programa 11.2, a diferencia del programa 11.1, sólo se despliegan los valores almacenados en cada elemento del arreglo y no los números índice. Aunque el segundo ciclo `for` se usó para acumular el total de cada elemento, la acumulación también podría haberse logrado en el primer ciclo colocando la instrucción `total = total + temp[i];` después de la instrucción `cin` usada para introducir un valor. También observe que la instrucción `cout` usada para desplegar el total se hace fuera del segundo ciclo `for`, así que el total se despliega sólo una vez, después que se han agregado todos los valores al total. Si esta instrucción `cout` se colocara dentro del ciclo `for` se des-

plegarían cinco totales, con sólo el último total desplegado conteniendo la suma de todos los valores del arreglo.

Ejercicios 11.1

1. Escriba declaraciones de arreglo para lo siguiente:
   a. una lista de 100 voltajes de precisión doble
   b. una lista de 50 temperaturas de precisión doble
   c. una lista de 30 caracteres, cada uno representando un código
   d. una lista de 100 años en número entero
   e. una lista de 32 velocidades de precisión doble
   f. una lista de 1000 distancias de precisión doble
   g. una lista de 6 números de código enteros
2. Escriba una notación apropiada para el primero, tercero y séptimo elementos de los siguientes arreglos:

   a. `int calificaciones[20]`
   b. `double voltios[10]`
   c. `double amperes[16]`
   d. `int dist[15]`
   e. `double velocidad[25]`
   f. `double tiempo[100]`

3. a. Escriba instrucciones de entrada individuales usando `cin` que puedan usarse para introducir valores en el primero, tercero y séptimo elementos de cada uno de los arreglos declarados en los ejercicios 2a a 2f.
   b. Escriba un ciclo `for` que pueda usarse para introducir valores para el arreglo completo declarado en los ejercicios 2a a 2f.
4. a. Escriba instrucciones de salida individuales usando `cout` que puedan utilizarse para imprimir los valores del primero, tercero y séptimo elementos de cada uno de los arreglos declarados en los ejercicios 2a a 2f.
   b. Escriba un ciclo `for` que pueda usarse para desplegar valores para el arreglo completo declarado en los ejercicios 2a a 2f.
5. Enliste los elementos que serán desplegados por las siguientes secciones de código:

   a.
   ```
   for (m = 1; m <= 5; m++)
      cout << a[m] << " ";
   ```
   b.
   ```
   for (k = 1; k <= 5; k = k + 2)
      cout <<  a[k] << " ";
   ```
   c.
   ```
   for (j = 3; j <= 10; j++)
       cout << b[j] << " ";
   ```
   d.
   ```
   for (k = 3; k <= 12; k = k + 3)
       cout << b[k] << " ";
   ```
   e.
   ```
   for (i = 2; i < 11; i = i + 2)
       cout << c[i] << " ";
   ```

6. a. Escriba un programa para introducir los siguientes valores en un arreglo nombrado `voltios`: 11.95, 16.32, 12.15, 8.22, 15.98, 26.22, 13.54, 6.45, 17.59. Después que se hayan introducido los datos, haga que su programa despliegue los valores.

b. Repita el ejercicio 6a, pero después que se hayan introducido los datos, haga que su programa despliegue en la forma siguiente:

```
11.95    16.32    12.15
 8.22    15.98    26.22
13.54     6.45    17.59
```

7. Escriba un programa para introducir ocho números enteros en un arreglo llamado `temp`. Conforme se introduce cada número, sume los números en un total. Después que se hayan introducido todos los números, despliegue los números y su promedio.

8. a. Escriba un programa para introducir 10 números enteros en un arreglo nombrado `fmax` y determine el valor máximo introducido. Su programa deberá contener sólo un ciclo y el máximo deberá ser determinado conforme se introducen los valores de los elementos del arreglo. (*Sugerencia*: Establezca el máximo igual al primer elemento del arreglo, el cual deberá ser introducido antes que se use el ciclo para introducir los valores restantes del arreglo.)

b. Repita el ejercicio 8a, siguiendo la pista tanto del elemento máximo en el arreglo como el número de índice para el máximo. Después de desplegar los números, imprima estos dos mensajes

```
El valor máximo es:
Este es el elemento numero _____ en la lista de numeros
```

Haga que su programa despliegue los valores correctos en lugar del subrayado en los mensajes.

c. Repita el ejercicio 8b, pero haga que su programa localice el mínimo de los datos introducidos.

9. a. Escriba un programa para introducir los siguientes números enteros en un arreglo llamado `calificaciones`: 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. Conforme introduce cada número, sume los números a un total. Después que todos los números son introducidos y se obtiene el total, calcule el promedio de los números y use el promedio para determinar la desviación de cada valor del promedio. Almacene cada desviación en un arreglo llamado `desviacion`. Cada desviación se obtiene como el valor del elemento menos el promedio de todos los datos. Haga que su programa despliegue cada desviación al lado de su elemento correspondiente del arreglo calificaciones.

b. Calcule la varianza de los datos usados en el ejercicio 9a. La varianza se obtiene elevando al cuadrado cada desviación individual y dividiendo la suma de las desviaciones cuadradas entre el número de `desviaciones`.

10. Escriba un programa que especifique tres arreglos unidimensionales llamados `corriente, resistencia` y `voltios`. Cada arreglo deberá ser capaz de contener 10 elementos. Usando un ciclo `for`, introduzca valores para los arreglos `corriente` y `resistencia`. Las entradas en el arreglo voltios deberán ser el producto de los valores correspondientes en los arreglos corriente y `resistencia` (por tanto, `voltios[i] = corriente[i] * resistencia[i]`). Después que se han introducido todos los datos, despliegue la siguiente salida:

```
Corriente        Resistencia        Voltios
```

Bajo cada encabezado de columna despliegue el valor apropiado.

11.2 INICIALIZACIÓN DE ARREGLOS

Los elementos del arreglo pueden inicializarse dentro de sus instrucciones de declaración de la misma manera como lo hacen las variables escalares, excepto que los elementos inicializados deben incluirse entre llaves. Son ejemplos de dicha inicialización

```
int temp[5] = {98, 87, 92, 79, 85};
char codigos[6] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
double pendientes[7] = {11.96, 6.43, 2.58, .86, 5.89,7.56, 8.22];
```

Los inicializadores se aplican en el orden en que se escriben, con el primer valor usado para inicializar el elemento 0, el segundo valor usado para inicializar el elemento 1, etc., hasta que todos los valores se han usado. Por tanto, en la declaración

```
int temp[5] = {98, 87, 92, 79, 85};
```

`temp[0]` es inicializado en 98, `temp[1]` es inicializado en 87, `temp[2]` es inicializado en 92, `temp[3]` es inicializado en 79 y `temp[4]` es inicializado en 85.

Debido a que el espacio en blanco es ignorado en C++, las inicializaciones pueden continuarse a lo largo de múltiples líneas. Por ejemplo, la declaración

```
int galones[20] = {19, 16, 14, 19, 20, 18, // la inicializacion
                                           de valores
                   12, 10, 22, 15, 18, 17,  // puede extenderse a
                                           lo largo
                   16, 14, 23, 19, 15, 18,  // multiples lineas
                   21, 5};
```

usa cuatro líneas para inicializar todos los elementos del arreglo.

Si el número de inicializadores es menor que el número de elementos declarado enlistados entre corchetes, los inicializadores se aplican empezando con el elemento 0 del arreglo. Por tanto, en la declaración

```
double largo[7] = {7.8, 6.4, 4.9, 11.2};
```

sólo se inicializan `largo[0]`, `largo[1]`, `largo[2]` y `largo[3]` con los valores enlistados. Los otros elementos del arreglo serán inicializados en cero.

Por desgracia, no hay un método para indicar la repetición de un valor de inicialización o para inicializar más tarde elementos del arreglo sin especificar primero valores para elementos anteriores.

Una característica única de los inicializadores es que puede omitirse el tamaño de un arreglo cuando los valores de inicialización están incluidos en la instrucción de declaración. Por ejemplo, la declaración

```
int galones[] = {16, 12, 10, 14, 11];
```

reserva suficiente espacio de almacenamiento para cinco elementos. Del mismo modo, las siguientes dos declaraciones son equivalentes:

```
char codigos[6] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
char codigos[] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
```

Ambas declaraciones apartan siete ubicaciones de caracteres para un arreglo llamado `codigos`. También puede usarse una simplificación interesante y útil cuando se inicializan arreglos de caracteres. Por ejemplo, la declaración

```
char codigos[] = "muestra"; // sin llaves ni comas
```

usa la cadena `"muestra"` para inicializar el arreglo codigos. Recuerde que una cadena es cualquier secuencia de caracteres encerrados entre comillas. Esta última declaración crea un arreglo llamado codigos que tiene ocho elementos y llena el arreglo con los ocho caracteres ilustrados en la figura 11.6. Los primeros siete caracteres, como se esperaba, consisten en las letras `m, u, e, s, t, r` y `a`. El último carácter, el cual es la secuencia de escape \0, se llama **carácter nulo**. El carácter nulo se agrega de manera automática a todas las cadenas que se usan para inicializar un arreglo de carácter y es lo que distingue a una cadena C de una cadena de la clase `string`. Este carácter tiene un código de almacenamiento interno que numéricamente es igual a cero (el código de almacenamiento para el carácter cero tiene un valor numérico de 48 decimal, así que no puede confundirlos la computadora), y se usa como un marcador, o centinela, para marcar el final de una cadena.

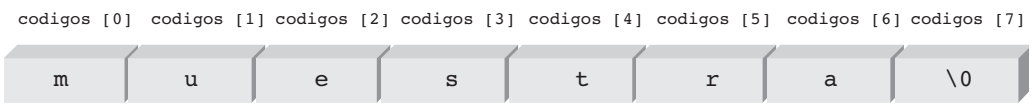
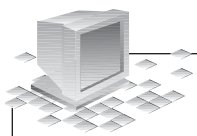


Figura 11.6 *Inicializar un arreglo de carácter con una cadena agrega un carácter \0 de terminación..*

Una vez que se han asignado los valores a los elementos del arreglo, ya sea a través de la inicialización dentro de la instrucción de declaración o usando entrada interactiva, los elementos del arreglo pueden ser procesados como se describió en la sección anterior. Por ejemplo, el programa 11.3 ilustra la inicialización de elementos del arreglo dentro de la declaración del arreglo y luego usa un ciclo `for` para localizar el valor máximo almacenado en el arreglo.



Programa 11.3

```
#include <iostream>
using namespace std;

int main()
{
  const int ELMAX = 5;

  int i, max, nums[ELMAX] = {2, 18, 1, 27, 16};

  max = nums[0];

  for (i = 1; i < ELMAX; i++)
    if (max < nums[i])
      max = nums[i];

  cout << "El valor máximo es " << max << endl;

  return 0;
}
```

La salida producida por el programa 11.3 es

```
El valor maximo es 27
```

Ejercicios 11.2

1. Escriba declaraciones de arreglos, incluyendo inicializadores, para lo siguiente
   - **a.** una lista de 10 voltajes en números enteros: 89, 75, 82, 93, 78, 95, 81, 88, 77, 82
   - **b.** una lista de cinco pendientes en número de precisión doble: 11.62, 13.98, 18.45, 12.68, 14.76
   - **c.** una lista de 100 distancias en número de precisión doble; las primeras seis distancias son 6.29, 6.95, 7.25, 7.35, 7.40, 7.42
   - **d.** una lista de 64 temperaturas en número de precisión doble; las primeras 10 temperaturas son 78.2, 69.6, 68.5, 83.9, 55.4, 67.0, 49.8, 58.3, 62.5, 71.6
   - **e.** una lista de 15 códigos de carácter; los primeros siete códigos son f, j, m, q, t, w, z
2. Escriba una instrucción de declaración de arreglo que almacene los siguientes valores en un arreglo llamado `voltios`: 16.24, 18.98, 23.75, 16.29, 19.54, 14.22, 11.13, 15.39. Incluya estas instrucciones en un programa que despliegue los valores en el arreglo.
3. Escriba un programa que use una instrucción de declaración de arreglo para inicializar los siguientes números en un arreglo llamado `pendientes:` 17.24, 25.63, 5.94, 33.92, 3.71, 32.84, 35.93, 18.24, 6.92. Su programa deberá localizar y desplegar los valores máximo y mínimo en el arreglo.
4. Escriba un programa que almacene los siguientes valores en un arreglo llamado `resistencia`: 16, 27, 39, 56 y 81. Su programa también deberá crear dos arreglos llamados `corriente` y `potencia`, cada uno capaz de almacenar cinco números de precisión doble. Usando un ciclo `for` y una instrucción `cin`, haga que su programa acepte cinco números introducidos por el usuario en el arreglo `corriente` cuando se esté ejecutando el programa. Su programa deberá almacenar el producto de los valores correspondientes del cuadrado del arreglo `corriente` y el arreglo `resistencia` en el arreglo `potencia` (por ejemplo, `potencia[1] = resistencia[1] * pow(corriente[1],2)` y despliegue la siguiente salida (llene la tabla de manera apropiada):

```
Resistencia      Corriente      Potencia
16                  .              .
27                  .              .
39                  .              .
56                  .              .
81                  .              .
                                  --------
Total:                            .
```

5. **a.** Escriba una declaración para almacenar la cadena `"Esta es una prueba"` en un arreglo llamado `pruebacadena`. Incluya la declaración en un programa para desplegar el mensaje usando el siguiente ciclo:

```
for (i = 0; i < DESPLIEGUENUM; i++)
  cout << pruebacadena[i];
```

donde `DESPLIEGUENUM` es una constante nombrada para el número 14.

b. Modifique la instrucción `for` en el ejercicio 5a para desplegar sólo los caracteres p, r, u, e, b y a del arreglo.
c. Incluya la declaración del arreglo escrita en el ejercicio 5a en un programa que use el objeto `cout` para desplegar caracteres en el arreglo. Por ejemplo, la instrucción `cout << pruebacadena;` causará que se despliegue la cadena almacenada en el `arreglo pruebacadena`. Utilizar esta instrucción requiere que el último carácter en el arreglo sea el marcador de fin de cadena `\0`.
d. Repita el ejercicio 5a usando un ciclo `while`. (*Sugerencia*: Detenga el ciclo cuando se detecte la secuencia de escape \0. Puede usarse la expresión `while (pruebacadena[i] != '\0').`)

11.3 DECLARACIÓN Y PROCESAMIENTO DE ARREGLOS BIDIMENSIONALES

Un ***arreglo bidimensional***, el cual a veces se denomina tabla, consiste de filas y columnas de elementos. Por ejemplo, el arreglo de números

8	16	9	52
3	15	27	6
14	25	2	10

se llama arreglo bidimensional de números enteros. Este arreglo consiste de tres filas y cuatro columnas. Para reservar almacenamiento para este arreglo, deben incluirse el número de filas y el número de columnas en la declaración del arreglo. Llamando al arreglo `val`, la especificación correcta para este arreglo bidimensional es

```
int val[3][4];
```

Del mismo modo, las declaraciones

```
double voltios[10][5];
char codigo[6][26];
```

declaran que el arreglo `voltios` consta de 10 filas y 5 columnas de números de precisión doble y que el arreglo `codigo` consta de 6 filas y 26 columnas, con cada elemento capaz de contener un carácter.

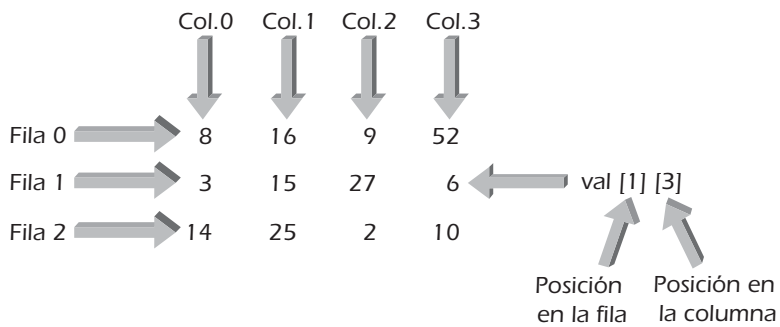


Figura 11.7 Cada elemento del arreglo es identificado por su posición en las filas y columnas.

Para localizar cada elemento en un arreglo bidimensional, un elemento se identifica por su posición en el arreglo. Como se ilustra en la figura 11.7, el término `val[1][3]` identifica en forma única al elemento en la fila 1, columna 3. Como con las variables de arreglo unidimensional, las variables de arreglo bidimensional pueden usarse en cualquier parte en que sean válidas las variables escalares. Son ejemplos del uso de elementos del arreglo `val`

```
watts = val[2][3];
val[0][0] = 62;
nuevonum = 4 * (val[1][0] - 5);
sumaFila0 = val[0][0] + val[0][1] + val[0][2] + val[0][3];
```

La última instrucción causa que los valores de los cuatro elementos en la fila 0 se sumen y la suma se almacene en la variable escalar `sumaFila0.`

Como con los arreglos unidimensionales, los arreglos bidimensionales pueden inicializarse desde dentro de sus instrucciones de declaración. Esto se hace enlistando los valores iniciales dentro de llaves y separándolos con comas. Además, las llaves pueden usarse para separar filas individuales. Por ejemplo, la declaración

```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                  {14,25,2,10} };
```

declara que `val` es un arreglo de números enteros con tres filas y cuatro columnas, con sus valores iniciales dados en la declaración. El primer conjunto de llaves internas contiene los valores para la fila 0 del arreglo, el segundo conjunto de llaves internas contiene los valores para la fila 1, y el tercer conjunto de llaves, los valores para la fila 2.

Aunque siempre se requieren las comas en las llaves de inicialización, las llaves internas pueden omitirse. Por tanto, la inicialización para `val` puede escribirse como

```
int val[3][4] = {8,16,9,52,
                 3,15,27,6,
                14,25,2,10};
```

La separación de los valores iniciales en filas en la instrucción de declaración no es necesaria en virtud que el compilador asigna valores comenzando con el elemento `[0][0]` y procede fila por fila para llenar los valores restantes. Por tanto, la inicialización

```
int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

es igual de válida pero no ilustra con claridad a otro programador dónde termina una fila y comienza otra.

Como se ilustra en la figura 11.8, la inicialización de un arreglo bidimensional se hace en el orden de las filas. Primero se inicializan los elementos de la primera fila, luego se inicializan los elementos de la segunda fila, y así en forma sucesiva, hasta que se completan las inicializaciones. Este ordenamiento por filas también es el mismo ordenamiento usado para almacenar arreglos bidimensionales. Es decir, el elemento `[0][0]` del arreglo es almacenado primero, seguido por el elemento `[0][1]`, luego el elemento `[0][2]`, etc. Después de los elementos de la primera fila están los elementos de la segunda fila, y así en forma sucesiva para todas las filas en el arreglo.

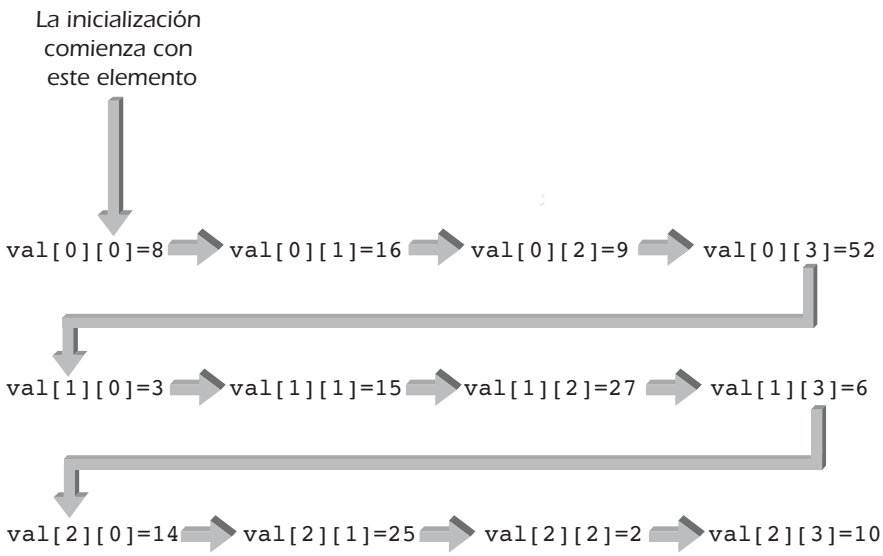


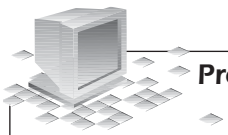
Figura 11.8 *Almacenamiento e inicialización del arreglo* `val`.

Como con los arreglos unidimensionales, los arreglos bidimensionales pueden desplegarse por notación de elemento individual o usando ciclos (ya sea `while` o `for`). Esto se ilustra con el programa 11.4, el cual despliega todos los elementos de un arreglo bidimensional de tres por cuatro usando dos técnicas diferentes. Hay que observar en el programa 11.4 que se han usado constantes para definir las filas y columnas del arreglo.

A continuación se muestra el despliegue producido por el programa 11.4.

```
Despliegue del arreglo val por elemento explícito
   8  16   9  52
   3  15  27   6
  14  25   2  10

Despliegue del arreglo val usando un ciclo for anidado
   8  16   9  52
   3  15  27   6
  14  25   2  10
```

El primer despliegue del arreglo `val` producido por el programa 11.4 se construye al designar en forma explícita cada elemento del arreglo. El segundo despliegue de valores de elementos del arreglo, el cual es idéntico al primero, se produce usando un ciclo `for` anidado. Los ciclos anidados son útiles en especial cuando se trata con arreglos bidimensionales debido a que le permiten al programador designar y llegar con facilidad a cada elemento. En el programa 11.4, la variable `i` controla el ciclo exterior y la variable `j` controla el ciclo interior. Cada pasada por el ciclo exterior corresponde a una sola fila, con el ciclo interior suministrando los elementos de columna apropiados. Después que se imprime una fila completa, se empieza una línea nueva para la siguiente fila. El efecto es un despliegue del arreglo en una forma fila por fila.



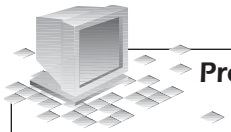
Programa 11.4

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{

  const int NUMFILAS = 3;
  const int NUMCOLS = 4;

  int i, j;
  int val[NUMFILAS][NUMCOLS] = {8, 16, 9, 52, 3, 15, 27, 6, 14, 25, 2, 10};

  cout << "\nDespliegue del arreglo val por elemento explícito"
       << endl << setw(4) << val[0][0] << setw(4) << val[0][1]
       << setw(4) << val[0][2] << setw(4) << val[0][3]
       << endl << setw(4) << val[1][0] << setw(4) << val[1][1]
       << setw(4) << val[1][2] << setw(4) << val[1][3]
       << endl << setw(4) << val[2][0] << setw(4) << val[2][1]
       << setw(4) << val[2][2] << setw(4) << val[2][3];

  cout << "\n\nDespliegue del arreglo val usando un ciclo for anidado";

  for (i = 0; i < NUMFILAS; i++)
  {
    cout << endl;L       // imprime una linea nueva para cada fila
    for (j = 0; j < NUMCOLS; j++)
      cout << setw(4) <<  val[i][j];
  }

  cout << endl;

  return 0;
}
```

Una vez que se han asignado los elementos del arreglo bidimensional, puede comenzar el procesamiento del arreglo. Por lo general, los ciclos `for` se usan para procesar arreglos bidimensionales porque, como se señaló antes, permiten al programador designar y llegar con facilidad a cada elemento del arreglo. Por ejemplo, el ciclo `for` anidado ilustrado en el programa 11.5 se usa para multiplicar cada elemento en el arreglo `val` por el número escalar 10 y desplegar el valor resultante.

A continuación se muestra la salida producida por el programa 11.5:

```
Despliegue de elementos multiplicados
   80  160   90  520
   30  150  270   60
  140  250   20  100
```



Programa 11.5

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUMFILAS = 3;
  const int NUMCOLS = 4;

  int i, j;
  int val[NUMFILAS][NUMCOLS] = {8, 16, 9, 52,
                                3,15,27,6,
                               14,25,2,10};

// multiplica cada elemento por 10 y lo despliega
  cout << "\nDespliegue de elementos multiplicados";
  for (i = 0; i < NUMFILAS; i++)
  {
    cout << endl; // empieza cada fila en una linea nueva
    for (j = 0; j < NUMCOLS; j++)
    {
      val[i][j] = val[i][j] * 10;
      cout << setw(5) << val[i][j];
    }  // fin del ciclo interior
  }    // fin del ciclo exterior
  cout << endl;

  return 0;
}
```

Arreglos dimensionales mayores

Aunque los arreglos con más de dos dimensiones no se usan por lo común, C++ permite que se declare cualquier número de dimensiones. Esto se hace enlistando el tamaño máximo de todas las dimensiones para el arreglo. Por ejemplo, la declaración `int respuesta[4][10][6];` declara un arreglo tridimensional. El primer elemento en el arreglo es designado como respuesta `[0][0][0]` y el último elemento como `respuesta[3][9][5]`.

Desde el punto de vista conceptual, como se ilustra en la figura 11.9, un arreglo tridimensional puede verse como un libro de tablas de datos. Usando esta visualización, el primer índice puede considerarse como la ubicación de la fila deseada en una tabla, el segundo valor índice como la columna deseada y el tercer valor índice, el cual con frecuencia se llama "rango", como el número de página de la tabla seleccionada.

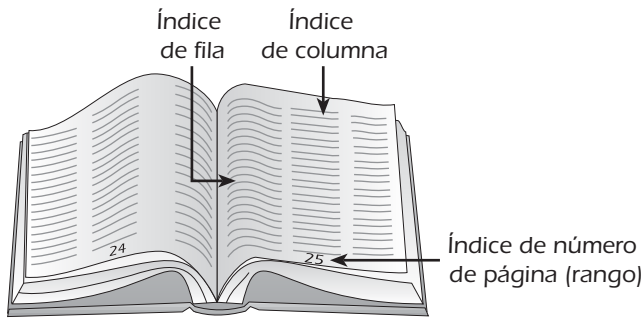


Figura 11.9 *Representación de un arreglo tridimensional.*

Del mismo modo, pueden declararse arreglos de cualquier dimensión. Desde el punto de vista conceptual, un arreglo tetradimensional puede representarse como un anaquel de libros, donde la cuarta dimensión se usa para declarar un libro deseado en el anaquel, y un arreglo pentadimensional puede verse como un librero lleno de libros en el que la quinta dimensión se refiere al anaquel seleccionado en el librero. Usando la misma analogía, un arreglo de seis dimensiones puede considerarse como una hilera de libreros donde la sexta dimensión hace referencia al librero deseado en la hilera; un arreglo de siete dimensiones puede considerarse como múltiples hileras de libreros donde la séptima dimensión hace referencia a la hilera deseada, etc. De manera alternativa, los arreglos de tres, cuatro, cinco, seis, etc. dimensiones pueden verse como *n* múltiplos matemáticos del orden de tres, cuatro, cinco, seis, etc., respectivamente.

Ejercicios 11.3

1. Escriba instrucciones de especificación apropiadas para
   - **a.** un arreglo de números enteros con 6 filas y 10 columnas
   - **b.** un arreglo de números enteros con 2 filas y 5 columnas
   - **c.** un arreglo de caracteres con 7 filas y 12 columnas
   - **d.** un arreglo de caracteres con 15 filas y 7 columnas
   - **e.** un arreglo de números de precisión doble con 10 filas y 25 columnas
   - **f.** un arreglo de números de precisión doble con 16 filas y 8 columnas

2. Determine la salida producida por el siguiente programa:

```
#include <iostream>
using namespace std;

int main()
{
  int i, j, val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

```
  for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j)
      cout << "  " << val[i][j];

  return 0;
}
```

3. a. Escriba un programa en C++ que sume los valores de todos los elementos en el arreglo val usados en el ejercicio 2 y despliegue el total.
b. Modifique el programa escrito para el ejercicio 3a para desplegar el total de cada fila por separado.

4. Escriba un programa en C++ que sume elementos equivalentes de los arreglos bidimensionales llamados `primero` y `segundo`. Ambos arreglos deberán tener dos filas y tres columnas. Por ejemplo, el elemento `[1][2]` del arreglo resultante deberá ser la suma de `primero[1][2]` y `segundo[1][2]`. Los arreglos `primero` y `segundo` deberán inicializarse como sigue:

```
     PRIMERO              SEGUNDO
16    18    23        24    52    77
54    91    11        16    19    59
```

5. a. Escriba un programa en C++ que encuentre y despliegue el valor máximo en un arreglo bidimensional de números enteros. El arreglo deberá ser declarado como un arreglo de números enteros de 4 por 5 e inicializarse con los datos 16, 22, 99, 4, 18, −258, 4, 101, 5, 98, 105, 6, 15, 2, 45, 33, 88, 72, 16, 3
b. Modifique el programa escrito en el ejercicio 5a de modo que también despliegue los números de los subíndices de fila y columna del valor máximo.

6. Escriba un programa en C++ para seleccionar los valores en un arreglo de cuatro por cinco de números enteros positivos en orden creciente y almacenar los valores seleccionados en el arreglo unidimensional llamado **ordenar**. Use la instrucción de datos dada en el ejercicio 5a para inicializar el arreglo bidimensional.

7. a. Un ingeniero ha construido un arreglo bidimensional de números reales que tiene 3 filas y 5 columnas. Este arreglo contiene en la actualidad los voltajes de prueba de un amplificador. Escriba un programa en C++ que introduzca de manera interactiva 15 valores del arreglo y luego determine el número total de voltajes en los rangos menor que 60, mayor que o igual a 60 y menor que 70, mayor que o igual a 70 y menor que 80, mayor que o igual a 80 y menor que 90, y mayor que o igual a 90.
b. Introducir 15 voltajes cada vez que se ejecuta el programa escrito para el ejercicio 7a es engorroso. Por consiguiente, ¿qué método es apropiado para inicializar el arreglo durante la fase de prueba?
c. ¿Cómo podría modificarse el programa que escribió para el ejercicio 7a para incluir el caso de que no haya ningún voltaje presente? Es decir, ¿qué voltaje podría usarse para indicar un voltaje inválido y cómo tendría que modificarse su programa para excluir el conteo de dicho voltaje?

11.4 APLICACIONES

Los arreglos son muy útiles en cualquier aplicación que requiera múltiples recorridos por el mismo conjunto de elementos de datos. Dos de dichas aplicaciones se presentan en esta sección. La primera aplicación es un análisis estadístico de datos que requiere dos recorridos por los datos. El primer recorrido se usa para introducir y determinar el promedio de los datos, mientras el segundo recorrido usa el promedio para determinar una desviación estándar. La segunda aplicación presenta un método simple pero elegante de graficar datos en una pantalla de video o en una impresora estándar. Aquí se usa un primer recorrido para inicializar el arreglo con los puntos de datos que se van a graficar y para determinar los valores mínimos y máximo del arreglo. Estos valores se usan luego para calcular un factor de escala apropiado para asegurar que la gráfica final cabe dentro del área de la pantalla de video o el papel. Por último, se hace un segundo recorrido por el arreglo para producir la gráfica.

Aplicación 1: Análisis estadístico

Se va a desarrollar un programa que acepte una lista de un máximo de 100 voltajes como entrada, determine tanto el promedio como la desviación estándar de los voltajes introducidos, y luego despliegue los resultados.

Paso 1 Analizar el problema

El planteamiento del problema indica que se requieren dos valores de salida: un promedio y una desviación estándar. En el paso 2 se verificará que se sabe cómo se calculan estos valores. El elemento de entrada definido en el planteamiento del problema es un máximo de 100 voltajes. Esto significa que cualquier número de voltajes, de cero a 100 podría ser introducido por el usuario cuando se ejecute el programa. Para facilitar esto, se tendrá que preguntar al usuario cuántos voltajes pretende introducir. Por tanto, la primera entrada será el número de voltajes que se introducirán.

Paso 2 Desarrollar una solución

Las especificaciones de E/S determinadas en el paso 1 definen que el usuario introducirá dos tipos de entradas: el número de voltajes seguido por los datos reales. Con base en esta entrada el programa calculará y desplegará el promedio y la desviación estándar de los datos. Estos elementos de salida se determinan como sigue:

Calcular el promedio sumando los voltajes y dividiéndolos entre el número de voltajes que se sumaron
Determinar la desviación estándar

1. ***Restando el promedio de cada voltaje individual: esto produce un conjunto de miembros nuevos, cada uno de los cuales se llama desviación***
2. ***Elevar al cuadrado cada desviación encontrada en el paso anterior***
3. ***Sumar las desviaciones cuadradas y dividir la suma entre el número de desviaciones***
4. ***La raíz cuadrada del número encontrado en el paso anterior es la desviación estándar***

Hay que observar que el cálculo de la desviación estándar requiere el promedio, lo cual significa que la desviación estándar sólo puede ser calculada después que se ha calculado el promedio. Ésta es la ventaja de especificar el algoritmo, en detalle, antes de realizar cualquier codificación; asegura que todas las entradas y todos los requisitos necesarios se descubren pronto en el proceso de programación.

Para asegurar que se entiende el procesamiento requerido, se hace un cálculo manual. Para este cálculo se supondrá de manera arbitraria que se van a determinar el promedio y la desviación estándar de los siguientes 10 voltajes: 98, 82, 67, 54, 78, 83, 95, 76, 68 y 63.

El promedio de estos datos se determina como

$$\begin{aligned}\text{Promedio} &= (98 + 82 + 67 + 54 + 78 + 83 + 95 + 76 + 68 + 63)/10\\ &= 76.4\end{aligned}$$

La desviación estándar se calcula determinando primero la suma de las desviaciones cuadradas. Entonces se obtiene la desviación estándar dividiendo la suma resultante entre 10 y extrayendo su raíz cuadrada.

$$\begin{aligned}\text{Suma de desviaciones cuadradas} &= (98 - 76.4)^2\\ &\quad + (82 - 76.4)^2\\ &\quad + (67 - 76.4)^2\\ &\quad + (54 - 76.4)^2\\ &\quad + (78 - 76.4)^2\\ &\quad + (83 - 76.4)^2\\ &\quad + (95 - 76.4)^2\\ &\quad + (76 - 76.4)^2\\ &\quad + (68 - 76.4)^2\\ &\quad + (63 - 76.4)^2\\ &= 1730.4007\end{aligned}$$

$$\begin{aligned}\text{Desviación estándar} &= \sqrt{1730.4007 / 10}\\ &= \sqrt{173.04007}\\ &= 13\,154470\end{aligned}$$

Habiendo especificado el algoritmo requerido de cada función, ahora se está en posición de codificarlos.

Paso 3 Codificar la solución

El programa 11.6 presenta la versión en C++ del algoritmo seleccionado. Hay que observar que el programa usa un ciclo `for` para introducir y sumar los voltajes individuales y un segundo ciclo `for` para determinar la suma de las desviaciones cuadradas. Debido a que el cálculo de las desviaciones cuadradas requiere el promedio, la desviación estándar sólo puede calcularse después que se ha calculado el promedio. Obsérvese también que el valor de terminación del contador del ciclo en ambos ciclos `for` es `numvoltios`, el cual es el número de voltajes introducidos por el usuario. El uso de este argumento le da al programa su generalidad y permite que sea usado en listas de cualquier número de voltajes, hasta `NUMELS`, el cual es definido como 100 por la instrucción `const`.

Programa 11.6

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{

  const int NUMELS = 100;

  int i, numvoltios;
  double voltio[NUMELS];
  double promedio, desvest;
  double sumavoltios = 0.0, sumdesv = 0.0;

  cout << "Introduzca el número de voltajes que se va a analizar: ";
  cin >> numvoltios;

  // lee los voltajes de entrada y los totaliza
  for (i = 0; i < numvoltios; i++)
  {
      cout << "Introduzca voltaje " << i+1 << ": ";
      cin >> voltio[i];
    sumvoltios = sumvoltios + voltio[i];
  }
  // calcula y despliega el promedio
  promedio = sumvoltios / numvoltios;
  cout << "\nEl promedio de los voltajes es "
       << setw(11) << setiosflags(ios::showpoint)
       << setprecision(8) << promedio << endl;

  // calcula y despliega la desviación estándar
  for (i = 0; i < numvoltios; i++)
     sumadesv = sumadesv + pow(voltio[i] - promedio),2);
  desvest = sqrt(sumadesv/numvoltios);
  cout << "La desviación estándar de los voltajes es "
       << setw(11) << setiosflags(ios::showpoint)
       << setprecision(8) << desvest << endl;

  return 0;

}
```

Paso 4 Prueba y corrección del programa

Una ejecución de prueba usando el programa 11.6 produjo el siguiente despliegue:

```
Introduzca el número de voltajes que se va a analizar: 10
Introduzca voltaje 1: 98
Introduzca voltaje 2: 82
Introduzca voltaje 3: 67
Introduzca voltaje 4: 54
Introduzca voltaje 5: 78
Introduzca voltaje 6: 83
Introduzca voltaje 7: 95
Introduzca voltaje 8: 76
Introduzca voltaje 9: 68
Introduzca voltaje 10: 63

El promedio de los voltajes es 76.400000
La desviación estandar de los voltajes es 13.154467
```

Aunque este resultado concuerda con nuestro cálculo manual previo, la prueba en realidad no está completa sin verificar el cálculo en los puntos límite. En este caso dicha prueba consiste en comprobar el cálculo con todos los valores iguales, como todos 0 y todos 100. Otra prueba simple sería usar cinco 0 y cinco 100. Se dejan estas pruebas como ejercicio.

Aplicación 2: Mantenimiento de una lista[3]

Un problema de programación común es mantener una lista en orden numérico o alfabético. Por ejemplo, los números de partes de inventario se mantienen por lo general en orden numérico, pero las listas telefónicas se conservan en orden alfabético.

Para esta aplicación, escriba una función que inserte un código de número de parte de tres dígitos dentro de una lista de números de parte. La lista se mantiene en orden numérico creciente y no se permiten códigos de número de parte duplicados. Se tiene que asignar un tamaño máximo de la lista de 100 valores y se usará un valor centinela de 9999 para indicar el final de la lista. Por ejemplo, si la lista actual contiene nueve códigos de número de parte, la décima posición en la lista contendrá el valor centinela.

Paso 1 Analizar el problema

La salida requerida es una lista actualizada de códigos de tres dígitos en la cual el código nuevo ha sido insertado en la lista existente. Los elementos introducidos para esta función son el arreglo existente de códigos de identificación y el nuevo código que se ha insertado en la lista.

[3]Este tema requiere una comprensión de la transmisión de arreglos a una función (véase la sección 11.5).

Paso 2 Desarrollar una solución

La inserción de un número de parte nuevo en la lista existente requiere el siguiente procedimiento:

Determinar en qué parte de la lista se colocará el código nuevo
Esto se hace comparando el código nuevo con cada valor en la lista actual hasta que se encuentre una coincidencia, se localiza un código de identificación más grande que el código nuevo o se encuentra el final de la lista
Si el código nuevo coincide con un código existente,
despliega un mensaje de que el código existe
Si no
Para hacer espacio para el nuevo elemento en el arreglo,
se mueve cada elemento una posición hacia abajo. Esto se hace comenzando desde el valor centinela y moviendo cada elemento hacia abajo una posición hasta que queda desocupada la posición deseada en la lista
Se inserta el código nuevo en la posición desocupada
Termina el si

Para asegurar que se entiende este algoritmo, se hará un cálculo manual. Para este cálculo, suponga que la lista de códigos de identificación consiste de los números mostrados en la figura 11.10a. Si el código número 142 se va a insertar en esta lista, debe colocarse en la cuarta posición en la lista después del número 136. Para hacer espacio para el código nuevo, todos los códigos de la cuarta posición hasta el final de la lista deben moverse una posición hacia abajo como se ilustra en la figura 11.10b. El movimiento siempre se inicia desde el final de la lista y procede desde el valor centinela hacia atrás hasta alcanzar la posición deseada en la lista. (Si la copia procediera hacia delante desde el cuarto elemento, el número 144 sería reproducido en todas las ubicaciones subsiguientes hasta alcanzar el valor centinela.) Después del movimiento de los elementos necesarios, se inserta el nuevo código en la posición correcta. Esto crea la lista actualizada que se muestra en la figura 11.10c.

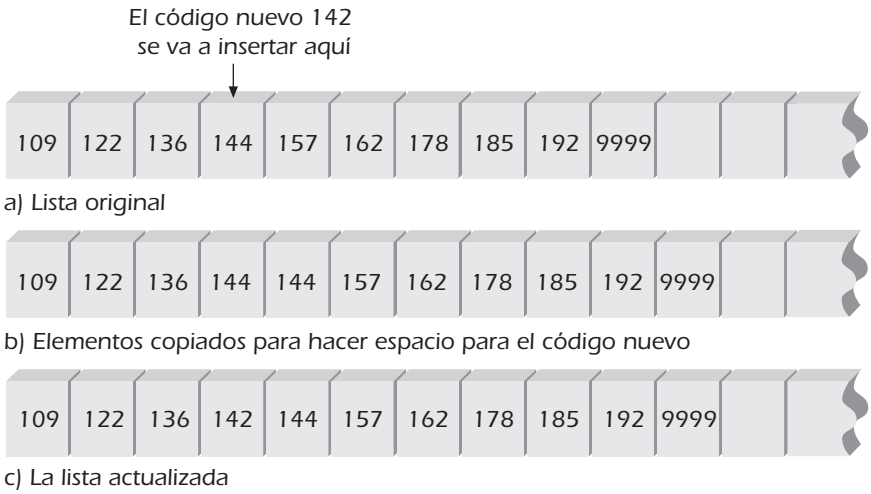


Figura 11.10 Actualización de una lista ordenada de números de identificación.

Paso 3 Codificar la solución

Para este problema se usa el nombre de argumento `idcodigo` para el arreglo transmitido de números de identificación y el nombre de argumento `codigonuevo` para el número de código nuevo que se va a insertar en el arreglo. Aquí, el arreglo transmitido se usa para recibir el arreglo original de números y como arreglo actualizado final. En forma interna respecto a la función, se usará una variable llamada `posnueva` para contener la posición en la lista donde se va a insertar el código nuevo y la variable llamada `posfinal` para contener el valor de la posición del centinela. La variable `i` se usará como un valor índice.

Usando estos nombres de argumento y variable, la función llamada `insert()` ejecuta el procesamiento requerido. Después de aceptar el arreglo y el valor del código nuevo como argumentos, `insert()` ejecuta las cuatro tareas principales descritas en el seudocódigo seleccionado en el paso 2.

```
void insert(int idcodigo[], int codigonuevo)
{
  int i, posnueva, posfinal;

  // encuentra la posición correcta para insertar el código nuevo
  i = 0;
  while (idcodigo[i] < codigonuevo)
    i++;
  if (idcodigo[i] == codigonuevo)
    cout << "\nEste código de identificacion ya esta en la lista";
  else
  {
    posnueva = i;  // encuentra la posición para el código nuevo

    // encuentra el final de la lista
    while (idcodigo[i] != 9999)
      i++;
    posfinal = i;

    // mueve idcodigos una posición
    for (i = posfinal; i >= posnueva; --i)
      idcodigo[i+1] = idcodigo[i];

    // inserta el código nuevo
    idcodigo[posnueva] = codigonuevo;
  }
}
```

La primera tarea de la función es determinar la posición correcta del código nuevo. Esto se hace recorriendo la lista en tanto cada valor encontrado sea menor que el código nuevo. En vista que el valor centinela de 9999 es mayor que cualquier código nuevo, el ciclo debe detenerse cuando se alcance el valor centinela.

Después que se determina la posición correcta, se encuentra la posición del valor centinela, la cual es el último elemento en la lista. Empezando desde esta última posición, cada elemento en la lista se mueve una posición hasta que se alcanza el valor en la posición nueva requerida. Por último, el nuevo código de identificación se inserta en la posición correcta.

Paso 4 Prueba y depuración de la función

El programa 11.7 incorpora la función `insert()` dentro de un programa completo. Esto permite probar la función con los mismos datos usados en el cálculo manual.

Programa 11.7

```
#include <iostream>
using namespace std;

void insert(int [], int);      // prototipo de la función

int main()
{
  const int NUM_MAX = 100;
  int codigonuevo, i;
  int id[NUM_MAX] = {109, 122, 136, 144, 157, 162, 178, 185, 192, 9999};

  cout << "\nIntroduzca el código de identificación nuevo: ";
  cin >> codigonuevo;

  insert(id, codigonuevo);

  cout << "\nLa lista actualizada es:";
  i = 0;
  while(id[i] != 9999)
  {
    cout << " " << id[i];
    i++;
  }
  cout << endl;

  return 0;
}
```

(Continúa)

(Continuación)

```
void insert(int idcodigo[], int codigonuevo)
{
  int i, posnueva, posfinal;

  // encuentra la posición correcta para insertar el código nuevo
  i = 0;
  while (idcodigo[i] < códigonuevo)
    i++;
  if (idcodigo[i] == codigonuevo)
    cout << "\nEste código de identificación ya está en la lista";
  else
  {
    posnueva = i;    // encuentra la posición para el código nuevo

    // encuentra el final de la lista
    while (idcodigo[i] != 9999)
      i++;
    posfinal = i;

    // mueve idcodigos una posición
    for (i = posfinal; i >= posnueva; i--)
      idcodigo[i+1] = idcodigo[i];

    // inserta el código nuevo
    idcodigo[posnueva] = codigonuevo;
  }

  return;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.7:

```
Introduzca el código de identificación nuevo: 142
La lista actualizada es: 109 122 136 142 144 157 162 178 185 192
```

Aunque este resultado concuerda con el cálculo manual previo, no constituye la prueba completa del programa. Para estar seguros que el programa funciona en todos los casos, deberán hacerse ejecuciones de prueba que realicen lo siguiente:

1. Introducir un código de identificación nuevo que duplique un código existente
2. Colocar un código de identificación nuevo al principio de la lista
3. Colocar un código de identificación nuevo al final de la lista

Por último, la restricción del tamaño máximo del arreglo de 100 números de partes designado en la especificación original del problema es poco realista en la práctica. Por lo general, el tamaño máximo nunca se conoce con certeza debido a que cambian condiciones que hacen poco realista el tamaño original. La solución para esta incertidumbre es declarar un arreglo grande que anticipe los posibles cambios, lo cual desperdicia memoria de la computadora, o crear un arreglo que pueda expandirse de manera dinámica y automática conforme se agreguen números de parte nuevos. En la sección 11.6 se presenta la forma en que puede implementarse esta segunda solución.

Ejercicios 11.4

1. Introduzca y ejecute el programa 11.6 en su computadora.
2. Ejecute el programa 11.6 para determinar el promedio y la desviación estándar de la siguiente lista de 15 voltajes: 68, 72, 78, 69, 85, 98, 95, 75, 77, 82, 84, 91, 89, 65, 74.
3. Introduzca y ejecute el programa 11.7 en su sistema de cómputo.
4. **a.** Pruebe el programa 11.7 usando un número de parte de 86, lo cual deberá colocar este código nuevo al principio de la lista existente.

   b. Pruebe el programa 11.7 usando un número de parte de 200, lo cual deberá colocar este número de parte nuevo al final de la lista existente.
5. **a.** Determine un algoritmo para eliminar una entrada de una lista ordenada de números.

   b. Escriba una función, `delete()`, la cual usa el algoritmo seleccionado en el ejercicio 5a, para eliminar un código de identificación de la lista de números ilustrados en la figura 11.10a.
6. Suponga que las siguientes letras son almacenadas en un arreglo de alfabeto: B, J, K, M, S, Z. Escriba y pruebe una función, `agregarletra()`, que acepte el arreglo de alfabeto y una letra nueva como argumentos y luego inserte la letra nueva en el orden alfabético correcto en el arreglo de alfabeto.

11.5 ARREGLOS COMO ARGUMENTOS

Los elementos del arreglo individuales son transmitidos a una función llamada de la misma manera que las variables escalares individuales; tan sólo se incluyen como variables subindexadas cuando se hace la llamada a la función. Por ejemplo, la llamada a la función

```
hallarMin(voltios[2], voltios[6]);
```

transmite los valores de los elementos `voltios[2]` y `voltios[6]` a la función `hallarMin()`.

Transmitir un arreglo completo de valores a una función es en muchos aspectos una operación más fácil que transmitir elementos individuales. La función llamada recibe acceso al arreglo real, en lugar de a una copia de los valores en el arreglo. Por ejemplo, si `voltios` es un arreglo, la llamada a la función `hallarMax(voltios)`; hace que el arreglo `voltios` completo esté disponible para la función `hallarMax()`. Esto es diferente al transmitir una sola variable a una función.

Se recordará que cuando se transmite un solo argumento escalar a una función, la función llamada sólo recibe una copia del valor transmitido, el cual es almacenado en uno de los parámetros de la función. Si los arreglos fueran transmitidos de esta manera, tendría que crearse una copia del arreglo completo. Para arreglos grandes, hacer copias duplicadas del arreglo para cada llamada a la función sería un desperdicio de almacenamiento de computadora y frustraría el esfuerzo de devolver cambios a elementos múltiples hechos por el programa invocado. (Se recordará que una función devuelve cuando mucho un valor directo.) Para evitar estos problemas, a la función llamada se le da acceso directo al arreglo original.[4] Por tanto, los cambios hechos por la función llamada se hacen en forma directa al arreglo en sí. Para los siguientes ejemplos específicos de llamadas a función, suponga que los arreglos `nums, claves, voltios` y `corriente` son declarados como:

```
int nums[5];              // un arreglo de cinco números enteros
char claves[256];         // un arreglo de 256 caracteres
double voltios[500], corriente[500];   // dos arreglos de 500
                                          números de precisión
                                          doble
```

Para estos arreglos, pueden hacerse las siguientes llamadas a función:

```
hallarMax(nums);
hallarCh(claves);
calcTot(nums, voltios, corriente);
```

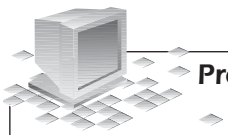
En cada caso, la función llamada recibe acceso directo al arreglo nombrado.

En el lado receptor, la función llamada debe ser alertada de que un arreglo está disponible. Por ejemplo, son líneas adecuadas para el encabezado de la función para las funciones previas

```
int hallarMax(int vals[5])
char hallarCh(char in_claves[256])
void calcTot(int arr1[5], double arr2[500], double arr3[500])
```

En cada una de estas líneas de encabezado de la función, los nombres en la lista de parámetros son elegidos por el programador. Sin embargo, los nombres de parámetros usados por las funciones todavía se refieren al arreglo original creado fuera de la función. Esto queda claro en el programa 11.8.

[4]Esto se logra debido a que la dirección inicial del arreglo en realidad se transmite como un argumento. El parámetro formal que recibe este argumento de dirección es un apuntador. La relación íntima entre los nombres de arreglos y apuntadores se presenta en el capítulo 12.



Programa 11.8

```
#include <iostream>
using namespace std;

const int MAXELS = 5;
int hallarMax(int [MAXELS]);       // prototipo de la función

int main()
{
  int nums[MAXELS] = {2, 18, 1, 27, 16};

  cout << "El valor máximo es " << hallarMax(nums) << endl;

  return 0;
}

// halla el valor máximo
int hallarMax(int vals[MAXELS])
{
  int i, max = vals[0];
  for (i = 1; i < MAXELS; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

Hay que observar que el prototipo de la función para `hallarMax()` declara que `hallarMax` devolverá un número entero y espera un arreglo de cinco números enteros como un argumento real. También es importante saber que sólo se crea un arreglo en el programa 11.9. En `main()` este arreglo se conoce como `nums`, y en `hallarMax` el arreglo se conoce como `vals`. Como se ilustra en la figura 11.11, ambos nombres se refieren al mismo arreglo. Por tanto, en la figura 11.11 `vals[3]` es el mismo elemento que `nums[3]`.

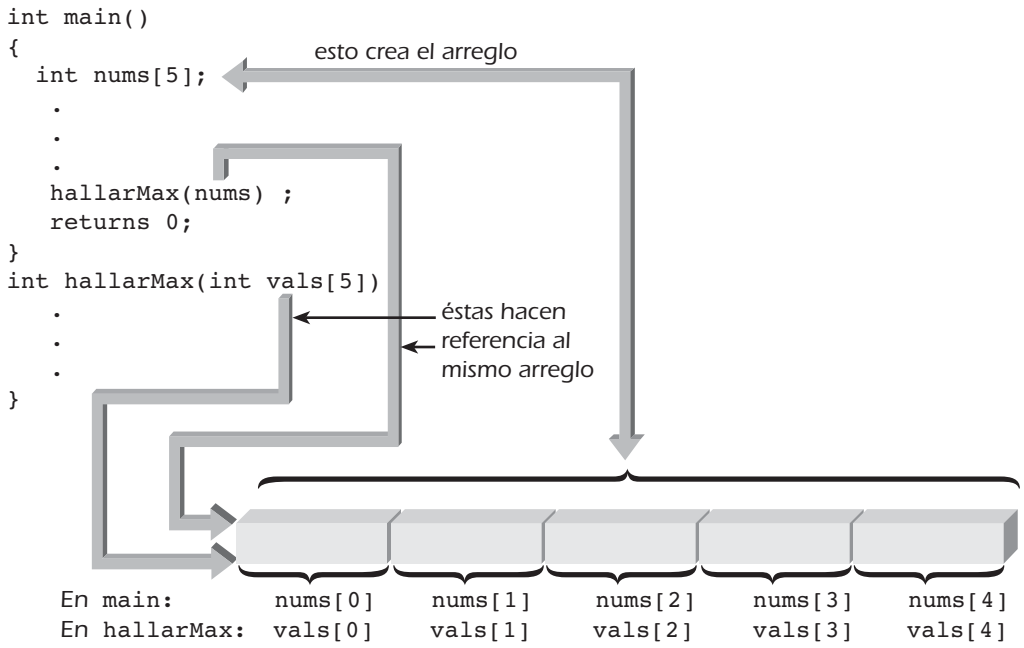


Figura 11.11 *Sólo se crea un arreglo.*

La declaración de parámetros en la línea de encabezado `hallarMax()` en realidad contiene información extra que la función no requiere. Todo lo que debe saber `hallarMax()` es que el parámetro `vals` hace referencia a un arreglo de números enteros. Dado que el arreglo se ha creado en `main()` y no se necesita espacio de almacenamiento adicional en `hallarMax()`, la declaración para `vals` puede omitir el tamaño del arreglo. Por tanto, una línea de encabezado de función alternativa es

```
int hallarMax(int vals[])
```

Esta forma del encabezado de la función tiene más sentido cuando nos damos cuenta que sólo un elemento es transmitido en realidad a `hallarMax` cuando se llama a la función, la dirección inicial del arreglo `nums`. Esto se ilustra en la figura 11.12.

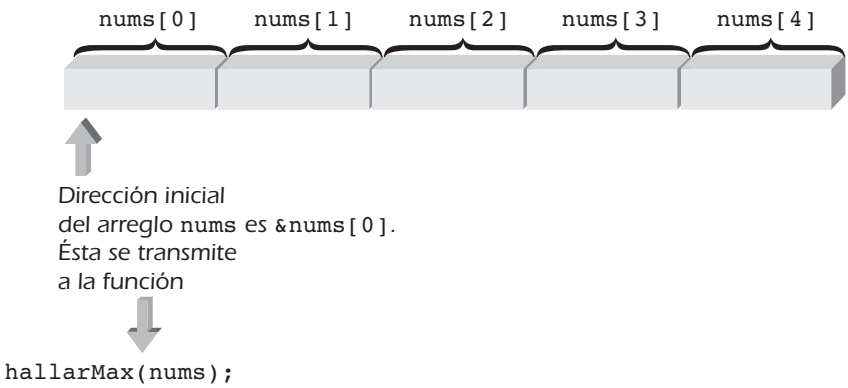


Figura 11.12 *Se transmite la dirección inicial del arreglo.*

En vista que sólo la dirección inicial de `vals` es transmitida a `hallarMax`, el número de elementos en el arreglo no necesita incluirse en la declaración para `vals`.[5] De hecho, por lo general es aconsejable omitir el tamaño del arreglo en la línea de encabezado de la función. Por ejemplo, considérese la forma más general de `hallarMax()`, la cual puede usarse para encontrar el valor máximo de un arreglo de números enteros de tamaño arbitrario.

```
int hallarMax(int vals[], int numels)   // encuentra el valor máximo
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
   if (max < vals[i])
     max = vals[i];

  return max;
}
```

La forma más general de `hallarMax()` declara que la función devuelve un valor entero. La función espera la dirección inicial de un arreglo de números enteros y el número de elementos en el arreglo como argumentos. Entonces, usando el número de elementos como el límite para su búsqueda, el ciclo `for` de la función causa que cada elemento del arreglo sea examinado en orden secuencial para localizar el valor máximo. El programa 11.9 ilustra el uso de `hallarMax()` en un programa completo.

La salida desplegada tanto por el programas 11.8 como por el 11.9 es

```
El valor máximo es 27
```

Transmitir arreglos bidimensionales a una función es un proceso idéntico a transmitir arreglos unidimensionales. La función llamada recibe acceso al arreglo entero. Por ejemplo, suponiendo que `val` es un arreglo bidimensional, la llamada a la función `desplegar(val);` hace que el arreglo `val` completo esté disponible para la función llamada `desplegar()`. Por tanto, los cambios hechos por `desplegar()` se harán de manera directa al arreglo `val`. Como ejemplos adicionales, suponga que los siguientes arreglos bidimensionales llamados prueba, factores y empuje se declaran como:

```
int prueba[7][9];
float factores[26][10];
double empuje[256][52];
```

entonces las siguientes llamadas a función son válidas:

```
hallarMax(prueba);
obtener(factores);
promedio(empuje);
```

[5]Una consecuencia importante de esto es que `hallarMax()` tiene acceso directo al arreglo transmitido. Esto significa que cualquier cambio a un elemento del arreglo `vals` es un cambio al arreglo `nums`. Esto es diferente de manera significativa de la situación con las variables escalares, donde la función llamada no recibe acceso directo a la variable transmitida.

Programa 11.9

```
// este programa despliega un mensaje
#include <iostream>
using namespace std;

int hallarMax(int [], int);    // prototipo de la función

int main()
{
  const int MAXELS = 5;
  int nums[MAXELS] = {2, 18, 1, 27, 16};

  cout << "El valor máximo es "
       << hallarMax(nums, MAXELS) << endl;

  return 0;
}

// encuentra el valor máximo
int hallarMax(int vals[], int numels)
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

En el lado receptor, debe alertarse a la función llamada que un arreglo bidimensional queda disponible. Por ejemplo, suponiendo que cada una de las funciones anteriores devuelve un número entero, son líneas de encabezado de función adecuadas para estas funciones:

```
int hallarMax(int nums[7][9])
int obtener(float valores[26][10])
int promedio(double vals[256][52])
```

En cada una de estas líneas de encabezado de función, los nombres de parámetros elegidos se usarán de manera interna en el cuerpo de la función. Sin embargo, estos nombres de parámetros se referirán aún al arreglo original creado fuera de la función. El programa 11.10 ilustra la transmisión de un arreglo bidimensional a una función que despliega los valores del arreglo.

Programa 11.10

```
#include <iostream>
#include <iomanip>
using namespace std;

const int FILAS = 3;
const int COLS = 4;
void desplegar(int [FILAS][COLS]);    // prototipo de la función

int main()
{
  int val[FILAS][COLS] = {8,16,9,52,
                          3,15,27,6,
                         14,25,2,10};

  desplegar(val);

  return 0;
}

void desplegar(int nums[FILAS][COLS])
{
  int num_fila, num_col;
  for (num_fila = 0; num_fila < FILAS; num_fila++)
  {
    for (num_col = 0; num_col < COLS; num_col++)
      cout << setw(4) << nums[num_fila][num_col];
    cout << endl;
  }

  return;
}
```

Sólo un arreglo se crea en el programa 11.10. Este arreglo se conoce como `val` en `main()` y como `nums` en `desplegar()`. Por tanto, `val[0][2]` se refiere al mismo elemento que `nums[0][2]`.

Hay que observar el uso del ciclo `for` anidado en el programa 11.10 para recorrer cada elemento del arreglo. En el programa 11.10, la variable `num_fila` controla el ciclo exterior y la variable `num_col` controla el ciclo interior. Para cada recorrido del ciclo exterior, el cual corresponde a una sola fila, el ciclo interior hace un recorrido por los elementos de columna. Después que se imprime una fila completa, se inicia una línea nueva para la siguiente fila. El efecto es un despliegue del arreglo en una forma fila por fila:

```
   8  16   9  52
   3  15  27   6
  14  25   2  10
```

La declaración de parámetros para `nums` en `desplegar()` contiene información adicional que no es requerida por la función. La declaración para `nums` puede omitir el tamaño de la filas del arreglo. Por tanto, un prototipo de la función alternativo es

```
desplegar(int nums[][4]);
```

La razón por la que debe incluirse el tamaño de las columnas mientras el tamaño de las filas es opcional se hace evidente cuando se considera cómo se almacenan los elementos del arreglo en la memoria. Empezando con el elemento `val[0][0]`, cada elemento subsiguiente se almacena de manera consecutiva, fila por fila, como `val[0][0], val[0][1], val[0][2], val[0][3], val[1][0], val[1][1]`, etc., como se ilustra en la figura 11.13.

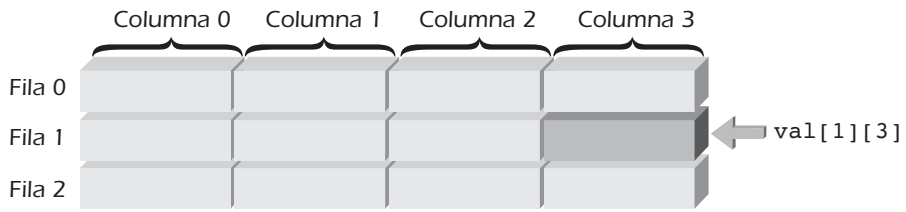
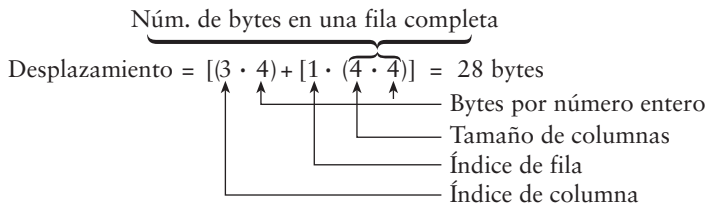


Figura 11.13 *Almacenamiento del arreglo* `val`.

Como ocurre con todos los accesos a arreglos, un elemento individual del arreglo `val` se obtiene sumando un desplazamiento a la ubicación inicial del arreglo. Por ejemplo, el elemento `val[1][3]` del arreglo `val` ilustrado en la figura 11.13 se localiza con un desplazamiento de 14 bytes desde el inicio del arreglo. Internamente, el compilador usa el índice de fila, el índice de columna y el tamaño de las columnas para determinar este desplazamiento usando el siguiente cálculo (una vez más, suponiendo cuatro bytes para un `int`):



El tamaño de las columnas es necesario en el cálculo del desplazamiento, de modo que el compilador pueda determinar el número de posiciones que se va a saltar con el fin de obtener la fila deseada.

Ejercicios 11.5

1. La siguiente declaración se usó para crear el arreglo `voltios`:

   ```
   int voltios[500];
   ```

   Escriba dos líneas de encabezado de función diferentes para una función llamada `ordenarArreglo()` que acepte el arreglo `voltios` como un parámetro llamado `enArreglo`.

2. La siguiente declaración se usó para crear el arreglo `factores`:

```
double factores[256];
```

Escriba dos líneas de encabezado de función diferentes para una función llamada `hallarClave()` que acepte el arreglo factores como un parámetro llamado `seleccionar`.

3. La siguiente declaración se usó para crear el arreglo `potencia`:

```
double potencia[256];
```

Escriba dos líneas de encabezado de función diferentes para una función llamada `principal()` que acepte el arreglo `potencia` como un argumento llamado `tasas`.

4. ***a.*** Modifique la función `hallarMax()` en el programa 11.8 para localizar el valor mínimo del arreglo transmitido.
b. Incluya la función escrita en el ejercicio 4a en un programa completo y ejecute el programa en una computadora.

5. Escriba un programa que tenga una declaración en `main()` para almacenar los siguientes números en un arreglo llamado `temps`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, 10.0. Deberá haber una llamada a la función `mostrar()` que acepte el arreglo `temps` como un parámetro llamado `temps` y luego despliegue los números en el arreglo.

6. Escriba un programa que declare tres arreglos unidimensionales llamados `voltios`, `corriente` y `resistencia`. Cada arreglo deberá declararse en `main()` y deberá ser capaz de contener diez números de precisión doble. Los números que deberán almacenarse en `corriente` son 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, 3.98. Los números que deberán almacenarse en `resistencia` son 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, 4.8. Su programa deberá transmitir estos tres arreglos a una función llamada `calc_voltios()`, la cual deberá calcular los elementos en el arreglo `voltios` como el producto de los elementos correspondientes en los arreglos `corriente` y `resistencia` (por ejemplo, `voltios[1] = corriente[1] * resistencia[1]`). Después que `calc_voltios()` ha puesto valores en el arreglo `voltios`, los valores en el arreglo deberán desplegarse desde adentro de `main()`.

7. Escriba un programa que incluya dos funciones llamadas `calc_prom()` y `varianza()`. La función `calc_prom()` deberá calcular y devolver el promedio de los valores almacenados en el arreglo llamado `valores_prueba`. El arreglo deberá ser declarado en `main()` e incluir los valores 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. La función `varianza()` deberá calcular y devolver la varianza de los datos. La varianza se obtiene restando el promedio de cada valor en `valores_prueba`, elevando al cuadrado los valores obtenidos, sumándolos y dividiéndolos entre el número de elementos en `valores_prueba`. Los valores devueltos de `calc_promedio()` y `varianza()` deberán desplegarse usando instrucciones `cout` en `main()`.

11.6 LA CLASE DE VECTOR STL

Muchas aplicaciones de programación requieren que se expandan y contraigan listas conforme se agregan y eliminan elementos de la lista. Aunque puede lograrse expandir y contraer un arreglo creando, copiando y eliminando arreglos, esta solución es costosa en función de la programación inicial, mantenimiento y tiempo de prueba. Para satisfacer la necesidad de proporcionar un conjunto de estructuras de datos probado y genérico que pueda modificarse, expandirse y contraerse, C++ proporciona un conjunto útil de clases en su Biblioteca Estándar de Plantillas (STL, por sus siglas en inglés)

Cada clase STL es codificada como una plantilla (véase la sección 6.1) que permite la construcción de una estructura de datos genérica, la cual se conoce como **contenedor**. Los términos **lista** y **colección** son sinónimos de contenedor, con cada término refiriéndose a un conjunto de elementos de datos que forma una unidad o grupo natural. Usando esta definición, un arreglo es un contenedor; sin embargo, un contenedor se proporciona como un tipo integrado en contraste con los contenedores creados usando STL.

En esta sección se presenta la clase contenedora de vector STL. Un vector se parece a un arreglo en que almacena elementos a los que se puede tener acceso usando un índice de números enteros que comience en cero, pero es diferente en que un vector se expandirá de manera automática conforme sea necesario y es proporcionado por diversos métodos (funciones) de clase útiles para operar en el vector. La tabla 11.1 enlista estos métodos de clase de vector, resaltando los métodos que se usarán en el programa de demostración.

Tabla 11.1 Resumen de métodos (funciones) y operaciones de clase de vector

Funciones (métodos de clase) y operaciones	Descripción
`vector<TipoDatos> nombre`	Crea un vector vacío con tamaño inicial dependiente del compilador
`vector<TipoDatos> nombre(fuente)`	Crea una copia del vector fuente
`vector<TipoDatos> nombre(`*n*`)`	Crea un vector de tamaño *n*
`vector<TipoDatos> nombre(`*n*`, `*elem*`)`	Crea un vector de tamaño *n* con cada elemento inicializado como `elem`
`vector<TipoDatos> nombre(src.beg, src.end)`	Crea un vector inicializado con elementos de un contenedor fuente que comienza en `src.beg` y termina en `src.end`
`~vector<TipoDatos>()`	Destruye el vector y todos los elementos que contiene
`nombre[índice]`	Devuelve el elemento en el índice designado, sin comprobación de límites
`nombre.at(índice)`	Devuelve el elemento en el argumento del índice especificado, sin comprobación de límites en el valor del índice
`nombre.front()`	Devuelve el primer elemento en el vector
`nombre.back()`	Devuelve el último elemento en el vector
`dest = src`	Asigna todos los elementos del vector `src` al vector `dest`

Tabla 11.1 Resumen de métodos (funciones) y operaciones de clase de vector (continuación)

Funciones (métodos de clase) y operaciones	Descripción
`nombre.assign(n, elem)`	Asigna *n* copias de `elem`
`nombre.assign(src.begin, src.end)`	Asigna los elementos del contenedor src (no necesita ser un vector), entre el rango `src.begin` y `src.end`, al vector nombre
`insertar(pos, elem)`	Inserta elem en la posición pos
`nombre.insertar(pos, n, elem)`	Inserta *n* copias de elem empezando en la posición *pos*
`nombre.insertar(pos, src.begin, src.end)`	Inserta
`nombre.push_back(elem)`	Añade `elem` al final del vector
`nombre.erase(pos)`	Elimina el elemento en la posición especificada
`nombre.erase(begin, end)`	Elimina los elementos dentro del rango especificado
`nombre.resize(valor)`	Redimensiona el vector a un tamaño mayor, con elementos nuevos instanciados usando el constructor por omisión
`nombre.resize(valor, elem)`	Redimensiona el vector a un tamaño mayor, con elementos nuevos instanciados como `elem`
`nombre.clear()`	Elimina todos los elementos del vector
`nombre.swap(nombreB)`	Intercambia los elementos de los vectores `nombreA` y `nombreB`; puede ejecutarse usando el algoritmo `swap()`
`nombreA == nombreB`	Devuelve un valor `verdadero` booleano si los elementos de `nombreA` son iguales a los elementos de `nombreB`; de lo contrario, devuelve un valor `falso`
`nombreA != nombreB`	Devuelve un valor `falso` booleano si los elementos de `nombreA` son iguales a los elementos de `nombreB`; de lo contrario, devuelve valor `verdadero`; igual que `!(nombreA == nombreB)`
`nombreA < nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es menor que nombreB; de lo contrario, devuelve un valor `falso`
`nombreA > nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es mayor que `nombreB`; de lo contrario, devuelve un valor `falso`; igual que `nombreB < nombreA`
`nombreA <= nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es menor que o igual a `nombreB`
`nombreA >= nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es mayor que o igual a `nombreB`
`nombre.size()`	Devuelve el tamaño de un vector
`nombre.empty()`	Devuelve un valor verdadero booleano si el vector está vacío; de lo contrario, devuelve un valor falso
`nombre.max_size()`	Devuelve el máximo posible de elementos como un número entero
`nombre.capacity()`	Devuelve el máximo posible de elementos, como un número entero, sin reubicación del vector

Además de los métodos de clase de vector específicos mostrados en la tabla 11.1, los vectores tienen acceso al conjunto completo de funciones STL genéricas, conocidas en STL como algoritmos. La tabla 11.2 resume los más usados de estos algoritmos.

Tabla 11.2 Algoritmos de la Biblioteca Estándar de Plantillas (STL) más usados

Nombre del algoritmo	Descripción
`accumulate`	Devuelve la suma de los números en un rango especificado
`binary_search`	Devuelve un valor booleano de `verdadero` si el valor especificado existe dentro del rango especificado; de lo contrario, devuelve `falso`. Sólo puede usarse en un conjunto de valores ordenado
`copy`	Copia elementos de un rango de origen a un rango de destino
`copy_backward`	Copia elementos de un rango de origen a un rango de destino en una dirección inversa
`count`	Devuelve el número de elementos en un rango especificado que corresponde a un valor especificado
`equal`	Compara los elementos en un rango de elementos, elemento por elemento, con los elementos en un segundo rango
`fill`	Asigna todos los elementos en un rango especificado a un valor especificado
`find`	Devuelve la posición de la primera ocurrencia de un elemento en un rango especificado que tenga un valor especificado si el valor existe. Ejecuta una búsqueda lineal, empezando con el primer elemento en un rango especificado y procede un elemento a la vez hasta que se ha buscado en el rango completo o se ha encontrado el elemento especificado
`max_element`	Devuelve el valor máximo de los elementos en el rango especificado
`min_element`	Devuelve el valor mínimo de los elementos en el rango especificado
`random_shuffle`	Revuelve al azar los valores de los elementos en un rango especificado
`remove`	Elimina un valor especificado dentro de un rango especificado sin cambiar el orden de los elementos restantes
`replace`	Reemplaza cada elemento en un rango especificado que tiene un valor especificado con un valor recién especificado
`reverse`	Invierte los elementos en un rango especificado
`search`	Encuentra la primera ocurrencia de un valor o secuencia de valores especificados dentro de un rango especificado
`sort`	Ordena los elementos en un rango especificado en orden ascendente
`swap`	Intercambia valores de elementos entre dos objetos
`unique`	Elimina elementos adyacentes duplicados dentro de un rango especificado

Hay que observar que hay un algoritmo `swap` (tabla 11.2) y un método de clase de vector `swap` (tabla 11.1). Debido a que el método de clase está dirigido a trabajar de manera específica con su tipo contenedor y por lo general se ejecutará más rápido siempre que una clase contenedora proporcione un método con el mismo nombre como un algoritmo, deberá usarse el método de clase.

Por último, diversos elementos adicionales, llamados iteradores, también son proporcionados por la STL. Los iteradores proporcionan el medio para especificar cuáles elementos en un contenedor han de ser operados o cuando se invoca a un algoritmo. Dos de los iteradores más útiles son devueltos por las funciones de iterador de la STL llamadas `begin()` y `end()`. Éstas son funciones de propósito general que devuelven las posiciones del primer y último elementos en un contenedor, respectivamente.

Punto de información

Cuándo usar un arreglo o un vector

Un arreglo es la estructura de datos de primera elección siempre que se tenga una lista de tipos de datos u objetos primitivos que no tengan que expandirse o contraerse.

Un vector es la estructura de datos de primera elección siempre que se tenga una lista de tipos de datos u objetos primitivos que puedan agruparse como un arreglo, pero que deban expandirse o contraerse.

Siempre que sea posible, siempre deben usarse algoritmos STL para operar en arreglos (véase la sección 12.4) y vectores (descritos en esta sección). Tanto las clases como los algoritmos de la STL proporcionan código verificado y confiable que puede acortar el tiempo de desarrollo del programa.

Para hacer esto más tangible y proporcionar una introducción significativa al uso de una clase contenedora STL, se usará la clase contenedora de vector para crear un vector que contenga una lista de números de partes. Como se verá, un vector es similar a un arreglo en C++, excepto que puede expandirse de manera automática según sea necesario.

El programa 11.11 construye un vector y lo inicializa con números enteros almacenados en un arreglo de números enteros. Una vez que se ha inicializado, se usan varios métodos de vector y algoritmos STL para operar en el vector. De manera específica, se usa un método para cambiar un valor existente, un segundo método se usa para insertar un valor dentro del vector y un tercer método se usa para añadir un valor al final de la lista. Después que se aplica cada método y algoritmo, se emplea un objeto `cout` para desplegar los resultados.

Programa 11.11

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  const int NUMELS = 4;
  int n[] ={136, 122, 109, 146};
  int i;

  // instancia un vector de cadenas usando el arreglo n[]
  vector<int> nums_partes(n, n + NUMELS);
```

(Continúa)

(Continuación)

```
    cout << "\nEl vector inicialmente tiene un tamanio de "
         << int(nums_partes.size()) << ",\n y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // modifica el elemento en la posicion 4 (es decir, indice = 3) en el vector
    nums_partes[3] = 144;
    cout << "\n\nDespués de reemplazar el cuarto elemento, el vector tiene
            un tamanio de "
         << int(nums_partes.size()) << ",\n y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // inserta un elemento en el vector en la posición 2 (es decir, indice = 1)
    nums_partes.insert(nums_partes.begin()+1, 142);
    cout << "\n\nDespués de insertar un elemento en la segunda posición,"
         << "\n el vector tiene un tamanio de " << int(nums_partes.size()) << ","
         << " y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // agrega un elemento al final del vector
    nums_partes.push_back(157);
    cout << "\n\nDespués de agregar un elemento al final de la lista,"
         << "\n el vector tiene un tamanio de " << int(nums_partes.size()) << ","
         << " y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // ordena el vector
    sort(nums_partes.begin(), nums_partes.end());
    cout << "\n\nDespués de ordenarlos, los elementos del vector son:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    cout << endl;

    return 0;
}
```

Al revisar el programa 11.11, se puede que se han incluido las cuatro líneas de encabezado `<iostream>, <string>, <vector>` y `<algorithm>` y la instrucción `using namespace std;`. El encabezado `<iostream>` es necesario para crear y usar el flujo `cout`, el encabezado `<string>` se requiere para construir cadenas, el encabezado `<vector>` se usa para crear o mover objetos `vector` y el encabezado `<algorithm>` se requiere para el algoritmo `sort` que se aplica después que se ha completado la adición y reemplazo de elementos de `vector`.

La instrucción en el programa 11.11 usada para crear e inicializar el vector llamado `nums_partes` es la siguiente:

```
vector<int> nums_partes(n, n + NUMELS);
```

Aquí, el vector `nums_partes` es declarado como un `vector` del tipo `int` y es inicializado con elementos del arreglo n, comenzando con el primer elemento del arreglo `(elemento n[0])` y terminando con el último elemento del arreglo, el cual se localiza en la posición `n + NUMELS`. Por tanto, el vector tiene un tamaño suficiente para cuatro valores enteros y se ha inicializado con los números enteros `136, 122, 109` y `146`. El siguiente conjunto de instrucciones en el programa 11.11 despliega los valores iniciales en el vector usando notación de vector subindexada estándar que es idéntica a la notación usada para tener acceso a los elementos del arreglo. Sin embargo, desplegar los valores del vector de esta manera requiere saber cuántos elementos contiene cada vector. Conforme se insertan y eliminan elementos, nos gustaría que el vector rastreara la ubicación del primer y último elementos. Esta capacidad es proporcionada de manera automática por dos métodos suministrados para cada contenedor STL: `begin()` y `end()`.

El siguiente conjunto importante de instrucciones consiste en lo siguiente:

```
// modifica el elemento en la posición 4 (es decir, indice = 3) en
   el vector
  nums_partes[3] = 144;


  // inserta un elemento en el vector en la posición 2 (es decir,
     indice = 1)
  nums_partes.insert(nums_partes.begin()+1, 142);
```

Estas instrucciones se usan para modificar un valor de vector existente e insertar un valor nuevo en el vector. De manera específica, la notación `nums_partes[3]` usa indexación estándar, y el método `insert()` usa un argumento que se conoce en STL como iterador. Dichos argumentos se construyen como desplazamientos usando las funciones `begin()` y `end()`. Además, se tiene que especificar el valor que se va a insertar en la posición designada. Por tanto, `nombres[3]` especifica que se cambiará el cuarto elemento en el vector. (Los vectores, como los arreglos, comienzan en la posición índice 0.) El método `insert()` se usa para insertar el valor entero 142 en la segunda posición del vector. Debido a que el método `begin()` devuelve un valor correspondiente al inicio del vector, agregarle 1 designa la segunda posición en el vector.[6] Es en esta posición donde se inserta el valor nuevo, todos los valores subsiguientes se mueven una posición en el vector y el vector se expande de manera automática para aceptar el valor insertado. En este punto en el programa, el vector `nums_partes` contiene ahora los siguientes elementos:

```
136  142  122  109  144
```

Esta ordenación de valores se obtuvo al reemplazar el valor original 146 con 144 e insertando el 142 en la segunda posición, lo cual en forma automática mueve todos los elementos subsiguientes una posición e incrementa el tamaño total del vector a cinco números enteros.

[6] Con más precisión, este método requiere un argumento *iterador*, no un argumento índice en número entero. Los algoritmos `begin()` y `end()` devuelven iteradores, a los cuales se les pueden aplicar desplazamientos. En esto, son parecidos a los apuntadores (que se presentan en el siguiente capítulo).

A continuación, se usa la instrucción `nums_partes.push_back(157);` para añadir el número entero 157 al final del vector, lo cual produce los siguientes elementos:

```
136  142  122  109  144  157
```

Por último, la sección final del código usado en el programa 11.11 usa el algoritmo `sort()` para ordenar los elementos en el vector. Después que se aplica el algoritmo, se despliegan de nuevo los valores en el vector. A continuación se presenta la salida completa producida por el programa 11.11:

```
El vector inicialmente tiene un tamanio de 4,
 y contiene los elementos:
136   122   109   146

Después de reemplazar el cuarto elemento, el vector tiene un
 tamanio de 4, y contiene los elementos:
136  122  109  144

Después de insertar un elemento en la segunda posición,
 el vector tiene un tamanio de 5, y contiene los elementos:
136  142  122  109  144

Después de agregar un elemento al final de la lista,
  el vector tiene un tamanio de 6, y contiene los elementos:
136  142  122  109  144  157

Después de ordenarlos, los elementos del vector son:
109  122  136  142  144  157
```

Ejercicios 11.6

1. Defina los términos “contenedor” y “Biblioteca Estándar de Plantillas”.
2. ¿Qué instrucciones `include` deberían incluirse con programas que usan la Biblioteca Estándar de Plantillas?
3. Introduzca y ejecute el programa 11.11.
4. Modifique el programa 11.11 de modo que el usuario introduzca el conjunto de números inicial cuando se ejecute el programa. Haga que el programa solicite la cantidad de números iniciales que se van a introducir.
5. Modifique el programa 11.11 para usar y desplegar los resultados reportados por los métodos `capacity ()` y `max_size()` de la clase de vector.
6. Modifique el programa 11.11 para usar el algoritmo `random_shuffle()`.
7. Modifique el programa 11.11 para usar los algoritmos `binary_search()` y `find()`. Haga que su programa solicite el número que se va a encontrar.
8. Usando el programa 11.11 como punto de partida, cree un programa equivalente que use un vector de cadenas. Inicialice el vector usando el arreglo `string nombres[] = {"Donovan", "Michaels", "Smith", "Jones"};`.

9. Use los algoritmos `max_element()` y `min_element()` para determinar los valores máximo y mínimo, respectivamente, en el vector creado para el ejercicio 8. (*Sugerencia*: Use la expresión `max_element(nombreVector.begin(), nombreVector.end())` para determinar el valor máximo almacenado en el vector. Luego use los mismos argumentos para el algoritmo `min_element()`.)

10. Vuelva a hacer la aplicación 2 en la sección 11.4 usando un vector en lugar de un arreglo.

11.7 ERRORES COMUNES DE PROGRAMACIÓN

Cuatro errores comunes se asocian con el uso de arreglos.

1. Olvidar declarar el arreglo. Este error produce un mensaje de error del compilador equivalente a "indirección inválida" cada vez que se encuentre una variable subindexada dentro de un programa. El significado exacto de este mensaje de error se aclarará cuando se establezca la correspondencia entre arreglos y apuntadores en el capítulo 12.
2. Usar un subíndice que haga referencia a un elemento del arreglo inexistente. Por ejemplo, declarar que el arreglo es de tamaño 20 y usar un valor subíndice de 25. Este error no es detectado por la mayor parte de los compiladores de C++. Sin embargo, producirá un error en tiempo de ejecución que causará una caída del programa o que se tenga acceso en la memoria a un valor que no tiene relación con el elemento pretendido. En cualquier caso, por lo general es un error difícil en extremo de localizar. La única solución para este problema es asegurarse, ya sea por instrucciones de programación específicas o por una codificación cuidadosa, que cada subíndice hace referencia a un elemento del arreglo válido.
3. No usar un valor contador lo bastante grande en un contador de ciclo `for` para recorrer todos los elementos del arreglo. Este error ocurre por lo general cuando en un principio se especifica que un arreglo es de tamaño *n* y hay un ciclo `for` dentro del programa de la forma `for (i = 0; i < n; i++)`. El tamaño del arreglo es expandido luego, pero el programador olvida cambiar los parámetros del ciclo for interior. Declarar un tamaño de arreglo usando una constante nombrada y usar de manera consistente la constante nombrada a lo largo de la función en lugar de la variable *n*, la cual elimina este problema.
4. Olvidar inicializar el arreglo. Aunque muchos compiladores establecen de manera automática todos los elementos de arreglos valorados en números enteros y reales en cero, y todos los elementos de arreglos de carácter en espacios en blanco, le corresponde al programador asegurarse que cada arreglo es inicializado en forma correcta antes que comience el procesamiento de los elementos del arreglo.

11.8 RESUMEN DEL CAPÍTULO

1. Un arreglo unidimensional es una estructura de datos que puede utilizarse para almacenar una lista de valores del mismo tipo de datos. Tales arreglos deben declararse dando el tipo de datos de los valores que están almacenados en el arreglo y el tamaño del arreglo. Por ejemplo, la declaración:

   ```
   int num[100];
   ```

   crea un arreglo de 100 números enteros. Un enfoque más adecuado es usar primero una constante nombrada para establecer el tamaño del arreglo, y luego utilizar esta constante en la definición del arreglo. Por ejemplo,

   ```
   const int TAMANIOMAX = 100;
   ```

   and

   ```
   int num[TAMANIOMAX];
   ```

2. Los elementos del arreglo son almacenados en ubicaciones contiguas en la memoria y se referencian usando el nombre del arreglo y un subíndice, por ejemplo, `num[22]`. Puede utilizarse cualquier expresión de valor en número entero no negativo como subíndice y el subíndice 0 siempre se refiere al primer elemento en un arreglo.

3. Un arreglo bidimensional se declara enlistando un tamaño de fila y uno de columna con el tipo de datos y el nombre del arreglo. Por ejemplo, la declaración

   ```
   int mat[5][7];
   ```

   crea un arreglo bidimensional consistente de cinco filas y siete columnas de valores enteros.

4. Los arreglos pueden inicializarse cuando se declaran. Para arreglos bidimensionales esto se logra enlistando los valores iniciales, fila por fila, dentro de llaves y separándolos con comas. Por ejemplo, la declaración

   ```
   int vals[3][2] = { {1, 2},
                      {3, 4},
                      {5, 6} };
   ```

   produce el siguiente arreglo de 3 filas por 2 columnas:

   ```
   1  2
   3  4
   5  6
   ```

   Como C++ usa la convención de que la inicialización procede en un orden por fila, las llaves interiores pueden omitirse. Por tanto, una inicialización equivalente es proporcionada por la instrucción:

   ```
   int vals[3][2] = { 1, 2, 3, 4,  5, 6};
   ```

5. Los arreglos son transmitidos a una función pasando el nombre del arreglo como un argumento. El valor transmitido en realidad es la dirección de la primera ubicación de almacenamiento del arreglo. Por tanto, la función linvocada recibe acceso directo al arreglo original y no a una copia de los elementos del arreglo. Dentro de la función llamada debe declararse un argumento formal para que reciba el nombre del arreglo transmitido. La declaración del argumento formal puede omitir el tamaño de filas del arreglo.

11.9 APÉNDICE DEL CAPÍTULO: BÚSQUEDA Y ORDENAMIENTO

La mayoría de los programadores se encuentran con la necesidad de ordenar y buscar en una lista de elementos de datos en algún momento en sus carreras de programación. Por ejemplo, los resultados experimentales podrían tener que ser colocados en orden ascendente o descendente para su análisis estadístico, listas de nombres pueden tener que ser clasificadas en orden alfabético, o una lista de fechas puede tener que reordenarse en orden ascendente de fechas. Del mismo modo, puede tener que buscar en una lista de nombres para encontrar un nombre particular en la lista, o puede tener que buscar en una lista de fechas para localizar una fecha particular. En esta sección se introducen los fundamentos del ordenamiento y búsqueda en listas. hay que observar que no es necesario ordenar una lista antes de hacer una búsqueda aunque, como se verá, son posibles búsquedas más rápidas si la lista está ordenada.

Algoritmos de búsqueda

Un requerimiento común de muchos programas es buscar un elemento determinado en una lista. Por ejemplo, en una lista de nombres y números telefónicos, podría buscarse un nombre específico de modo que pueda imprimirse el número telefónico correspondiente, o podría desearse buscar en la lista tan sólo para determinar si un nombre está ahí. Los dos métodos más comunes para llevar a cabo dichas búsquedas son los algoritmos de búsqueda lineal y binaria.

Búsqueda lineal

En una **búsqueda lineal**, la cual también se conoce como **búsqueda secuencial**, cada elemento en la lista es examinado en el orden en el que ocurre en la lista, hasta que se encuentra el elemento deseado o se llega al final de la lista. Esto es análogo a ver cada nombre en el directorio telefónico, comenzando con Aardvark, Aaron, hasta que se encuentra el que se busca o hasta que se llega a Zzxgy, Zora. Es obvio que ésta no es la forma más eficiente de buscar en una lista alfabetizada larga. Sin embargo, una búsqueda lineal tiene estas ventajas:

1. El algoritmo es simple.
2. La lista no necesita estar en ningún orden particular.

En una búsqueda lineal ésta comienza en el primer elemento en la lista y continúa de manera secuencial, elemento por elemento, a lo largo de la lista. El seudocódigo para una función que ejecute una búsqueda lineal es

```
Para todos los elementos en la lista
   Compare el elemento con el elemento deseado
```

```
   Si el elemento se encuentra
      Devolver el valor de índice del elemento actual
   Termina el si
Termina el para
Devolver -1 si el elemento no se encontró
```

Hay que observar que el valor `return` de la función indica si el elemento se encontró o no. Si el valor `return` es `-1`, el elemento no estaba en la lista; de lo contrario, el valor `return` dentro del ciclo `for` proporciona el índice de la ubicación del elemento dentro de la lista.

La función `busquedaLineal()` ilustra este procedimiento como una función de C++:

```
// esta funcion devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaLineal(int lista[], int tamanio, int clave)
{
  int i;

  for (i = 0; i < tamanio; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

Al revisar `busquedaLineal()` observe que el ciclo for tan sólo se utiliza para tener acceso a cada elemento en la lista, desde el primer elemento al último, hasta que se encuentra una coincidencia con el elemento deseado. Si se localiza el elemento deseado se devuelve el valor índice del elemento actual, lo cual causa que termine el ciclo; de lo contrario, la búsqueda continúa hasta que se encuentra el final de la lista.

Para probar esta función se ha escrito una función controladora `main()` para llamar y desplegar los resultados devueltos por `busquedaLineal()`. El programa de prueba completo se ilustra en el programa 11.12.

A continuación se presentan ejecuciones de muestra del programa 11.12:

```
Introduzca el elemento que esta buscando: 101
El elemento fue encontrado en la posición índice 9
```

and

```
Introduzca el elemento que esta buscando: 65
El elemento no se encontró en la lista
```

Como ya se ha señalado, una ventaja de las búsquedas lineales es que la lista no tiene que ordenarse para ejecutar la búsqueda. Otra ventaja es que si el elemento deseado está hacia el principio de la lista, sólo se hará un pequeño número de comparaciones. El peor caso, por supuesto, ocurre cuando el elemento deseado está al final de la lista. En promedio, sin embargo, y suponiendo que el elemento deseado tiene una probabilidad igual de estar en cualquier parte dentro de la lista, el número de comparaciones requeridas será $n/2$, donde n es el tamaño de la lista. Por tanto, para una lista de 10 elementos, el número promedio de comparaciones necesarias para una búsqueda lineal es 5, y para una lista de 10 000 elementos, el número

promedio de comparaciones necesarias es 5 000. Como se muestra a continuación, este número puede reducirse de manera significativa usando un algoritmo de búsqueda binaria.

Programa 11.12

```
#include <iostream>
using namespace std;

int busquedaLineal(int [], int, int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
  int elemento, ubicación;


  cout << "Introduzca el elemento que esta buscando: ";
  cin >> elemento;

  ubicación = busquedaLineal(nums, NUMEL, elemento);

  if (ubicación > -1)
    cout << "El elemento fue encontrado en la posición índice " << ubicacion
         << endl;
  else
    cout << "El elemento no se encontro en la lista\n";

  return 0;
}

// esta función devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaLineal(int lista[], int tamanho, int clave)
{
  int i;

  for (i = 0; i < tamanio; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

Búsqueda binaria

En una **búsqueda binaria** la lista debe estar ordenada. Empezando por una lista ordenada, el elemento deseado se compara primero con los elementos en medio de la lista (para listas con un número par de elementos, puede usarse cualquiera de los dos elementos intermedios). Se presentan tres posibilidades una vez que se hace la comparación: el elemento deseado puede ser igual al elemento intermedio, puede ser mayor o puede ser menor que el elemento intermedio.

En el primer caso la búsqueda ha sido exitosa, y ya no se requiere una mayor búsqueda. En el segundo caso, dado que el elemento deseado es mayor que el elemento intermedio, si se encuentra en absoluto debe estar en la segunda mitad de la lista. Esto significa que la primera parte de la lista, consistente en todos los elementos desde el primero hasta el intermedio, puede descartarse para cualquier búsqueda posterior. En el tercer caso, dado que el elemento deseado es menor que el elemento intermedio, si se encuentra debe hallarse en la primera parte de la lista. Para este caso la segunda mitad de la lista que contiene todos los elementos desde el intermedio hasta el último pueden desecharse para cualquier búsqueda posterior.

El algoritmo para implementar esta estrategia de búsqueda se ilustra en la figura 11.14 y se define en el siguiente seudocódigo:

```
Establecer el índice inferior en 0
Establecer el índice superior en uno menos que el tamaño de la lista
Comenzar con el primer elemento en la lista
Mientras el índice inferior sea menor que o igual al índice superior
   Establecer el índice del punto medio como el promedio en número entero de los valo-
   res de índice inferior y superior
   Comparar el elemento deseado con el elemento intermedio
      Si el elemento deseado es igual al elemento intermedio
         Devolver el valor del índice del elemento actual
    De no ser así si el elemento deseado es mayor que el elemento intermedio
       Establecer el valor del índice inferior en el valor del punto intermedio más 1
    De no ser así si el elemento deseado es menor que el elemento intermedio
       Establecer el valor del índice superior en el valor del punto intermedio menos 1
    Termina el si
Termina el mientras
Devolver -1 si el elemento no se encuentra
```

Como se ilustra tanto con el seudocódigo como con el diagrama de flujo de la figura 11.14, se utiliza un ciclo `while` para controlar la búsqueda. La lista inicial se define estableciendo el valor del índice izquierdo en 0 y el valor del índice derecho en uno menos que el número de elementos en la lista. El elemento intermedio se toma entonces como el promedio, en número entero, de los valores izquierdo y derecho. Una vez que se ha hecho la comparación con el elemento intermedio, la búsqueda se restringe en lo subsiguiente al mover el índice izquierdo a un valor entero por encima del punto medio o al mover el índice derecho a un valor entero por debajo del punto medio. Este proceso continúa hasta que se encuentra el elemento deseado o los valores de índice izquierdo y derecho se vuelven iguales. La función `busquedaBinaria()` presenta la versión en C++ del algoritmo.

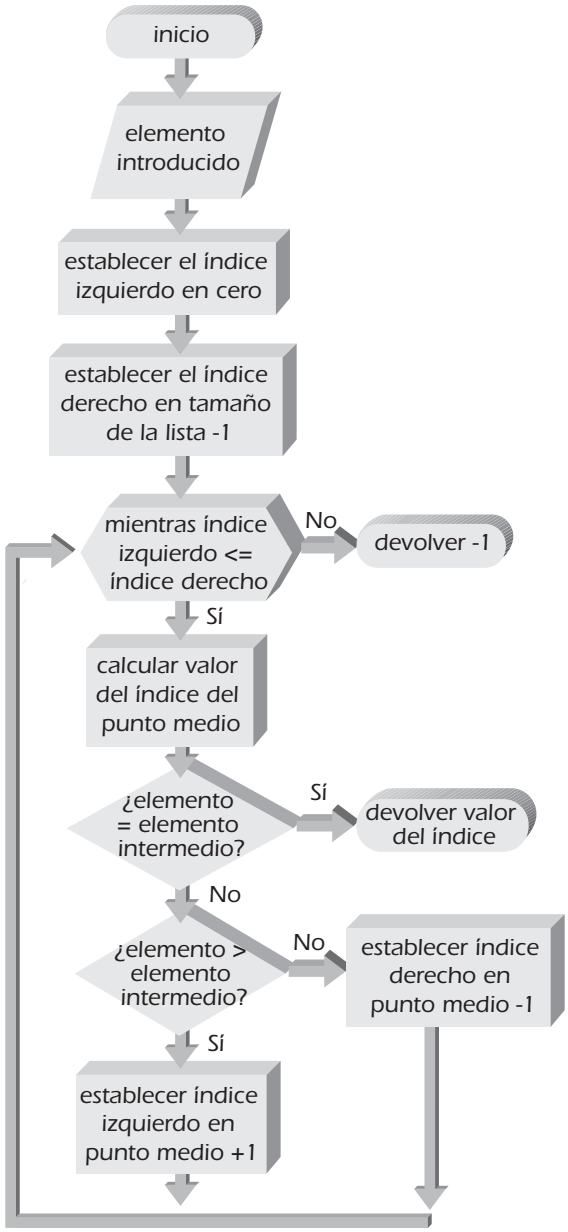


Figura 11.14 El algoritmo de búsqueda binaria.

```
// esta función devuelve la ubicación de clave en la lista
// se devuelve -1 si no se encuentra el valor
int busquedaBinaria(int lista[], int tamaño, int clave)
{
  int izquierdo, derecho, puntomedio;

  izquierdo = 0;
  derecho = tamanho -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio -1;
  }

  return -1;
}
```

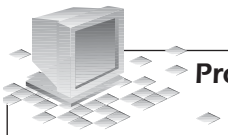
Para probar esta función se usó el programa 11.13.

A continuación se presenta una ejecución de muestra del programa 11.13:

```
Introduzca el elemento que esta buscando: 101
El elemento fue encontrado en la posición índice 9
```

La ventaja de usar un algoritmo de búsqueda binaria es que el número de elementos en el que debe buscarse se reduce a la mitad cada vez por medio del ciclo `while`. Por tanto, la primera vez se recorre el ciclo debe buscarse en n elementos; la segunda vez que se recorre el ciclo se han eliminado $n/2$ de los elementos y sólo quedan $n/2$. La tercera vez que se recorre el ciclo otra mitad de los elementos restantes se ha eliminado, y así en forma sucesiva.

En general, después de p recorridos por el ciclo, el número de valores restantes en los que se va a buscar es $n/(2^p)$. En el peor caso la búsqueda puede continuar hasta que haya menos que o igual a 1 elemento restante en el cual buscar. Matemáticamente, esto puede expresarse como $n/(2^p) \leq 1$. De manera alternativa, esto puede replantearse como que p es el número entero menor tal que $2^p \geq n$. Por ejemplo, para un arreglo de 1000 elementos, n es 1000 y el número máximo de recorridos, p, requerido para una búsqueda binaria es 10. La tabla 11.3 compara el número de recorridos del ciclo necesarios para una búsqueda lineal y binaria para varios tamaños de lista.



Programa 11.13

```
#include <iostream>
using namespace std;

int busquedaBinaria(int [], int, int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
  int elemento, ubicación;
  cout << "Introduzca el elemento que esta buscando: ";
  cin >> elemento;
  ubicacion = busquedaBinaria(nums, NUMEL, elemento);
  if (ubicacion > -1)
    cout << "El elemento fue encontrado en la posición indice "
       << ubicacion << endl;
  else
    cout << "El elemento no se encontró en el arreglo\n";

  return 0;
}

// esta función devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaBinaria(int lista[], int tamanio, int clave)
{
  int izquierdo, derecho, puntomedio

  izquierdo = 0;
  derecho = tamanio -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio -1;
  }

  return -1;
}
```

Tabla 11.3 Una comparación de recorridos del ciclo `while` para búsquedas lineales y binarias

Tamaño del arreglo	10	50	500	5000	50 000	500 000	5 000 000	50 000 000
Recorridos de búsqueda lineal promedio	5	25	250	2500	25 000	250 000	2 500 000	25 000 000
Recorridos de búsqueda lineal máximos	10	50	500	5000	50 000	500 000	5 000 000	50 000 000
Recorridos de búsqueda binaria máximos	4	6	9	13	16	19	23	26

Como se ilustra, el número máximo de recorridos del ciclo para una lista de 50 elementos es casi 10 veces mayor para una búsqueda lineal que para una búsqueda binaria, y es aún más espectacular para listas más grandes. Como regla general, por lo común se toman 50 elementos como el punto de intercambio: para listas más pequeñas que 50 elementos son aceptables las búsquedas lineales; para listas más grandes deberá usarse un algoritmo de búsqueda binaria.

La notación de la O grande

En promedio, en una gran cantidad de búsquedas lineales con n elementos en una lista, se esperaría examinar la mitad ($n/2$) de los elementos antes de localizar el elemento deseado. En una búsqueda binaria el número máximo de recorridos, p, ocurre cuando $n/(2)^p = 1$. Esta relación puede manipularse algebraicamente a $2^p = n$, lo cual produce $p = \log_2 n$, lo cual aproximadamente es igual a $3.33\log_{10} n$.

Por ejemplo, encontrar un nombre particular en un directorio alfabético con $n = 1000$ nombres requeriría un promedio de 500 ($=n/2$) comparaciones usando una búsqueda lineal. Con una búsqueda binaria, sólo se requerirían alrededor de 10 ($\approx 3.33 * \log_{10}1000$) comparaciones.

Una forma común de expresar el número de comparaciones requeridas en cualquier algoritmo de búsqueda usando una lista de n elementos es dar el orden de magnitud del número de comparaciones requeridas, en promedio, para localizar un elemento deseado. Por tanto, se dice que la búsqueda lineal es del orden n y la búsqueda binaria del orden $\log_2 n$. Desde el punto de vista de la notación, esto se expresa como $O(n)$ y $O(\log_2 n)$, donde la O se lee como "del orden de".

Algoritmos de ordenamiento

Para ordenar datos, existen dos categorías principales de técnicas de ordenamiento, llamadas ordenamientos internos y externos, respectivamente. Los **ordenamientos internos** se usan cuando la lista de datos no es demasiado grande y la lista completa puede ordenarse dentro de la memoria de la computadora, por lo general en un arreglo. Los **ordenamientos externos** se usan para conjuntos de datos mucho más grandes que se almacenan en archivos de disco o cinta externos grandes y no pueden acomodarse dentro de la memoria de la computadora como una unidad completa. Aquí se presentan dos algoritmos de ordenamiento interno que pueden utilizarse en forma efectiva cuando se ordenan listas con menos de aproximadamente 50 elementos. Para listas más grandes de manera típica se emplean algoritmos de ordenamiento más complejos.

Ordenamiento por selección

Una de las técnicas de ordenamiento más simples es el **ordenamiento por selección**. En un ordenamiento por selección el valor más pequeño se selecciona al inicio de la lista completa

de datos y se intercambia con el primer elemento en la lista. Después de esta primera selección e intercambio, el siguiente elemento más pequeño en la lista revisada se selecciona e intercambia con el segundo elemento en la lista. Dado que el elemento más pequeño ya está en la primera posición en la lista, este segundo recorrido sólo necesita considerar del segundo al último elementos. Para una lista consistente en n elementos, este proceso se repite $n - 1$ veces, con cada recorrido a lo largo de la lista requiriendo una comparación menos que el recorrido anterior.

Por ejemplo, considérese la lista de números ilustrada en la figura 11.15. El primer recorrido a través de la lista inicial produce que se seleccione el número 32 y se intercambie con el primer elemento en la lista. El segundo recorrido, hecho en la lista reordenada, produce que se seleccione el número 155 del segundo al quinto elementos. Este valor se intercambia entonces con el segundo elemento en la lista. El tercer recorrido selecciona el número 307 del tercero al quinto elementos en la lista e intercambia este valor con el tercer elemento. Por último, el cuarto y último recorrido a través de la lista selecciona el valor mínimo restante y lo intercambia con el cuarto elemento de la lista. Aunque cada recorrido en este ejemplo produjo un intercambio, no se habría hecho ningún intercambio en un recorrido si el valor más pequeño ya estuviera en la ubicación correcta.

lista inicial	recorrido 1	recorrido 2	recorrido 3	recorrido 4
690	32	32	32	32
307	307	155	144	144
32	690	690	307	307
155	155	307	690	426
426	426	426	426	690

Figura 11.15 Una muestra de ordenamiento por selección.

En seudocódigo, el ordenamiento por selección se describe como

```
Establecer contador de intercambio en cero (no se requiere, pero se hace para dar seguimiento
a los intercambios)
Para cada elemento en la lista del primero al penúltimo
  Encontrar el elemento menor a partir del elemento actual referido al último elemento;
    Establecer el valor mínimo igual al elemento actual
    Guardar (almacenar) el índice del elemento actual
    Para cada elemento en la lista desde el elemento actual + 1 hasta el último elemento en la lista
      Si elemento[índice de ciclo interior] < valor mínimo
        Establecer valor mínimo = elemento[índice de ciclo interior]
        Guardar el índice del nuevo valor mínimo encontrado
      Termina el si
    Termina el para
    Intercambiar el valor actual con el nuevo valor mínimo
    Incrementar el contador de intercambio
Termina el para
Devolver la cuenta de intercambio
```

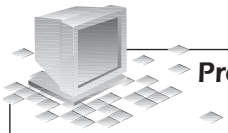
La función `ordenSeleccion()` incorpora este procedimiento en una función de C++.

```
int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];  // supone que el mínimo es el primer elemento
                      del arreglo
    indicemin = i; // índice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)    // si se ha localizado un valor inferior
      {                    // se captura
      min = num[j];
      indicemin = j;
      }
    }
    if (min < num[i])  // comprueba si se tiene un nuevo mínimo
    {                  // y si se tiene, intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```

La función `ordenSeleccion()` espera dos argumentos: la lista que se va a ordenar y el número de elementos en la lista. Como lo especifica el seudocódigo, un conjunto de ciclos `for` anidados ejecuta el ordenamiento. El ciclo `for` exterior causa un recorrido menos a lo largo de la lista que el número total de elementos de datos en la lista. Para cada recorrido, al principio se le asigna a la variable `min` el valor `num[i]`, donde `i` es la variable contadora del ciclo `for` exterior. dado que `i` comienza en 0 y termina en uno menos que `numel`, cada elemento en la lista, excepto el último, es designado en forma sucesiva como el elemento actual.

El ciclo interior recorre los elementos por debajo del elemento actual y se usa para seleccionar el siguiente valor más pequeño. Por tanto, este ciclo comienza en el valor índice `i + 1` y continúa hasta el final de la lista. Cuando se encuentra un nuevo mínimo, se almacenan su valor y posición en la lista en las variables llamadas `min` e `indicemin`, respectivamente. Al completar el ciclo interior se hace un intercambio sólo si se encontró un valor menor que el de la posición actual.

Con el propósito de probar `ordenSeleccion()` se construyó el programa 11.14. Este programa implementa un ordenamiento por selección para la misma lista de 10 números que se usó antes para probar los algoritmos de búsqueda. Para una comparación posterior con los otros algoritmos de ordenamiento que se presentarán, se cuenta y despliega el número de movimientos que realice el programa para ordenar los datos.



Programa 11.14

```
#include <iostream>
using namespace std;

int ordenSeleccion(int [], int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
  int i, movimientos;

  movimientos = ordenSeleccion(nums, NUMEL);

  cout << "La lista ordenada, en orden ascendente, es:\n";
  for (i = 0; i < NUMEL; i++)
    cout << "  " <<nums[i];

  cout << "\nSe hicieron " << movimientos << " movimientos para ordenar
          esta lista\n";

  return 0;
}

int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];    // supone que el mínimo es el primer elemento del arreglo
    indicemin = i;  // índice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)   // si se ha localizado un valor inferior
      {                   // se captura
      min = num[j];
      indicemin = j;
      }
    }
    if (min < num[i])   // comprueba si se tiene un nuevo minimo
    {                   // y si se tiene, intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```

La salida producida por el programa 11.14 es la siguiente:

```
La lista ordenada, en orden ascendente, es:
  5  10  22  32  45  67  73  98  99  101
Se hicieron 8 movimientos para ordenar esta lista
```

Es evidente que el número de movimientos desplegado depende del orden inicial de los valores en la lista. Una ventaja del ordenamiento por selección es que el número máximo de movimientos que deben hacerse es $n - 1$, donde n es el número de elementos en la lista. Además, cada movimiento es un movimiento final que produce un elemento que reside en su ubicación final en la lista ordenada.

Una desventaja del ordenamiento por selección es que siempre se requieren $n(n-1)/2$ comparaciones, sin importar el orden inicial de los datos. Este número de comparaciones se obtiene como sigue: el último recorrido siempre requiere una comparación, el penúltimo recorrido requiere dos comparaciones, etc., hasta el primer recorrido, el cual requiere $n - 1$ comparaciones. Por tanto, el número total de comparaciones es

$$1 + 2 + 3 + \cdots + n - 1 = n(n-1)/2 = n^2/2 - n/2.$$

Para valores grandes de n domina el n^2, y el orden del ordenamiento por selección es $O(n^2)$.

Ordenamiento por intercambio (burbuja)

En un **ordenamiento por intercambio**, elementos adyacentes de la lista son intercambiados entre sí, de tal manera que la lista queda ordenada. Un ejemplo de dicha secuencia de intercambios es proporcionado por el ordenamiento burbuja, donde se comparan valores sucesivos en la lista, comenzando por los primeros dos elementos. Si la lista se va a poner en orden ascendente (de menor a mayor), el valor más pequeño de los dos que se están comparando siempre se coloca antes del valor más grande. Para listas en orden descendente (de mayor a menor), el menor de los dos valores que se están comparando siempre se coloca después del valor mayor.

Por ejemplo, suponiendo que se va a ordenar una lista de valores en orden ascendente, si el primer elemento en la lista es mayor que el segundo, los dos elementos son intercambiados. Luego se comparan el segundo y tercer elementos. Una vez más, si el segundo elemento es mayor que el tercero, estos dos elementos se intercambian. Este proceso continúa hasta que se han comparado e intercambiado, si es necesario, los últimos dos elementos. Si no se hicieron intercambios durante este recorrido inicial a lo largo de los datos, éstos están en el orden correcto y el proceso termina; de lo contrario, se hace un segundo recorrido a lo largo de los datos, comenzando desde el primer elemento y deteniéndose en el penúltimo elemento. La razón para detenerse en el penúltimo elemento en el segundo recorrido es que el primer recorrido siempre produce que el valor más positivo se "hunda" hasta el final de la lista.

Como un ejemplo específico de este proceso, considérese la lista de números ilustrada en la figura 11.16. La primera comparación produce el intercambio de los valores de los primeros dos elementos, 690 y 307. La siguiente comparación, entre los elementos dos y tres en la lista revisada, produce el intercambio de valores entre el segundo y tercer elementos, 690 y 32. Esta comparación y el posible intercambio de valores adyacentes continúa hasta que se han comparado e intercambiado, de ser el caso, los últimos dos elementos. Este proceso completa el primer recorrido a lo largo de los datos y produce que el número mayor se mueva hasta el final de la lista. Conforme el valor mayor se hunde hasta su lugar de reposo al final de la lista, los elementos más pequeños se elevan lentamente, como "burbujas", hasta el principio de la lista. Este efecto de burbuja de los elementos más pequeños es lo que le da el nombre de ordenamiento de "burbuja" a este algoritmo de ordenamiento.

690 ←	307	307	307	307
307 ←	690 ←	32	32	32
32	32 ←	690 ←	155	155
155	155	155 ←	690 ←	426
426	426	426	426 ←	690

Figura 11.16 El primer recorrido de un ordenamiento por intercambio.

Debido a que el primer recorrido a lo largo de la lista asegura que el valor mayor siempre se mueve al final de la lista, el segundo recorrido se detiene en el penúltimo elemento. Este proceso continúa con cada recorrido deteniéndose en un elemento anterior que el recorrido previo, hasta que se han completado $n - 1$ recorridos a lo largo de la lista o no son necesarios intercambios en cualquier recorrido. En ambos casos la lista resultante está ordenada. El seudocódigo que describe este ordenamiento es

```
Establecer contador de intercambio en cero (no se requiere, pero se hace para dar seguimiento a los intercambios)
Para el primer elemento en la lista hasta llegar a uno menor que el último elemento (índice i)
  Para el segundo elemento en la lista hasta el último elemento (índice j)
    Si num[j] < num[j – 1]
    {
      Intercambiar num[j] con num[j – 1]
      Incrementar contador de intercambio
    }
  Termina el para
Termina el para
Devolver cuenta de intercambio
```

Este algoritmo de ordenamiento se codifica en C++ como la función `ordenBurbuja()`, la cual se incluye dentro del programa 11.15 con propósitos de prueba. Este programa prueba `ordenBurbuja()` con la misma lista de 10 números usada en el programa 11.14 para probar `ordenSeleccion()`. Para hacer una comparación con el ordenamiento por selección anterior, también se cuenta y despliega el número de movimientos adyacentes (intercambios) hechos por `ordenBurbuja()`.

Aquí está la salida producida por el programa 11.15.

```
La lista ordenada, en orden ascendente, es:
  5  10  22  32  45  67  73  98  99  101
Se hicieron 18 movimientos para ordenar esta lista
```

Como con el ordenamiento por selección, el número de comparaciones usando un ordenamiento de burbuja es $O(n^2)$ y el número de movimientos requerido depende del orden inicial de los valores en la lista. En el peor caso, cuando los datos están en orden inverso, el ordenamiento por selección se desempeña mejor que el ordenamiento de burbuja. Aquí ambos ordenamientos requieren $n(n - 1)/2$ comparaciones, pero el ordenamiento por selección sólo necesita $n - 1$ movimientos mientras el ordenamiento de burbuja necesita $n(n - 1)/2$ movimientos. Los movimientos adicionales requeridos por el ordenamiento de burbuja resultan de los intercambios intermedios entre elementos adyacentes para "colocar" cada elemento en su posición final. En este aspecto el ordenamiento por selección es superior, porque no son necesarios movimientos intermedios. Para datos aleatorios, como los usados en los programas 11.14 y 11.15, el ordenamiento por selección por lo general se desempeña igual o mejor que el ordenamiento de burbuja.

Programa 11.15

```
#include <iostream>
using namespace std;

int ordenBurbuja(int [], int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
  int i, movimientos;

  movimientos = ordenBurbuja(nums, NUMEL);

  cout << "La lista ordenada, en orden ascendente, es:\n";
  for (i = 0; i < NUMEL; ++i)
    cout << "   " <<nums[i];

  cout << "\nSe hicieron " << movimientos << " movimientos para ordenar
          esta lista\n";

  return 0;
}

int ordenBurbuja(int num[], int numel)
{
  int i, j, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    for(j = 1; j < numel; j++)
    {
      if (num[j] < num[j-1])
      {
        temp = num[j];
        num[j] = num[j-1];
        num[j-1] = temp;
        movimientos++;
      }
    }
  }

  return movimientos;
}
```