CAPÍTULO 12

Apuntadores

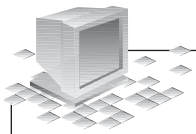
TEMAS

Una de las ventajas de C++ es que permite al programador tener acceso a las direcciones de variables usadas en un programa. Esto permite a los programadores entrar en forma directa en el funcionamiento interno de una computadora y tener acceso a su estructura de almacenamiento básica; lo que da al programador en C++ capacidades y poder de programación que no está disponible en otros lenguajes de nivel alto. Esto se logra usando una característica llamada apuntadores. Aunque otros lenguajes proporcionan apuntadores, C++ extiende esta característica al proveer aritmética de apuntadores; es decir, los valores de apuntadores pueden sumarse, restarse y compararse.

De manera fundamental, los apuntadores son tan sólo variables que se usan para almacenar direcciones de memoria. Este capítulo presenta lo esencial de la declaración de apuntadores, y luego proporciona métodos de aplicación de variables apuntadoras para tener acceso a sus direcciones almacenadas y usarlas en formas significativas.

12.1 DIRECCIONES Y APUNTADORES

Como se vio en la sección 2.5, para desplegar la dirección de una variable se puede usar el operador de dirección de C++, `&`. Para determinar la dirección de `num` puede usarse el operador de dirección de C++, `&`, el cual significa "la dirección de". Cuando se utiliza en una instrucción que no es declarativa, el operador de dirección antepuesto al nombre de una variable se refiere a la dirección de la variable.[1] Por ejemplo, en una instrucción que no es declarativa, `&num` significa *la dirección* de `num`, `millas` significa *la dirección* de `millas` y `&foo` significa la dirección de `foo`. El programa 12.1, el cual es una copia del programa 2.10, usa el operador de dirección para desplegar la dirección de la variable `num`.



Programa 12.1

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

int main()
{
  int num;

  num = 22;
  cout << "num = " << num << endl;
  cout << "La dirección de num = " << &num << endl;

  return 0;
}
```

La salida del programa 12.1 es

```
num = 22
La dirección de num = 0012FED4
```

La figura 12.1 ilustra tanto el contenido como la dirección de la variable `num` proporcionada por la salida del programa 12.1.

[1] Como se vio en el capítulo 6, cuando se usa para declarar argumentos de referencia, el signo & se refiere al tipo de datos que lo *precede*. Por tanto, tanto la declaración `double& num;` como `double &num;` se leen como "`num` es la dirección de un `double`" o, de manera más común, "`num` es una referencia a un `double`".

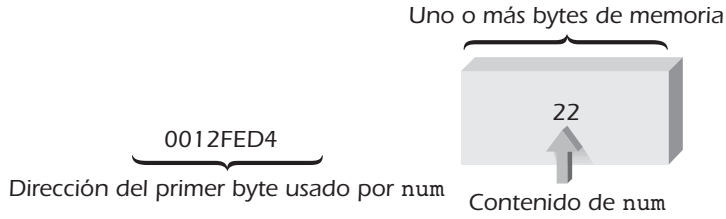


Figura 12.1 *Una representación más completa de la variable* `num`.

Como se mencionó en la sección 2.5, la información de dirección cambiará dependiendo de cuál computadora esté ejecutando el programa y cuántos otros programas estén cargados en la memoria en ese momento.

Almacenamiento de direcciones

Además de desplegar la dirección de una variable, como se hizo en el programa 12.1, también se puede almacenar direcciones en variables declaradas de manera adecuada. Por ejemplo, la instrucción

```
dirNum = &num;
```

almacena la dirección correspondiente a la variable `num` en la variable `dirNum`, como se ilustra en la figura 12.2. Del mismo modo, las instrucciones

```
d = &m;
apunta_tab = &lista;
apunta_car = &car;
```

almacenan las direcciones de las variables `m`, `lista` y `car` en las variables `d`, `apunta_tab` y `apunta_car`, respectivamente, como se ilustra en la figura 12.3.

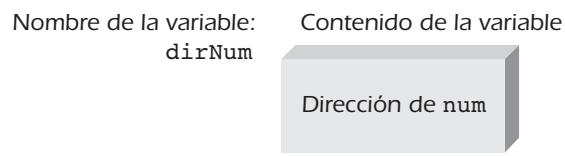


Figura 12.2 *Almacenamiento de la dirección de* `num` *en* `dirNum`.

Las variables `dirNum, d, apunta_tab` y `apunta_car` se llaman de manera formal ***variables apuntadoras***, o ***apuntadores*** para abreviar. Los ***apuntadores*** son tan sólo variables que se usan para almacenar las direcciones de otras variables.

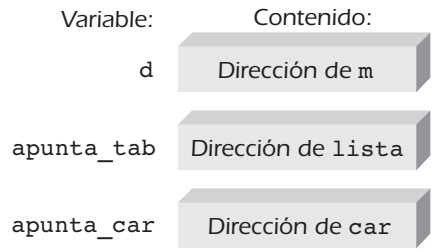


Figura 12.3 *Almacenamiento de más direcciones.*

Uso de direcciones

Para usar una dirección almacenada, C++ proporciona un **operador de indirección**, *. El símbolo *, cuando es seguido por un apuntador (con un espacio permitido tanto antes como después de *), significa *la variable cuya dirección está almacenada en*. Por tanto, si `dirNum` es un apuntador (recuérdese que un apuntador es una variable que almacena una dirección), `*dirNum` significa *la variable cuya dirección está almacenada en* `dirNum`. Del mismo modo, `*apunta_tab` significa *la variable cuya dirección está almacenada en* `apunta_tab` y `*apunta_car` significa *la variable cuya dirección está almacenada en* `apunta_car`. La figura 12.4 muestra la relación entre la dirección contenida en una variable apuntadora y la variable direccionada finalmente.

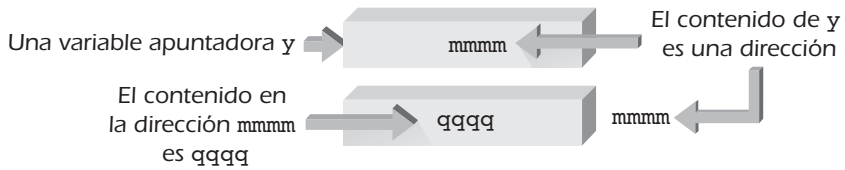


Figura 12.4 *Uso de una variable apuntadora.*

Aunque `*d` significa de manera literal *la variable cuya dirección está almacenada en* `d`, esto por lo común se abrevia como *la variable a la que apunta* `d`. Del mismo modo, refiriéndose a la figura 12.4, `*y` puede leerse como *la variable a la que apunta* `y`. El valor obtenido al final, como se muestra en la figura 12.4, es `qqqq`.

Cuando se usa una variable apuntadora, el valor que se obtiene al final siempre se encuentra yendo primero a la variable apuntadora (o apuntador, para abreviar) por una dirección. La dirección contenida en el apuntador se usa entonces para obtener el contenido deseado. Desde luego, ésta es una forma bastante indirecta de conseguir el valor final y, como era de esperar, se usa el término direccionamiento indirecto para describir este procedimiento.

Debido a que el uso de un apuntador requiere que la computadora haga una búsqueda doble (primero se recupera la dirección, luego se usa la dirección para recuperar los datos reales), una pregunta que vale la pena es, ¿por qué desearía almacenar una dirección en primer lugar? La respuesta a esta pregunta se encuentra en la relación íntima entre apuntadores y arreglos y en la capacidad de los apuntadores para crear y eliminar en forma dinámica ubicaciones de almacenamiento de variables nuevas, mientras un programa se está ejecutando. Estos temas se presentan más adelante en este capítulo. Por ahora, sin

embargo, dado que cada variable tiene una dirección de memoria asociada con ella, la idea de almacenar en verdad una dirección no debería parecer demasiado extraña.

Declaración de apuntadores

Como todas las variables, los apuntadores deben ser declarados antes que puedan utilizarse para almacenar una dirección. Cuando se declara una variable apuntadora, C++ requiere que también se especifique el tipo de variable al que apunta. Por ejemplo, si la dirección en el apuntador `dirNum` es la dirección de un número entero, la declaración correcta para el apuntador es

```
int *dirNum;
```

Esta declaración se lee como *la variable a la que apunta* `dirNum` (de `*dirNum` en la declaración) *es un número entero*.[2]

Hay que observar que la declaración int `*dirNum`; especifica dos cosas: primera, que la variable a la que apunta `dirNum` es un número entero, y segunda, que `dirNum` debe ser un apuntador (debido a que se usa con el operador de indirección *). Del mismo modo, si el apuntador `apunta_tab` apunta a (contiene la dirección de) un número de precisión doble y `apunta_car` apunta a una variable de carácter, las declaraciones requeridas para estos apuntadores son

```
double *tabPoint;
char *chrPoint;
```

Estas dos declaraciones se pueden leer, respectivamente, como la variable a la que apunta `tabPoint` es una doble variable señalada por `chrPoint` es un `char`. Porque aparecen todas las direcciones iguales, esta información adicional es necesitada por el recopilador para saber cuántas localizaciones del almacenaje al acceso cuando utiliza la dirección almacenada en el indicador. Otros ejemplos de las declaraciones del indicador son

```
char *enClave;
int *pt_num;
double *Imprnum1;
```

Para entender las declaraciones de apuntadores, es útil leerlas "a la inversa", empezando con el operador de indirección, el asterisco, `*`, y traduciéndolo como *la variable cuya dirección está almacenada en* o *como la variable a la que apunta*. Al aplicar esto a declaraciones de apuntadores, la declaración `char *enClave`, por ejemplo, puede leerse como la variable cuya dirección está almacenada en `enClave` es un carácter o como la variable a la que apunta `enClave` *es un carácter*. Ambos enunciados se abrevian con frecuencia en el enunciado más simple `enClave` *apunta* a *un carácter*. Debido a que las tres interpretaciones de la instrucción de declaración son correctas, puede seleccionarse y usarse cualquier descripción que tenga más sentido para usted. Ahora uniremos todo esto para construir un programa usando apuntadores. Considérese el programa 12.2.

---

[2]Las declaraciones de apuntadores también pueden escribirse en la forma *tipoDatos* nombreApuntador*; donde se coloca un espacio entre el símbolo del operador de indirección y el nombre de la variable apuntadora. Sin embargo, esta forma es propensa a error cuando se declaran múltiples variables apuntadoras en la misma instrucción de declaración y el símbolo de asterisco se omite de manera inadvertida después que se declara el nombre del primer apuntador. Por ejemplo, la declaración `int* num1, num2;` declara `num1` como una variable apuntadora y `num2` como una variable en número entero. Con el fin de acomodar con más facilidad múltiples apuntadores en la misma declaración y marcar con claridad una variable como apuntador, nos apegaremos a la convención que coloca un asterisco de manera directa enfrente del nombre de cada variable apuntadora. Este posible error ocurre rara vez con declaraciones de referencia debido a que las referencias se usan casi exclusivamente como parámetros formales y es obligatoria la declaración única de parámetros.

Programa 12.2

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

int main()
{
  int *dirNum;         // declara un apuntador a un int
  int millas, dist;    // declara dos variables enteras

  dist = 158;          // almacena el número 158 en dist
  millas = 22;         // almacena el número 22 en millas
  dirNum = &millas;    // almacena la 'direccion de millas' en numDir

  cout << "La dirección almacenada en dirNum es " << dirNum << endl;
  cout << "El valor al que apunta dirNum es " << *dirNum << "\n\n";

  dirNum = &dist;   // ahora almacena la dirección de dist en dirNum
  cout << "La dirección almacenada ahora en dirNum es " << dirNum << endl;
  cout << "El valor al que apunta ahora dirNum es " << *dirNum << endl;

  return 0;
}
```

La salida del programa 12.2 es:

```
La dirección almacenada en dirNum es 0012FEC8
El valor al que apunta dirNum es 22

La dirección almacenada ahora en dirNum es 0012FEBC
El valor al que apunta ahora dirNum es 158
```

El único uso del programa 12.12 es ayudar a entender "qué está almacenado y dónde". Se revisará el programa para ver cómo se produjo la salida.

La instrucción de declaración int `*dirNum`; declara que `dirNum` es una variable apuntadora usada para almacenar la dirección de una variable en número entero. La instrucción `dirNum = &millas;` almacena la dirección de la variable millas en el apuntador `dirNum`. La primera activación de `cout` causa que se despliegue esta dirección. La segunda activación de `cout` en el programa 12.2 usa el operador de indirección para recuperar e imprimir *el valor al que apunta* `dirNUm`, el cual es, por supuesto, el valor almacenado en millas.

Debido a que `dirNum` se ha declarado como un apuntador a una variable en número entero, se puede usar este apuntador para almacenar la dirección de cualquier variable en número entero. La instrucción `dirNum = &dist` ilustra esto al almacenar la dirección de la variable `dist` en `dirNum`. Las últimas dos instrucciones `cout` verifican el cambio en el valor de `dirNum` y que la nueva dirección almacenada apunta a la variable `dist`. Como se ilustra en el programa 12.2, sólo deberán almacenarse direcciones en los apuntadores.

Por supuesto que habría sido mucho más simple si el apuntador usado en el programa 12.2 se hubiera podido declarar como `pointer dirNum;`. Dicha declaración, sin embargo, no transmite información del almacenamiento usado por la variable cuya dirección está almacenada en `dirNum`. Esta información es esencial cuando se usa el apuntador con el operador de indirección, como en la segunda instrucción `cout` en el programa 12.2. Por ejemplo, si la dirección de un número entero es almacenada en `dirNum`, entonces por lo general sólo se recuperan cuatro bytes de almacenamiento cuando se usa la dirección. Si la dirección de un carácter se almacena en `dirNum`, sólo se recuperaría un byte de almacenamiento, y un `double` por lo general requiere la recuperación de ocho bytes de almacenamiento. La declaración de un apuntador debe incluir, por consiguiente, el tipo de variable a la que se apunta. La figura 12.5 ilustra este concepto.

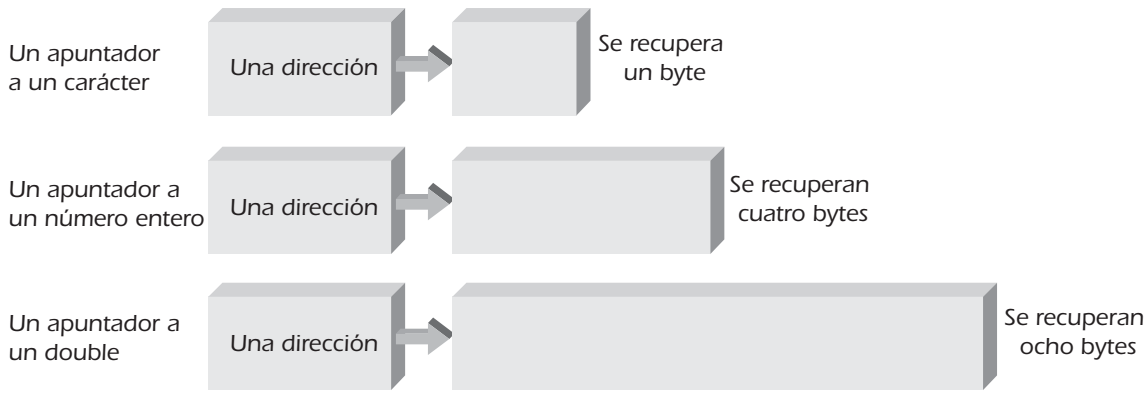


Figura 12.5 *Direccionamiento de diferentes tipos de datos usando apuntadores.*

Referencias y apuntadores

En este punto podría preguntarse cuál es la diferencia entre un apuntador y una referencia. En esencia, una referencia es una constante nombrada para una dirección; como tal, la dirección nombrada como una referencia no puede ser alterada. Dado que un apuntador es una variable, la dirección en el apuntador puede cambiarse. Para la mayor parte de las aplicaciones el uso de referencias sobre los apuntadores como argumentos para funciones es más fácil y se prefiere con claridad. Esto se debe a la notación más simple usada para localizar un parámetro de referencia, la cual elimina el uso del operador de dirección, `&`, y el operador para desreferenciar, `*`, requerido por los apuntadores. Desde el punto de vista técnico, se dice que las referencias se ***desreferencian automáticamente*** o se ***desreferencian implícitamente*** (los dos términos se usan como sinónimos), mientras los apuntadores deben desreferenciarse de manera explícita para localizar el valor al que tienen acceso.

Por ejemplo, al transmitir la dirección de una variable escalar como un argumento de una función, las referencias proporcionan una interfaz de notación más simple y por lo general se prefieren. Los apuntadores se requieren para otras situaciones, como asignar de manera dinámica secciones de memoria nuevas para variables adicionales mientras se ejecuta un programa o cuando se usan alternativas a la notación de arreglos (ambos temas se presentan en este capítulo).

Variables de referencia[3]

Las referencias se usan de manera casi exclusiva como parámetros formales de función y como tipos a devolver. No obstante, las variables de referencia también están disponibles en C++. Para completar el tema, ahora se muestra cómo pueden declararse y usarse estas variables.

Una vez que se ha declarado una variable se le pueden asignar nombres adicionales. Esto se logra usando una declaración de referencia, la cual tiene la forma

tipoDatos& nuevoNombre = nombreExistente;

Por ejemplo, la declaración de referencia

```
double& suma = total;
```

iguala el nombre `suma` al nombre `total`, ambos se refieren ahora a la misma variable, como se ilustra en la figura 12.6.

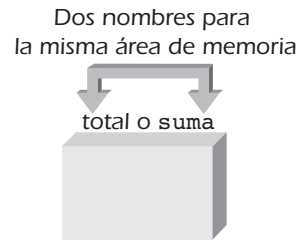


Figura 12.6 `suma` es un nombre alternativo para `total`.

Una vez que se ha establecido otro nombre para una variable usando una declaración de referencia, el nuevo nombre, el cual se conoce como un alias, puede usarse en lugar del nombre original. Por ejemplo, considérese el programa 12.3.

La siguiente salida es producida por el programa 12.3:

```
suma = 20.5
total = 18.6
```

Dado que la variable `suma` tan sólo es otra referencia a la variable `total`, es el valor almacenado en total el que es obtenido por el primer objeto `cout` en el programa 12.3. Cambiar el valor en `suma` cambia entonces el valor en `total`, el cual es desplegado por el segundo objeto `cout` en el programa 12.3.

[3]Esta sección puede omitirse sin perder la continuidad temática.

Programa 12.3

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

int main()
{
  double total = 20.5     // declara e inicializa total
  double& suma = total;   // declara otro nombre para total

  cout << "suma = " << suma << endl;
  suma = 18.6;            // esto cambia el valor en total
  cout << "total = " << total << endl;

  return 0;
}
```

Al construir referencias, deben tenerse en cuenta dos consideraciones. Primera, la referencia deberá ser del mismo tipo de datos que la variable a la que se refiere. Por ejemplo, la secuencia de declaraciones

```
int num = 5;
double& numref = num;    // INVALIDO, CAUSA UN ERROR DE
                            COMPILADOR
```

no iguala `numref` con `num`; más bien, causa un error de compilador porque las dos variables son de tipos de datos diferentes. En segundo lugar, se produce un error de compilador cuando se hace un intento para igualar una referencia a una constante. Por ejemplo, la siguiente declaración es inválida:

```
int& val = 5; // INVALIDO, CAUSA UN ERROR DE COMPILADOR
```

Una vez que se ha igualado en forma correcta un nombre de referencia con un nombre de variable, la referencia no puede cambiarse para referirse a otra variable.

Como con todas las instrucciones de declaración, pueden declararse múltiples referencias en una sola instrucción en tanto cada nombre de referencia esté precedido por el símbolo &. Por tanto, la declaración

```
double& suma = total, & promedio;
```

crea dos variables de referencia llamadas suma y promedio.[4]

[4]Las declaraciones de referencia también pueden escribirse en la forma *tipoDatos &nombreNuevo = nombreExistente*; donde se coloca un espacio entre el símbolo & y el tipo de datos. Sin embargo, esta forma no se usa mucho, tal vez para distinguir la notación de dirección de una variable de referencia de la usada al asignar direcciones a las variables apuntadoras.

Otra forma de considerar las referencias es tomarlas como apuntadores con capacidades restringidas que ocultan de manera implícita una gran cantidad de desreferenciación que se requiere de manera explícita con los apuntadores.

Por ejemplo, considérense las instrucciones

```
int b;       // b es una variable en número entero
int& a = b; // a es una variable de referencia que almacena la
               direccion de b
a = 10;      // esto cambia el valor de b a 10
```

Aquí, `a` es declarada como una variable de referencia que efectivamente es una constante nombrada para la dirección de la variable `b`. Dado que el compilador sabe a partir de la declaración que `a` es una variable de referencia, de manera automática asigna la dirección de `b` (en lugar del contenido de `b`) a `a` en la instrucción de declaración. Por último, en la instrucción `a = 10;` el compilador usa la dirección almacenada en `a` para cambiar el valor almacenado en `b` a 10. La ventaja de usar la referencia es que ejecuta de manera automática un acceso indirecto al valor de `b` sin la necesidad de usar de forma explícita el símbolo de indirección, `*`. Como se ha señalado antes, este tipo de acceso se conoce como ***desreferencia automática***.

Para implementar esta misma correspondencia entre `a` y `b` usando apuntadores se hace con la siguiente secuencia de instrucciones:

```
int b;       // b es una variable en número entero
int *a = &b; // a es un apuntador, almacena la direccion
                de b en a
*a = 10;     // esto cambia el valor de b a 10 al
                desreferenciarlo
             // de manera explícita de la direccion en a
```

Aquí `a` se define como un apuntador que se ha inicializado para almacenar la dirección de `b`. Por tanto, `*a`, lo cual puede leerse como "la variable cuya dirección está en `a`" o "la variable a la que apunta `a`" es `b`, y la expresión `*a = 10` cambia el valor de `b` a `10`. Hay que observar en el caso del apuntador que la dirección almacenada puede alterarse para apuntar a otra variable; en el caso de la referencia la variable de referencia no puede alterarse para referirse a cualquier variable excepto aquella con la que se inicializó. También hay que observar que para desreferenciar `a` se debe usar de manera explícita el operador de indirección, `*`. Como podría esperarse, el símbolo `*` también se conoce como operador de desreferencia.

Ejercicios 12.1

1. Si `promedio` es una variable, ¿qué significa `&promedio`?
2. Para las variables y direcciones ilustradas en la figura 12.7, determine `&temp`, `&dist`, `&fecha` y `&millas`.

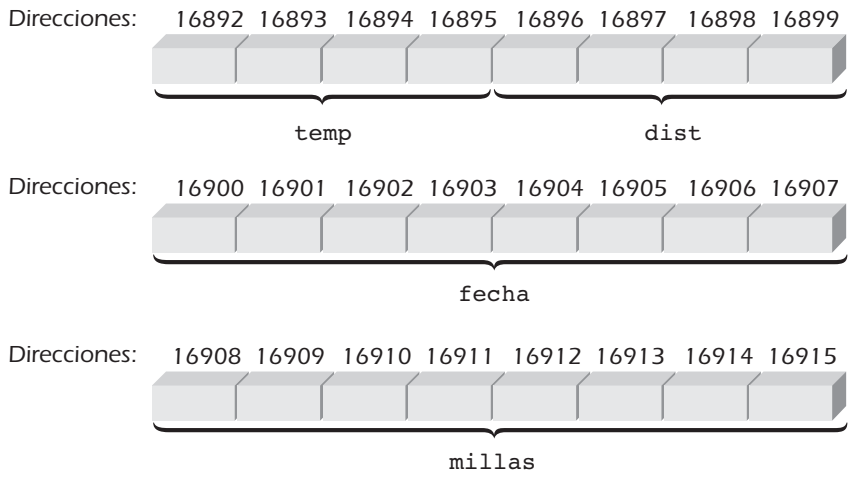


Figura 12.7 *Bytes de memoria para el ejercicio 2.*

3. a. Escriba un programa en C++ que incluya las siguientes instrucciones de declaración. Haga que el programa use el operador de dirección y el objeto `cout` para desplegar las direcciones correspondientes a cada variable.

```
int num, cuenta;
long fecha;
float pendiente;
double potencia;
```

b. Después de ejecutar el programa escrito para el ejercicio 3a, dibuje un diagrama de la manera en que su computadora ha apartado almacenamiento para las variables en el programa. En su diagrama, llene las direcciones desplegadas por el programa.

c. Modifique el programa escrito en el ejercicio 3a para desplegar la cantidad de almacenamiento que reserva su computadora para cada tipo de datos (use el operador `sizeof()`). Con esta información y la información de direcciones proporcionada en el ejercicio 3b, determine si su computadora apartó almacenamiento para las variables en el orden en que fueron declaradas.

4. Si una variable es declarada como un apuntador, ¿qué debe almacenarse en la variable?

5. Usando el operador de indirección, escriba expresiones para lo siguiente:

a. La variable a la que apunta `dir_x`
b. La variable cuya dirección está en `dir_y`
c. La variable a la que apunta `pt_prod`
d. La variable a la que apunta `pt_millas`
e. La variable a la que apunta `impr_m`
f. La variable cuya dirección está en `fecha_p`
g. La variable a la que apunta `impr_dist`

h. La variable a la que apunta `pt_tab`
i. La variable cuya dirección está en `pt_horas`

6. Escriba instrucciones de declaración para lo siguiente:
a. La variable a la que apunta `dir_y` es un número entero.
b. La variable a la que apunta `dir_car` es un carácter.
c. La variable a la que apunta `pt_anio` es un entero largo.
d. La variable a la que apunta `cant` es una variable de precisión doble.
e. La variable a la que apunta `z` es un número entero.
f. La variable a la que apunta `qp` es una variable de precisión simple.
g. `pt_fecha` es un apuntador a un número entero.
h. `dir_prod` es un apuntador a una variable de precisión doble.
i. `pt_cant` es un apuntador a una variable de precisión simple.
j. `pt_car` es un apuntador a un carácter.

7. **a.** ¿Cómo se denominan las variables `dir_y, dir_car, pt_anio, cant, z, qp, pt_fecha, dir_prod, pt_cant` y `pt_car` usadas en el ejercicio 6?
b. ¿Por qué los nombres de las variables `cant`, `z` y `qp` usados en el ejercicio 6 no son buenas elecciones para nombres de variables apuntadoras?

8. Escriba enunciados en español que describan lo que contienen las siguientes variables declaradas:
a. `char *dir_clave;`
b. `int *m;`
c. `double *dir_prod;`
d. `long *impr_y;`
e. `double *pCou;`
f. `int *pt_fecha;`

9. ¿Cuáles de las siguientes son declaraciones para apuntadores?
a. `long a;`
b. `char b;`
c. `char *c;`
d. `int x;`
e. `int *p;`
f. `double w;`
g. `float *k;`
h. `float l;`
i. `double *z;`

10. Para las siguientes declaraciones,

```
int *pt_x, *dir_y;
long *dir_dt, *dir_pt;
double *pt_Z;
int a;
long b;
double c;
```

determine cuál de las siguientes instrucciones es válida:

```
a. dir_y = &a;     b. dir_y = &b;       c. dir_y = &c;
d. dir_y = a;      e. dir_y = b;        f. dir_y = c;
g. dir_dt = &a;    h. dir_dt = &b;      i. dir_dt = &c;
j. dir_dt = a;     k. dir_dt = b;       l. dir_dt = c;
m.pt_z = &a;       n. dir_pt = &b;      o. dir_pt = &c;
p. dir_pt = a;     q. dir_pt = b;       r. dir_pt = c;
s. dir_y = pt_x;   t. dir_y = dir_dt;   u. dir_y = dir_pt;
```

11. Para las variables y direcciones ilustradas en la figura 12.8, escriba los datos apropiados determinados por las siguientes instrucciones:

```
a. pt_num = &m;
b. dir_cant = &cant;
c. *dir_z = 25;
d. k = *dir_num;
e. pt_dia = dir_z;
f. *pt_anio = 1987;
g. *dir_cant = *dir_num;
```

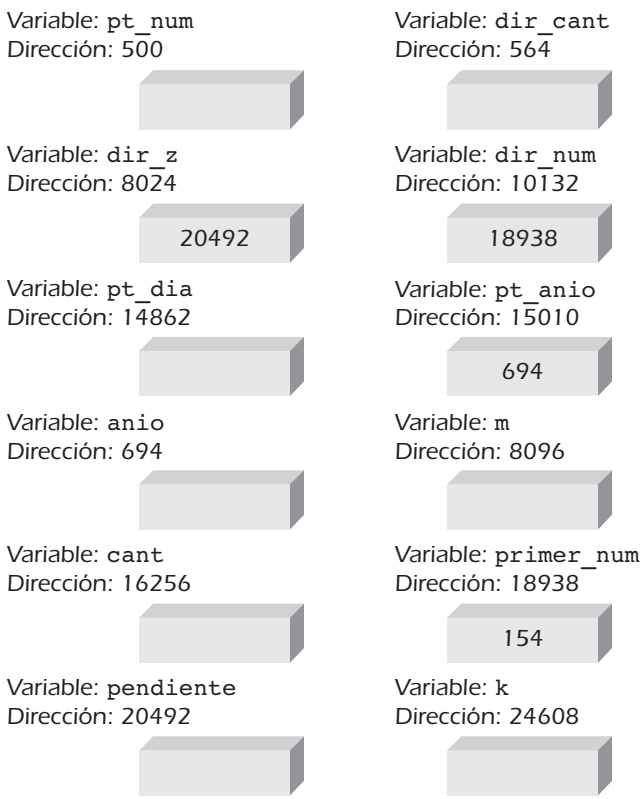


Figura 12.8 Ubicaciones de memoria para el ejercicio 11.

12. Usando el operador `sizeof()`, determine el número de bytes usados por su computadora para almacenar la dirección de un número entero, un carácter y un número de precisión doble. (*Sugerencia:* puede utilizar `sizeof(*int)` para determinar el número de bytes de memoria usados por un apuntador a un número entero.) ¿Esperaría que fuera igual el tamaño de todas las direcciones? ¿Por qué sí o por qué no?

12.2 NOMBRES DE ARREGLOS COMO APUNTADORES

Aunque los apuntadores son, por definición, tan sólo variables usadas para almacenar direcciones, también existe una relación directa y estrecha entre los nombres de los arreglos y los apuntadores. En esta sección se describe esta relación con detalle.

La figura 12.9 ilustra el almacenamiento de un arreglo unidimensional llamado `calif`, el cual contiene cinco números enteros. Suponga que cada número entero requiere dos bytes de almacenamiento.

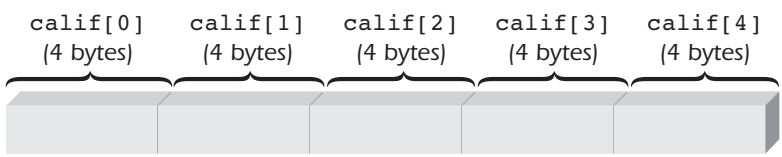


Figura 12.9 *El arreglo* `calif` *en almacenamiento.*

Usando subíndices, se hace referencia al cuarto elemento en el arreglo calif como `calif[3]`. El uso de un subíndice, sin embargo, oculta el uso extenso de direcciones por la computadora. Interiormente, la computadora utiliza de inmediato el subíndice para calcular la dirección del elemento deseado basándose tanto en la dirección inicial del arreglo como en la cantidad de almacenamiento usado por cada elemento. Llamar al cuarto elemento `calif[3]` obliga al compilador, en forma interna, a hacer el cálculo de dirección

```
&calif[3] = &calif[0] + (3 * sizeof(int))
```

Si se recuerda que el operador de dirección, `&`, significa "la dirección de", esta última instrucción se lee "la dirección de `calif[3]` es igual a la dirección de `calif[0]` más 12". La figura 12.10 ilustra el cálculo de dirección usado para localizar `calif[3]`.

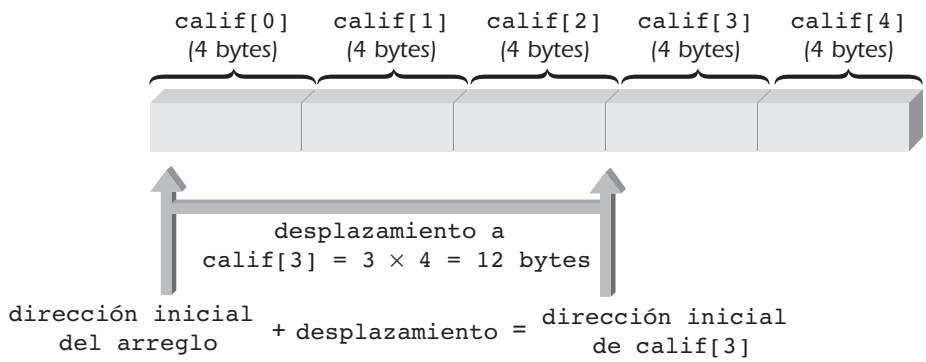
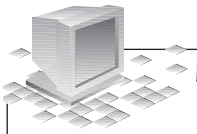


Figura 12.10 *Uso de un subíndice para obtener una dirección.*

Hay que recordar que un apuntador es una variable usada para almacenar una dirección. Si se crea un apuntador para almacenar la dirección del primer elemento en el arreglo `calif`, se puede imitar la operación utilizada por la computadora para tener acceso a los elementos del arreglo. Antes de hacer esto, se considerará primero el programa 12.4.

Cuando se ejecuta el programa 12.4, se obtiene el siguiente despliegue:

```
El elemento 0 es 98
El elemento 1 es 87
El elemento 2 es 92
El elemento 3 es 79
El elemento 4 es 85
```



Programa 12.4

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

int main()
{
  const int TAMANIOARREGLO = 5;

  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << calif[i];
  cout << endl;

  return 0;
}
```

El programa 12.4 despliega los valores del arreglo `calif` usando notación de subíndice estándar. Ahora, se almacenará la dirección del elemento 0 del arreglo en un apuntador. Entonces, usando el operador de indirección, `*`, se puede usar la dirección en el apuntador para tener acceso a cada elemento del arreglo. Por ejemplo, si se almacena la dirección de `calif[0]` en un apuntador llamado `g_Ptr` (usando la instrucción de asignación `g_Ptr = &calif[0];`), entonces, como se ilustra en la figura 12.11, la expresión `*g_Ptr`, la cual significa "la variable a la que apunta `g_Ptr`", hace referencia a `calif[0]`.

Una característica única de los apuntadores es que pueden incluirse desplazamientos en las expresiones que usan apuntadores. Por ejemplo, el 1 en la expresión `*(g_Ptr + 1)` es un **desplazamiento**. La expresión completa hace referencia al número entero que está un lugar adelante de la variable a la que apunta `g_Ptr`. Del mismo modo, como se ilustra en la figura 12.12, la expresión `*(g_Ptr + 3)` hace referencia a la variable que está tres números enteros adelante de la variable a la que apunta `g_Ptr`. Ésta es la variable `calif[3]`.

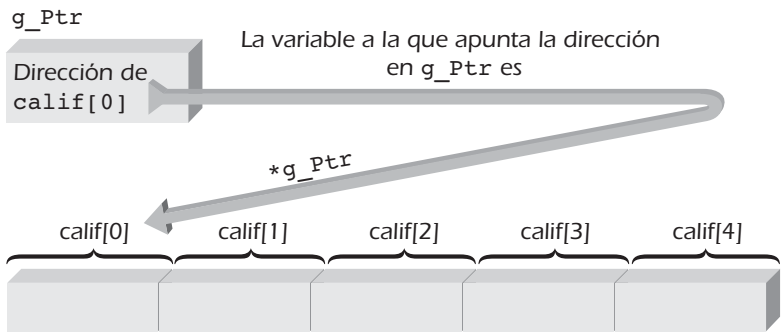


Figura 12.11 *La variable a la que apunta* `*g_Ptr` *es* `calif[0]`.

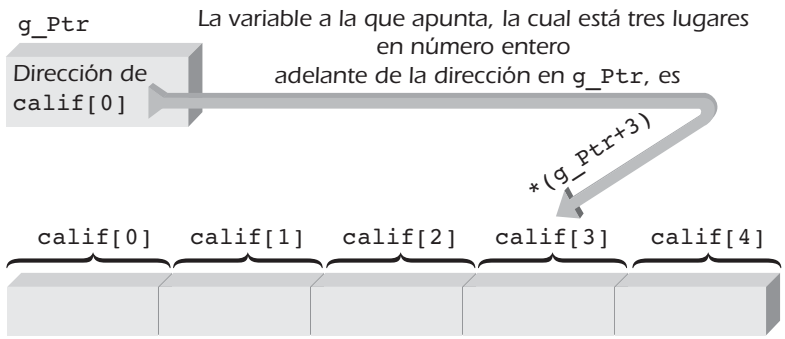


Figura 12.12 *Un desplazamiento de tres desde la dirección en* `g_Ptr`.

La tabla 12.1 muestra la correspondencia completa entre elementos referenciados por subíndices y por apuntadores y desplazamientos. Las relaciones enumeradas en la tabla 12.1 se ilustran en la figura 12.13.

Tabla 12.1 Los elementos del arreglo se pueden referenciar de dos maneras

Elemento del arreglo	Notación de subíndice	Notación de apuntador
Elemento 0	`calif[0]`	`*g_Ptr y (g_Ptr + 0)`
Elemento 1	`calif[1]`	`*(g_Ptr + 1)`
Elemento 2	`calif[2]`	`*(g_Ptr + 2)`
Elemento 3	`calif[3]`	`*(g_Ptr + 3)`
Elemento 4	`calif[4]`	`*(g_Ptr + 4)`

Usando la correspondencia entre apuntadores y subíndices ilustrada en la figura 12.13, ahora pueden utilizarse apuntadores para tener acceso a los elementos del arreglo a los que se tuvo acceso antes en el programa 12.4 usando subíndices. Esto se hace en el programa 12.5.

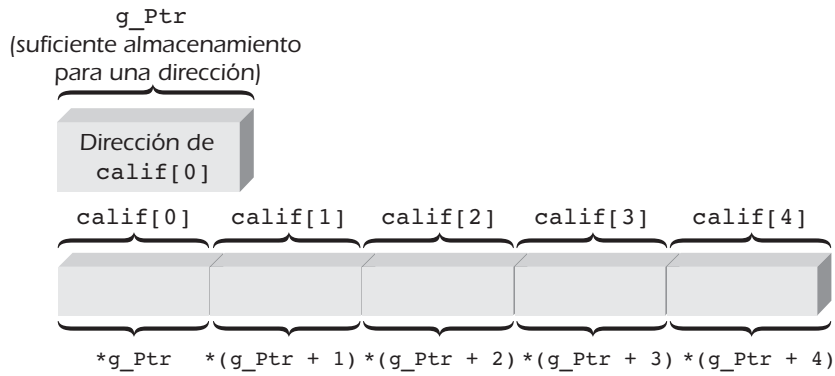


Figura 12.13 *La relación entre elementos del arreglo y apuntadores.*

Programa 12.5

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

int main()
{

  const int TAMANIOARREGLO = 5;

  int *g_Ptr;                // declara un apuntador a un int
  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  g_Ptr = &cali[0];         // almacena la dirección inicial del arreglo
  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << *(g_Ptr + i);
  cout << endl;

  return 0;
}
```

Cuando se ejecuta el programa 12.5 se obtiene el siguiente despliegue:

```
El elemento 0 es 98
El elemento 1 es 87
El elemento 2 es 92
El elemento 3 es 79
El elemento 4 es 85
```

Hay que observar que es el mismo despliegue producido por el programa 12.4.

El método usado en el programa 12.5 para tener acceso a elementos individuales del arreglo simula la forma en que el compilador hace referencia de manera interna a todos los elementos del arreglo. Cualquier subíndice usado por un programador es convertido de manera automática a una expresión de apuntador equivalente por el compilador. En este caso, dado que la declaración de `g_Ptr` incluía la información de que se apunta a números enteros, cualquier desplazamiento añadido a la dirección en `g_Ptr` se escala de manera automática al tamaño de un número entero. Por tanto, `*(g_Ptr + 3)`, por ejemplo, se refiere a la dirección de `calif[0]` más un desplazamiento de doce bytes `(3 * 4)`, donde se ha hecho la suposición que `sizeof(int) = 4`. Ésta es la dirección de `calif[3]` ilustrada en la figura 12.13.

Los paréntesis en la expresión `*(g_Ptr + 3)` son necesarios para hacer referencia en forma correcta al elemento del arreglo deseado. Omitir los paréntesis produce la expresión `*g_Ptr + 3`. Debido a la precedencia de los operadores, esta expresión agrega 3 a "la variable a la que apunta `g_Ptr`". Dado que `g_Ptr` apunta a `calif[0]`, esta expresión suma el valor de `calif[0]` y 3. Hay que observar también que la expresión `*(g_Ptr + 3)` no cambia la dirección almacenada en `g_Ptr`. Una vez que la computadora usa el desplazamiento para localizar la variable correcta a partir de la dirección inicial en `g_Ptr`, el desplazamiento es desechado y la dirección en `g_Ptr` permanece sin cambios.

Aunque el apuntador `g_Ptr` usado en el programa 12.4 se creó de manera específica para almacenar la dirección inicial del arreglo `calif`, esto era, de hecho, innecesario. Cuando se crea un arreglo, el compilador crea de manera automática una constante apuntadora interna para éste y almacena la dirección inicial del arreglo en este apuntador. En casi todos los aspectos, una constante apuntadora es idéntica a una variable apuntadora creada por un programador; pero, como se verá, hay algunas diferencias.

Para cada arreglo creado, el nombre del arreglo se convierte en el nombre de la constante apuntadora creada por el compilador para el arreglo, y la dirección inicial de la primera ubicación reservada para el arreglo es almacenada en este apuntador. Por tanto, al declarar el arreglo `calif` en los programas 12.4 y 12.5 en realidad se reservó suficiente almacenamiento para cinco números enteros, se creó un apuntador interno llamado `calif` y almacenó la dirección de `calif[0]` en el apuntador. Esto se ilustra en la figura 12.14.

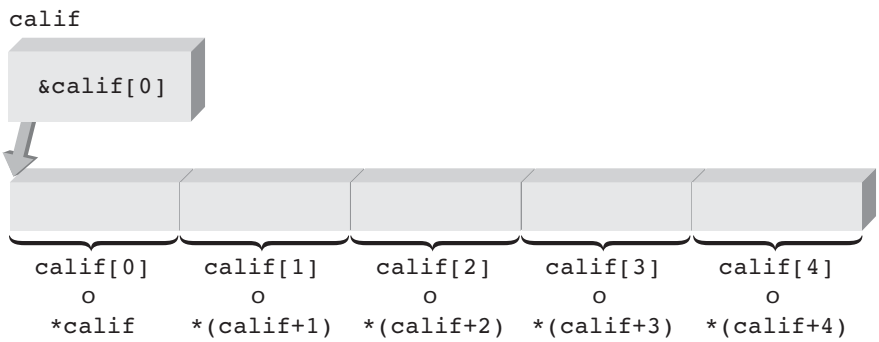
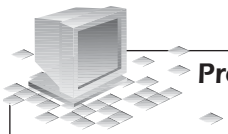


Figura 12.14 *Crear un arreglo también crea un apuntador.*

La implicación es que toda referencia a `calif` utilizando un subíndice puede reemplazarse por una referencia equivalente usando `calif` como apuntador. Por tanto, siempre que se use la expresión `calif[i]`, también puede emplearse la expresión `*(calif + i)`. Esto se ilustra en el programa 12.6, donde se usa `calif` como un apuntador para hacer referencia a todos sus elementos.

La ejecución del programa 12.6 produce la misma salida realizada antes por el programa 12.4 y el programa 12.5. Sin embargo, usar `calif` como apuntador hace innecesario declarar e inicializar el apuntador `g_Ptr` utilizando en el programa 12.5.

En la mayor parte de los aspectos un nombre de arreglo y un apuntador pueden usarse de manera indistinta. *Sin embargo, un apuntador verdadero es una variable y la dirección almacenada en ella puede cambiarse. Un nombre de arreglo es una constante apuntadora y la dirección almacenada en el apuntador no puede ser cambiada por una instrucción de asignación*. Por tanto, una instrucción como `calif = &calif[2];` es inválida. Esto no debería ser una sorpresa. En vista que el propósito de un nombre de arreglo es localizar de manera correcta el principio del arreglo, permitir a un programador cambiar la dirección almacenada en el nombre del arreglo iría en contra de este propósito y conduciría a estragos siempre que se hiciera referencia a sus elementos. Además, las expresiones que toman la dirección de un nombre de arreglo son inválidas porque el apuntador creado por el compilador es interno en la computadora, y no está almacenado en la memoria, como las variables apuntadoras. Por tanto, tratar de almacenar la dirección de `calif` usando la expresión `&calif` produce un error de compilador.



Programa 12.6

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

int main()
{
  const int TAMNIOARREGLO = 5;

  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << *(calif + i);
  cout << endl;

  return 0;
}
```

Un aspecto secundario interesante de la observación de que puede hacerse referencia a los elementos de un arreglo usando apuntadores es que una referencia con apuntador siempre puede reemplazarse con una referencia con subíndice. Por ejemplo, si se declara `num_Ptr` como una variable apuntadora, la expresión `*(num_Ptr + i)` también puede escribirse como `num_Ptr[i]`. Esto es cierto aun cuando `num_Ptr` no es creado como un arreglo. Como antes, cuando el compilador encuentra la notación de subíndice, la reemplaza en forma interna con la notación de apuntador.

Asignación dinámica de arreglos[5]

Conforme se define cada variable en un programa se le asigna suficiente almacenamiento de una reserva de ubicaciones de memoria en la computadora que se ponen a disposición del compilador. Una vez que se han reservado ubicaciones de memoria específicas para una variable, estas ubicaciones se fijan para toda la vida de esa variable, se usen o no. Por ejemplo, si una función solicita almacenamiento para un arreglo de 500 números enteros, se asigna y se fija el almacenamiento para el arreglo desde el punto de la definición del arreglo. Si la aplicación requiere menos de 500 números enteros, el almacenamiento asignado no utilizado no es liberado de nuevo en el sistema hasta que el arreglo deja de existir. Por otra parte, si la aplicación requiere más de 500 números enteros, el tamaño del arreglo de números enteros debe incrementarse y recompilarse la función que define al arreglo.

Una alternativa para esta asignación fija o estática de ubicaciones de almacenamiento en memoria es la asignación dinámica de memoria. Bajo un esquema de ***asignación dinámica***, la cantidad de almacenamiento que se va a asignar es determinada y ajustada conforme se ejecuta el programa, en lugar de fijarse en tiempo de compilación.

La asignación dinámica de memoria es útil en extremo cuando se manejan listas, debido a que permite que se expanda la lista conforme se agregan nuevos elementos y se contraiga conforme se eliminan. Por ejemplo, al construir una lista de calificaciones puede no conocerse el número exacto de calificaciones que se van a necesitar a final de cuentas. En lugar de crear un arreglo fijo para almacenar las calificaciones, es muy útil tener un mecanismo mediante el cual el arreglo pueda agrandarse y reducirse según sea necesario. Dos operadores en C++, `new` y `delete`, que proporcionan esta capacidad se describen en la tabla 12.2. (Estos operadores requieren el archivo de encabezado `stdlib.h`).

Tabla 12.2 Los operadores `new` ***y*** `delete` ***(requieren el archivo de encabezado*** `new`***)***

Nombre del operador	Descripción
`new`	Reserva el número de bytes solicitado por la declaración. Devuelve la dirección de la primera ubicación reservada o `NULL` si no se dispone de suficiente memoria.
`delete`	Libera un bloque de bytes reservado con anterioridad. La dirección de la primera ubicación reservada debe transmitirse como un argumento al operador.

Las solicitudes de almacenamiento dinámico explícito de variables escalares o arreglos se hacen como parte de una instrucción de declaración o de asignación.[6] Por ejemplo, la instrucción de declaración `int *num = new int;` reserva un área suficiente para contener un número entero y coloca la dirección de esta área de almacenamiento en el apuntador `num`. Esta misma asignación dinámica puede hacerse declarando primero el apuntador usando la instrucción de declaración `int *num;` y luego asignar al apuntador una dirección con la instrucción de asignación `num = new int;`. En cualquier caso el área de almacenamiento asignada proviene del área de almacenamiento libre de la computadora.[7]

[5] Este tema puede omitirse en la primera lectura sin perder la continuidad temática.

[6] Debe señalarse que el compilador proporciona de manera automática esta asignación y desasignación dinámicas desde la pila para todas las variables auto.

[7] El área de almacenamiento libre de una computadora se conoce de manera formal como *montículo*. El montículo consiste en memoria no asignada que puede asignarse a un programa, según se requiera, mientras el programa se está ejecutando.

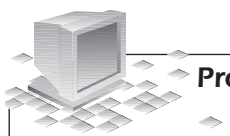
Una manera similar y de mayor utilidad es la asignación dinámica de arreglos. Por ejemplo, la declaración

```
int *calif = new int[200];
```

reserva un área suficiente para almacenar 200 números enteros y coloca la dirección del primer número entero en el apuntador `calif`. Aunque se ha usado la constante 200 en este ejemplo de declaración, puede usarse una dimensión variable. Por ejemplo, considérese la secuencia de instrucciones

```
cout << "Introduzca el número de calificaciones que se van
a procesar: ";
cin >> numcalif;
int *calif = new int[numcalif];
```

En esta secuencia el tamaño real del arreglo que se crea depende del número introducido por el usuario. Dado que los nombres del apuntador y del arreglo están relacionados, puede tenerse acceso a cada valor en el área de almacenamiento recién creada usando una notación de arreglo estándar, como `calif[i],` en lugar de la notación de apuntador equivalente `*(calif + i)`. El programa 12.7 ilustra esta secuencia de código en el contexto de un programa completo.



Programa 12.7

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

int main()
{
  int numcalif, i;

  cout << "Introduzca el número de calificaciones que se van a procesar: ";
  cin >> numcalif;

  int *calif = new int[numcalif]; // crea el arreglo

  for (i = 0; i < numcalif; i++)
  {
    cout << " Introduzca una calificación: ";
    cin >> calif[i];
  }
  cout << "\nSe creó un arreglo para " << numcalif << " números enteros\n";
  cout << " Los valores almacenados en el arreglo son: ";
  for (i = 0; i < numcalif; i++)
    cout << "\n   " << calif[i]
  cout << endl;

  delete[] calif;     // devuelve el almacenamiento al montículo

  return 0;
}
```

En el programa 12.7 se puede observar que el operador `delete` se usa con corchetes siempre que se haya empleado antes el operador `new` para crear un arreglo. La instrucción `delete[]` reintegra al sistema operativo el bloque de almacenamiento asignado mientras el programa se está ejecutando.[8] La única dirección requerida por `delete` es la dirección inicial del bloque de almacenamiento que fue asignado en forma dinámica. Por tanto, cualquier dirección devuelta por new puede ser usada en lo subsiguiente por `delete` para reintegrar a la computadora la memoria reservada. El operador `delete` no altera la dirección que se le transmite, sino tan sólo elimina el almacenamiento al que hace referencia la dirección. A continuación hay una muestra de la ejecución del programa 12.7:

```
Introduzca el número de calificaciones que se van a proce-
sar: 4
  Introduzca una calificación: 85
  Introduzca una calificación: 96
  Introduzca una calificación: 77
  Introduzca una calificación: 92

Se creó un arreglo para 4 números enteros
 Los valores almacenados en el arreglo son:
   85
   96
   77
   92
```

Ejercicios 12.2

1. Reemplace cada una de las siguientes referencias a una variable subindexada con una referencia a un apuntador.

   a. `precios[5]` **b.** `calif[2]` **c.** `produc[10]`
   d. `dist[9]` **e.** `millas[0]` **f.** `temp[20]`
   g. `celsius[16]` **h.** `num[50]` **i.** `tiempo[12]`

2. Reemplace cada una de las siguientes referencias usando un apuntador con una referencia subindexada.

   a. `*(mensaje + 6)` **b.** `*cant` **c.** `*(anios + 10)`
   d. `*(existencias + 2)` **e.** `*(tasas + 15)` **f.** `*(codigos + 19)`

3. **a.** Enliste las tres acciones que causan que el compilador genere la instrucción de declaración `double pendientes[5];`.
   b. Si cada número de precisión doble usa ocho bytes de almacenamiento, ¿cuánto almacenamiento se reserva para el arreglo `pendientes`?
   c. Trace un diagrama parecido a la figura 12.14 para el arreglo `pendientes`.
   d. Determine el desplazamiento en bytes relativo al inicio del arreglo `pendientes`, correspondiente al desplazamiento en la expresión `*(pendientes + 3)`.

[8] El almacenamiento asignado será devuelto de manera automática al montículo, por el sistema operativo, cuando el programa haya completado su ejecución. Sin embargo, dado que esto no siempre sucede así, es importante en extremo reintegrar formalmente al montículo la memoria asignada de manera dinámica cuando el almacenamiento ya no se necesita. Se usa el término **fuga de memoria** para describir la condición que ocurre cuando la memoria asignada en forma dinámica no se devuelve formalmente usando el operador `delete` y el sistema operativo no reclama el área de memoria asignada.

4. Escriba una declaración para almacenar los siguientes valores en un arreglo llamado `tasas`: 12.9, 18.6, 11.4, 13.7, 9.5, 15.2, 17.6. Incluya la declaración en un programa que despliegue los valores en el arreglo usando notación de apuntador.

12.3 ARITMÉTICA DE APUNTADORES

Las variables apuntadoras, como todas las variables, contienen valores. El valor almacenado en un apuntador es, por supuesto, una dirección. Por tanto, al sumar y restar números a los apuntadores se pueden obtener direcciones diferentes. Además, las direcciones en los apuntadores pueden compararse usando cualquiera de los operadores relacionales (`==`, `!=`, `<` , `>`, etc.) que son válidos para comparar otras variables. Al realizar aritmética en los apuntadores debe tenerse cuidado de producir direcciones que apunten a algo significativo. Al comparar apuntadores también deben hacerse comparaciones que tengan sentido. Considérense las declaraciones:

```
int nums[100];
int *nPt;
```

Para establecer la dirección de `nums[0]` en `nPt` se puede usar cualquiera de las siguientes dos instrucciones de asignación:

```
nPt = &nums[0];
nPt = nums;
```

Las dos instrucciones de asignación producen el mismo resultado debido a que `nums` es una constante apuntadora que contiene la dirección de la primera ubicación en el arreglo. Ésta es, por supuesto, la dirección de `nums[0]`. La figura 12.15 ilustra la asignación de memoria resultante de las instrucciones de declaración y de asignación previas, suponiendo que cada número entero requiere cuatro bytes de memoria y que la ubicación del inicio del arreglo `nums` está en la dirección 18934.

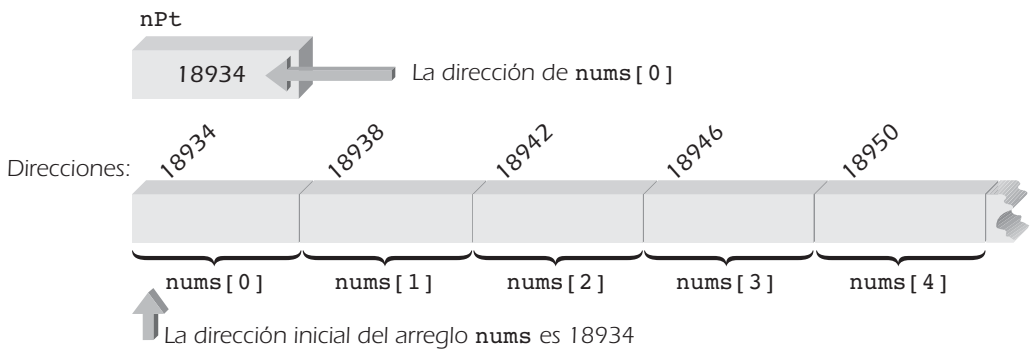


Figura 12.15 El arreglo `nums` en la memoria.

Una vez que `nPt` contiene una dirección válida, pueden sumarse y restarse valores de la dirección para producir direcciones nuevas. Cuando se suman o restan números a apuntadores, la computadora ajusta de manera automática el número para asegurar que el resultado aún "apunta" a un valor del tipo correcto. Por ejemplo, la instrucción `nPt = nPt +4;` obliga a la computadora a escalar el 4 por el número correcto para asegurar que la

dirección resultante es la dirección de un número entero. Suponiendo que cada número entero requiere cuatro bytes de almacenamiento, como si ilustró en la figura 12.15, la computadora multiplica el 4 por 4 y suma 16 a la dirección en `nPt`. La dirección resultante es 18950, la cual es la dirección correcta de `nums[4]`.

Este escalamiento automático que hace la computadora asegura que la expresión `nPt + i`, donde `i` es cualquier número entero positivo, apunte en forma correcta al `i-ésimo` elemento delante de aquel al que apunta en la actualidad `nPt`. Por tanto, si `nPt` contiene inicialmente la dirección de `nums[0]`, `nPt + 4` es la dirección de `nums[4]`, `nPt + 50` es la dirección de `nums[50]` y `nPt + i` es la dirección de `nums[i]`. Aunque se han usado direcciones reales en la figura 12.15 para ilustrar el proceso de escalamiento, el programador no necesita conocer ni preocuparse por las direcciones reales utilizadas por la computadora. La manipulación de direcciones usando apuntadores por lo general no requiere el conocimiento de la dirección real.

Las direcciones también pueden aumentarse o disminuirse usando prefijos y posfijos con operadores de incremento y decremento. Agregar uno a un apuntador causa que éste apunte al siguiente elemento del tipo al que se apunta. Disminuir un apuntador causa que éste apunte al elemento anterior. Por ejemplo, si la variable apuntadora `p` es un apuntador a un número entero, la expresión `p++` causa que la dirección en el apuntador sea incrementada para apuntar al siguiente número entero. Esto se ilustra en la figura 12.16.

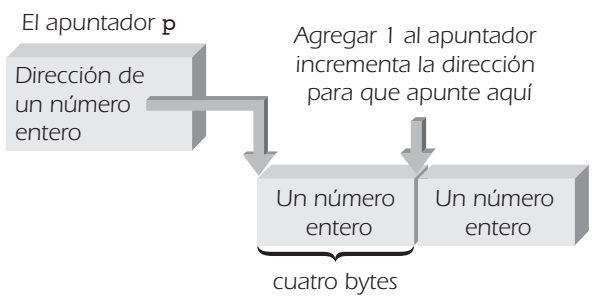


Figura 12.16 *Los incrementos se escalan cuando se usan con apuntadores.*

Al revisar la figura 12.16 se puede observar que el incremento añadido al apuntador es escalado en forma correcta para justificar el hecho que el apuntador se usa para apuntar a números enteros. Por supuesto, le corresponde al programador asegurarse que el tipo de datos correctos está almacenado en la nueva dirección contenida en el apuntador.

Los operadores de incremento y decremento pueden aplicarse como operadores de apuntador de prefijo o posfijo. Todas las siguientes combinaciones que usan apuntadores son válidas:

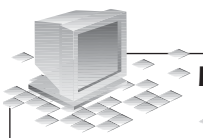
```
*ptNum++    // usa el apuntador y luego lo incrementa
*++ptNum    // incrementa el apuntador antes de usarlo
*ptNum--    // usa el apuntador y luego lo decrementa
*--ptNum    // decrementa el apuntador antes de usarlo
```

De las cuatro formas posibles, la más usada es la forma `*ptNum++`. Esto se debe a que dicha expresión permite que se tenga acceso a cada elemento en el arreglo conforme la dirección "se mueve" desde la dirección inicial del arreglo hasta la dirección del último elemento del arreglo. El uso del operador de incremento se muestra en el programa 12.8. En este programa cada elemento en el arreglo `nums` se recupera incrementando de manera sucesiva la dirección en `nPt`.

La salida producida por el programa 12.8 es:

```
El total de los elementos del arreglo es 115
```

La expresión `total = total + *nPt++` usada en el programa 12.8 acumula los valores "a los que apunta" la variable apuntadora `nPt`. Dentro de esta expresión, el término `*nPt++` causa primero que la computadora recupere el número entero al que apunta `nPt`. Esto lo realiza la parte `*nPt` del término. El posfijo de incremento, `++`, añade luego uno a la dirección en `nPt` de modo que `nPt` ahora contiene la dirección del siguiente elemento del arreglo. El incremento es escalado, por supuesto, por la computadora de modo que la dirección real en `nPt` es la dirección correcta del siguiente elemento.



Programa 12.8

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

int main()
{
  const int NUMS = 5;

  int nums[NUMS] = {16, 54, 7, 43, -5};
  int i, total = 0, *nPt;

  nPt = nums;     // almacena la dirección de nums[0] en nPt
  for (i = 0; i < NUMS; i++)
    total = total + *nPt++;
  cout << "El total de los elementos del arreglo es " << total << endl;

  return 0;
}
```

Los apuntadores también pueden compararse. Esto es útil en particular cuando se trata con apuntadores que apuntan a elementos en el mismo arreglo. Por ejemplo, en lugar de usar un contador en un ciclo `for` para tener acceso en forma correcta a cada elemento en un arreglo, la dirección en un apuntador puede compararse con la dirección inicial y final del mismo arreglo. La expresión

```
nPt <= &nums[4]
```

es verdadera (no es cero) en tanto la dirección en `nPt` sea menor que o igual a la dirección de `nums[4]`. Dado que `nums` es una constante apuntadora que contiene la dirección de `nums[0]`, el término `&nums[4]` puede reemplazarse por el término equivalente `nums + 4`. Usando cualquiera de estas formas, el programa 12.8 puede volver a escribirse en el programa 12.9 para continuar añadiendo elementos del arreglo mientras la dirección en `nPt` sea menor que o igual a la dirección del último elemento del arreglo.

En el programa 12.9 se usó la forma compacta de la expresión de acumulación, `total += *nPt++`, en lugar de la forma más larga, `total = total + *nPt++`. Además, la expresión `nums + 4` no cambia la dirección en `nums`. Dado que `nums` es un nombre de arreglo y no una variable apuntadora, su valor no puede cambiarse. La expresión `nums + 4` recupera primero la dirección en `nums`, suma 4 a esta dirección (escalada de manera apropiada) y usa el resultado con propósitos de comparación. Expresiones como `*nums++`, la cual intenta cambiar la dirección, son inválidas. Expresiones como `*nums` o `*(nums + i)`, las cuales usan la dirección sin intentar alterarla, son válidas.

Programa 12.9

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

int main()
{
  const int NUMS = 5;

  int nums[NUMS] = {16, 54, 7, 43, -5};
  int total = 0, *nPt;

  nPt = nums;     // almacena la dirección de nums[0] en nPt
  while (nPt < nums + NUMS)
    total += *nPt++;
  cout << "El total de los elementos del arreglo es " << total << endl;

  return 0;
}
```

Inicialización de apuntadores

Como todas las variables, los apuntadores pueden inicializarse cuando se declaran. Sin embargo, cuando se inicializan apuntadores debe tenerse cuidado en establecer una dirección en el apuntador. Por ejemplo, una inicialización como

```
int *ptNum = &millas;
```

sólo es válida si `millas` en sí es declarada como una variable en número entero antes que `ptNum`. Aquí se crea un apuntador a un número entero y se establece la dirección en el apuntador a la dirección de una variable en número entero. Si la variable `millas` se declara después de declarar `ptNum`, como sigue,

```
int *ptNum = &millas;
int millas;
```

ocurre un error. Esto se debe a que se usa la dirección de `millas` antes de definir `millas`. En vista que el área de almacenamiento reservada para `millas` no se ha asignado cuando se declara `ptNum`, la dirección de millas no existe todavía.

Los apuntadores a arreglos también pueden inicializarse dentro de sus instrucciones de declaración. Por ejemplo, si se ha declarado `voltios` como un arreglo de números de precisión doble, puede usarse cualquiera de las siguientes dos declaraciones para inicializar el apuntador nombrado `zing` a la dirección del primer elemento en `voltios`:

```
double *zing = &voltios[0];
double *zing = voltios;
```

La última inicialización es correcta porque `voltios` es en sí misma una constante apuntadora que contiene una dirección del tipo apropiado. (En este ejemplo se seleccionó el nombre de la variable `zing` para reforzar la idea que cualquier nombre de variable puede seleccionarse para un apuntador.)

Ejercicios 12.3

1. Reemplace la instrucción `while` en el programa 12.9 con una instrucción `for`.
2. **a.** Escriba un programa que almacene los siguientes números en el `arreglo` nombrado tasas: 6.25, 6.50, 6.8, 7.2, 7.35, 7.5, 7.65, 7.8, 8.2, 8.4, 8.6, 8.8, 9.0. Despliegue los valores en el arreglo cambiando la dirección en un apuntador llamado `dispPt`. Use una instrucción `for` en su programa.
   b. Modifique el programa escrito en el ejercicio 2a para usar una instrucción `while`.
3. Escriba un programa que almacene los siguientes números en el arreglo llamado `millas`: 15, 22, 16, 18, 27, 23, 20. Haga que su programa copie los datos almacenados en `millas` a otro arreglo llamado `dist` y luego despliegue los valores en el arreglo `dist`. Haga que su programa use una notación de apuntador cuando copie y despliegue elementos del arreglo.
4. Escriba un programa que declare tres arreglos unidimensionales llamados `millas, galones` y `mpg`. Cada arreglo deberá ser capaz de contener diez elementos. En el arreglo `millas` almacene los números 240.5, 300.0, 189.6, 310.6, 280.7, 216.9, 199.4, 160.3, 177.4, 192.3. En el arreglo `galones` almacene los números 10.3, 15.6, 8.7, 14, 16.3, 15.7, 14.9, 10.7, 8.3, 8.4. Cada elemento del arreglo `mpg` deberá calcularse como el elemento correspondiente del arreglo `millas` dividido entre el elemento equivalente del arreglo `galones`: por ejemplo, `mpg[0] = millas[0] / galones[0]`. Use apuntadores cuando calcule y despliegue los elementos del arreglo `mpg`.
5. Defina un arreglo de diez apuntadores a números de precisión doble. Luego lea diez números en las ubicaciones individuales referenciadas por los apuntadores. Ahora sume todos los números y almacene el resultado en una ubicación referenciada a un apuntador. Despliegue el contenido de todas las ubicaciones.

12.4 TRANSMISIÓN DE DIRECCIONES

Ya se ha visto un método para transmitir direcciones a una función. Esto se logró usando variables de referencia, como se describió en la sección 6.3. Aunque la transmisión de variables de referencia a una función le proporciona a ésta la dirección de las variables trans-

mitidas, es un uso implícito de las direcciones debido a que la llamada a la función no revela el hecho que se están usando variables de referencia. Por ejemplo, la llamada a la función `intercambio(num1, num2);` no revela si `num1` o `num2` es una variable de referencia. Sólo al observar la declaración para estas variables o examinar la línea de encabezado de la función para `intercambio()` se revelan los tipos de datos de `num1` y `num2`.

En contraste con la transmisión implícita de direcciones usando variables de referencia, las direcciones pueden transmitirse de manera explícita usando variables apuntadoras. Veamos cómo se logra esto.

Para transmitir de manera explícita una dirección a una función todo lo que se necesita hacer es colocar el operador de dirección, `&`, enfrente de la variable que se está transmitiendo. Por ejemplo, la llamada a la función

```
intercambio(&primernum, &segundonum);
```

transmite las direcciones de las variables `primernum` y `segundonum` a `intercambio()`, como se ilustra en la figura 12.17. Transmitir direcciones en forma explícita usando el operador de dirección en efecto es una *transmisión por referencia* debido a que la función llamada puede hacer referencia, o tener acceso, a las variables en la función que llama usando las direcciones transmitidas. Como se vio en la sección 6.3, las llamadas por referencia también se logran usando parámetros de referencia. Aquí se usarán las direcciones transmitidas y apuntadores para tener acceso directo a las variables `primernum` y `segundonum` desde adentro de `intercambio()` y permutar sus valores, un procedimiento que se logró con anterioridad en el programa 6.8 usando parámetros de referencia.

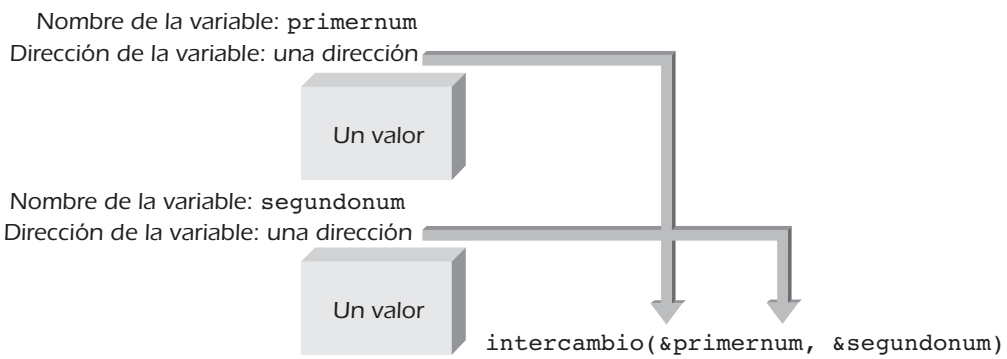


Figura 12.17 *Transmisión explícita de direcciones a* `intercambio()`.

Uno de los primeros requisitos al escribir `intercambio()` es construir una línea de encabezado de función que reciba y almacene en forma correcta los valores transmitidos, que en este caso son dos direcciones. Como se vio en la sección 12.1, las direcciones se almacenan en apuntadores, lo cual significa que los parámetros de `intercambio()` deben declararse como apuntadores.

Suponiendo que `primernum` y `segundonum` son variables de precisión doble, y que `intercambio()` no devuelve un valor, una línea de encabezado de función adecuada para `intercambio` es

```
void intercambio(double *dirNm1, double *dirNm2);
```

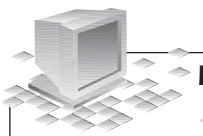
La elección de los nombres de parámetros `dirNm1` y `dirNm2`, como todos los nombres de parámetros, le corresponde al programador. Sin embargo, la declaración `double *dirNm1` declara que el parámetro llamado `dirNm1` se usará para almacenar la dirección

de un valor de precisión doble. Del mismo modo, la declaración `double *dirNm2` declara que `dirNm2` también almacenará la dirección de un valor de precisión doble.

Antes de escribir el cuerpo de `intercambio()` para permutar los valores en `primernum` y `segundonum`, se comprobará primero que los valores a los que se tiene acceso usando las direcciones en `dirNm1` y `dirNm2` son correctos. Esto se hace en el programa 12.10.

La salida desplegada cuando se ejecuta el programa 12.10 es

```
El número cuya dirección está en dirNm1 es 20.5
El número cuya dirección está en dirNm2 es 6.25
```



Programa 12.10

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

void intercambio(double *, double *);      // prototipo de la función

int main()
{
  double primernum = 20.5, segundonum = 6.25;

  intercambio(&primernum, &segundonum);    // llama a intercambio

  return 0;
}

// esta función ilustra la transmisión de argumentos apuntadores
void intercambio(double *dirNm1, double *dirNm2)
{

  cout << "El número cuya dirección está en dirNm1 es "
       << *dirNm1 << endl;
  cout << "El número cuya dirección está en dirNm1 es "
       << *dirNm2 << endl;

  return;
}
```

Al revisar el programa 12.10 se notarán dos cosas. Primera, el prototipo de la función para `intercambio()`

```
void intercambio(double * , double*)
```

declara que `intercambio()` no devuelve ningún valor en forma directa y que sus parámetros son dos apuntadores que "apuntan" a valores de precisión doble. Como tal, cuando se llame a la función ésta requerirá que se transmitan dos direcciones, y que cada una sea la dirección de un valor de precisión doble.

Segunda, que dentro de `intercambio()` se usa el operador de indirección para tener acceso a los valores almacenados en `primernum` y `segundonum`. La función `inter-`

`cambio()` en sí no tiene conocimiento de estos nombres de variables, pero tiene la dirección de `primernum` almacenado en `dirNm1` y la dirección de `segundonum` almacenada en `dirNm2`. La expresión `*dirNm1` usada en la primera instrucción `cout` significa "la variable cuya dirección está en `dirNm1`". Ésta es por supuesto la variable `primernum`. Del mismo modo, la segunda instrucción `cout` obtiene el valor almacenado en `segundonum` como "la variable cuya dirección está en `dirNm2`". Por tanto, se han usado con éxito apuntadores para permitir que `intercambio()` tenga acceso a variables en `main()`. La figura 12.18 ilustra el concepto de almacenar direcciones en parámetros.

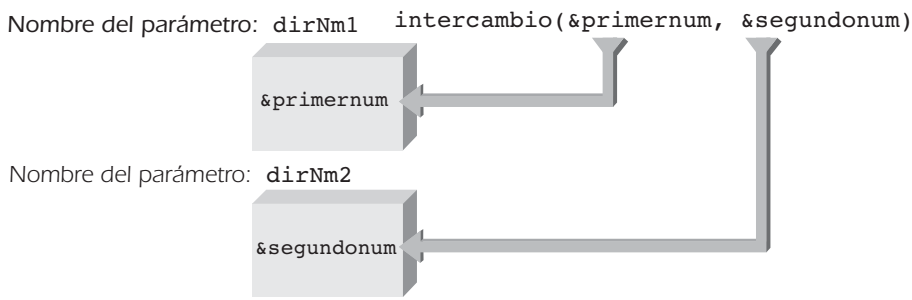


Figura 12.18 *Almacenar direcciones en parámetros.*

Habiendo verificado que `intercambio()` puede tener acceso a las variables locales de `main()` `primernum` y `segundonum`, ahora se puede expandir `intercambio()` para permutar los valores en estas variables. Los valores en las variables `primernum` y `segundonum` de `main()` pueden intercambiarse desde dentro de `intercambio()` usando el algoritmo de intercambio de tres pasos descrito antes en la sección 6.3, el cual por comodidad se muestra de nuevo a continuación:

1. Almacenar el valor de `primernum` en una ubicación temporal.
2. Almacenar el valor de `segundonum` en `primernum`.
3. Almacenar el valor temporal en `segundonum`.

Usando apuntadores desde dentro de `intercambio()`, esto adopta la forma:

1. Almacenar el valor de la variable a la que apunta `dirNm1` en una ubicación temporal. La instrucción `temp = *dirNm1;` hace esto (véase la figura 12.19).
2. Almacenar el valor de la variable cuya dirección está en `dirNm2` en la variable cuya dirección están en `dirNm1`. La instrucción `*dirNm1 = *dirNm2;` hace esto (véase la figura 12.20).
3. Mover el valor en la ubicación temporal a la variable cuya dirección está en `dirNm2`. La instrucción `*dirNm2 = temp;` hace esto (véase la figura 12.21).

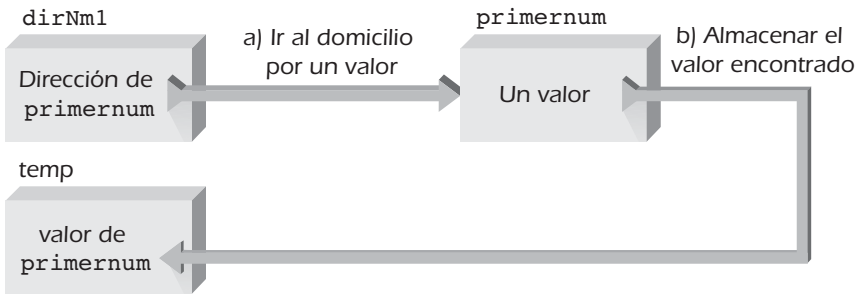


Figura 12.19 *Almacenar el valor de* `primernum` *en forma indirecta.*

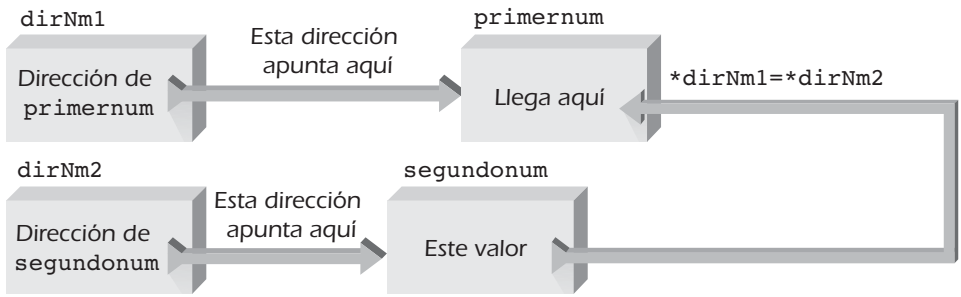


Figura 12.20 *Cambiar de manera indirecta el valor de* `primernum`.

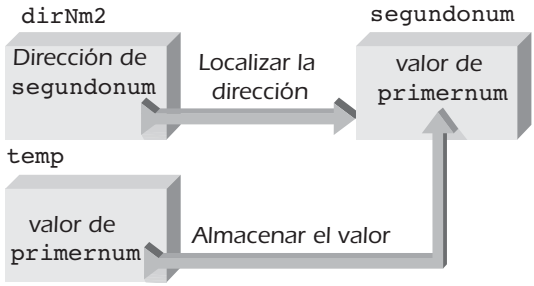


Figura 12.21 *Cambiar en forma indirecta el valor de* `segundonum`.

El programa 12.11 contiene la forma final de `intercambio()`, escrito de acuerdo con nuestra descripción.

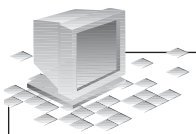
La siguiente muestra de ejecución se obtuvo usando el programa 12.11:

```
El valor almacenado en primernum es: 20.5
El valor almacenado en segundonum es: 6.25

El valor almacenado ahora en primernum es: 6.25
El valor almacenado ahora en segundonum es: 20.5
```

Como lo ilustra esta salida, los valores almacenados en las variables de `main()` se han modificado desde dentro de `intercambio()`, lo cual se hizo posible por el uso de apuntadores. El lector interesado deberá comparar esta versión de `intercambio()` con la versión que utiliza referencias que se presentó en el programa 6.10. La ventaja de usar apuntadores en lugar de referencias es que la llamada a la función en sí designa de manera explícita qué direcciones se van a usar, lo cual es una alerta directa de que la función con toda probabilidad alterará variables de la función que llama. Las ventajas de usar referencias es que la notación es mucho más simple.

En general, para funciones como inter`cambio()`, gana la conveniencia de la notación y se usan referencias. Sin embargo, al transmitir arreglos a funciones, lo cual es el siguiente tema, el compilador transmite de manera automática una dirección. Esto dicta que se usarán variables apuntadoras para almacenar la dirección.



Programa 12.11

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

void intercambio(double *, double *);      // prototipo de la función

int main()
{
  double primernum = 20.5, segundonum = 6.25;


  cout << "El valor almacenado en primernum es: " << primernum << endl;
  cout << "El valor almacenado en segundonum es: " << segundonum << "\n\n";

  intercambio(&primernum, &segundonum);    // llama a intercambio

  cout << "El valor almacenado ahora en primernum es: "
       << primernum << endl;
  cout << "El valor almacenado ahora en segundonum es: "
       << segundonum << endl;

  return 0;
}

// esta función intercambia los valores en sus dos argumentos
void intercambio(double *dirNm1, double *dirNm2)
{
  double temp;
```

(Continúa)

(Continuación)

```
   temp = *dirNm1;              // guarda el valor de primernum
   *dirNm1 = *dirNm2;           // mueve el valor de segundonum a primernum
   *dirNm2 = temp;              // cambia el valor de segundonum

   return;
}
```

Transmisión de arreglos

Cuando se transmite un arreglo a una función, su dirección es el único elemento que se transmite en realidad. Con esto se quiere decir la dirección de la primera ubicación usada para almacenar el arreglo, como se ilustra en la figura 12.22. Dado que la primera ubicación reservada para un arreglo corresponde al elemento 0 del arreglo, la "dirección del arreglo" también es la dirección del elemento 0.

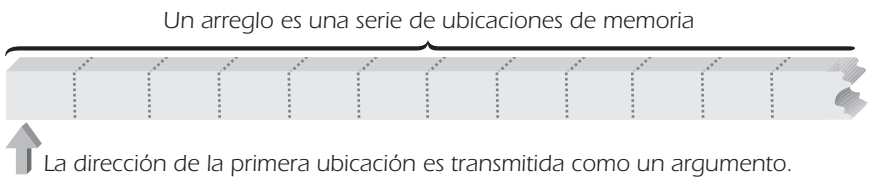
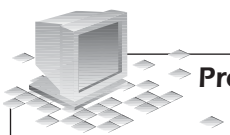


Figura 12.22 *La dirección de un arreglo es la dirección de la primera ubicación reservada para el arreglo.*

Para proporcionar un ejemplo específico en el cual un arreglo es transmitido a una función, considérese el programa 12.12. En este programa, el arreglo `nums` es transmitido a la función `hallarMax()` usando notación de arreglos convencional.



Programa 12.12

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

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

int main()
{
  const int NUMPTS = 5;

  int nums[NUMPTS] = {2, 18, 1, 27, 16};
```

(Continúa)

(Continuación)

```
  cout << "\nEl valor máximo es "
       << hallarMax(nums,NUMPTS) << endl;
  return 0;
}
// esta función devuelve el valor máximo en un arreglo de números enteros
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;
}
```

La salida desplegada cuando se ejecuta el programa 12.12 es

```
El valor máximo es 27
```

El parámetro llamado `vals` en la declaración de la línea de encabezado para `hallarMax()` en realidad recibe la dirección del arreglo `nums`. Como tal, `vals` en realidad es un apuntador, dado que los apuntadores son variables (o parámetros) usados para almacenar direcciones. En vista que la dirección transmitida a `hallarMax()` es la dirección de un número entero, otra línea de encabezado adecuada para `hallarMax()` es

```
int hallarMax(int *vals, int numels) // aquí vals se declara como
                                     // un apuntador a un numero entero
```

La declaración `int *vals` en la línea de encabezado declara que `vals` se usa para almacenar una dirección de un número entero. La dirección almacenada es, por supuesto, la ubicación del inicio de un arreglo. La siguiente es una versión replanteada de la función `hallarMax()` que usa la nueva declaración de apuntador para `vals`, pero conserva el uso de subíndices para referirse a elementos individuales del arreglo:

```
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;
}
```

Sin tomar en cuenta cómo se declaró `vals` en el encabezado de la función o cómo se usa dentro del cuerpo de la función, en verdad es una variable apuntadora. Por tanto, la dirección en `vals` puede modificarse. Esto no sucede con el nombre `nums`. Dado que nums es el nombre del arreglo originalmente creado, es una constante apuntadora. Como se describió en la sección 12.2, esto significa que la dirección en `nums` no puede cambiarse y que

no puede tomarse la dirección de `nums` en sí. Sin embargo, ninguna de estas restricciones se aplican a la variable apuntadora llamada `vals`. Toda la aritmética de direcciones que se aprendió en la sección anterior puede aplicarse de manera legítima a `vals`.

Escribiremos dos versiones adicionales de `hallarMax()`, ambas usando apuntadores en lugar de subíndices. En la primera versión tan sólo se sustituye la notación de apuntador por notación de subíndice. En la segunda versión usaremos aritmética de direcciones para cambiar la dirección en el apuntador.

Como ya se señaló, el acceso a un elemento del arreglo usando la notación de subíndice `nombreArreglo[i]` siempre puede reemplazarse por la notación de apuntador `*(nombreArreglo + i)`. En la primera modificación a `hallarMax()`, se puede usar esta correspondencia con sólo reemplazar todas las referencias a `vals[i]` con la expresión equivalente `*(vals + i)`.

```
int hallarMax(int *vals, int numels) // halla el valor máximo
{
  int i, max = *vals;

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

  return max;
}
```

La siguiente versión de `hallarMax()` aprovecha el hecho que puede cambiarse la dirección almacenada en `vals`. Después que se recupera cada elemento del arreglo usando la dirección en `vals`, la dirección en sí se incrementa en uno en la lista que se altera en la instrucción `for`. La expresión `max = *vals` usada antes para establecer `max` con el valor de `vals[0]` es reemplazada por la expresión `max = *vals++`, la cual ajusta la dirección en vals para que apunte al segundo elemento en el arreglo. El elemento asignado a `max` por esta expresión es el elemento del arreglo al que apunta `vals` antes que se incremente `vals`. El posfijo de incremento, `++`, no cambia la dirección en `vals` hasta después que se ha usado la dirección para recuperar el primer elemento del arreglo.

```
int hallarMax(int *vals, int numels) // encuentra el valor máximo
{
  int i, max = *vals++; // obtiene el primer elemento y lo
                        incrementa
  for (i = 1; i < numels; i++, vals++)
  {
    if (max < *vals)
      max = *vals;
  }
  return max;
}
```

Ahora se revisará esta versión de `hallarMax()`. Al principio el valor máximo se establece como "el elemento al que apunta `vals`". Dado que `vals` contiene al principio la dirección del primer elemento en el arreglo transmitido a `hallarMax()`, el valor de este primer elemento se almacena en `max`. La dirección en `vals` se incrementa entonces en uno. El uno que se suma a `vals` se escala de manera automática con el número de bytes

usados para almacenar números enteros. Por tanto, después del incremento, la dirección almacenada en `vals` es la dirección del siguiente elemento del arreglo. Esto se ilustra en la figura 12.23. El valor de este siguiente elemento se compara con el máximo y la dirección se incrementa de nuevo, esta vez desde dentro de la lista de alteración de la instrucción `for`. Este proceso continúa hasta que se han examinado todos los elementos del arreglo.

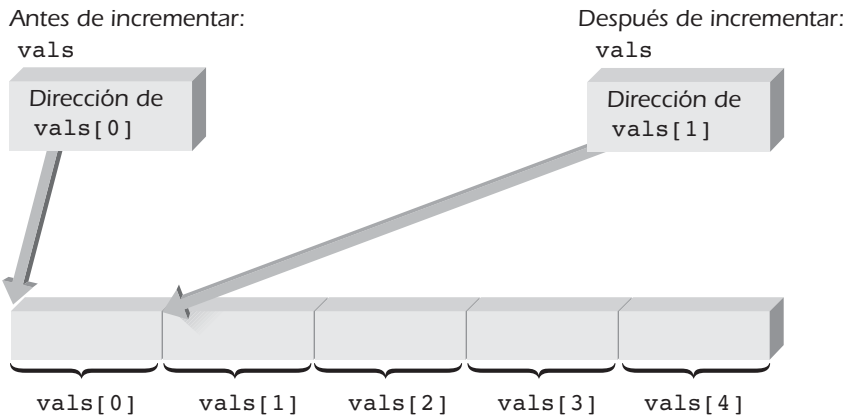


Figura 12.23 *Apuntando a diferentes elementos.*

La versión de `hallarMax()` que deberá elegir es cuestión de estilo y gusto personales. En general, los programadores principiantes se sienten más cómodos usando subíndices en lugar de apuntadores. Además, si el programa usa un arreglo como la estructura de almacenamiento natural para la aplicación y los datos disponibles, el acceso a un arreglo usando subíndices es más apropiado para indicar con claridad la intención del programa. Sin embargo, conforme se aprende sobre estructuras de datos, el uso de apuntadores se vuelve una herramienta cada vez más útil y poderosa por su propio derecho. En estos casos no hay una equivalencia simple o fácil para el uso de subíndices.

Puede deducirse un "truco ingenioso" de esta exposición. Dado que la transmisión de un arreglo a una función en realidad implica la transmisión de una dirección, se puede transmitir igual de bien cualquier dirección válida. Por ejemplo, la llamada a la función `hallarMax(&nums[2],3)` transmite la dirección de `nums[2]` a `hallarMax()`. Dentro de `hallarMax()` el apuntador `vals` almacena la dirección y la función comienza la búsqueda de un máximo en el elemento correspondiente a esta dirección. Por tanto, desde la perspectiva de `hallarMax()`, ha recibido una dirección y procede en forma apropiada.

Notación avanzada para apuntadores[9]

También puede tenerse acceso a arreglos multidimensionales usando notación de apuntador, aunque la notación se vuelve cada vez más críptica conforme aumentan las dimensiones del

[9]Este tema puede omitirse sin perder la continuidad temática.

arreglo. Una aplicación muy útil de esta notación ocurre con los arreglos bidimensionales de caracteres. Aquí se considerará la notación de apuntador para arreglos bidimensionales numéricos. Por ejemplo, considérese la declaración

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

Esta declaración crea un arreglo de elementos y un conjunto de constantes apuntadoras llamadas nums, `nums[0]` y `nums[1]`. La relación entre estas constantes apuntadoras y los elementos del arreglo `nums` se ilustra en la figura 12.24.

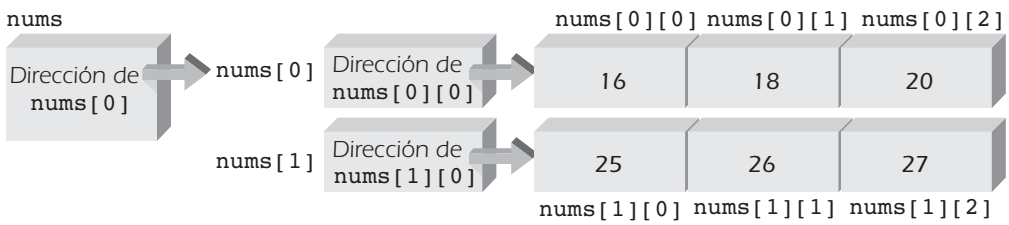


Figura 12.24 *Almacenamiento del arreglo* `nums` *y constantes apuntadoras asociadas.*

La disponibilidad de las constantes apuntadoras asociadas con un arreglo bidimensional permite tener acceso a los elementos del arreglo en una variedad de formas. Una de ellas es considerar el arreglo bidimensional como un arreglo de filas, donde cada fila en sí misma es un arreglo de tres elementos. Considerado bajo esta luz, la dirección del primer elemento en la primera fila es proporcionada por `nums[0]` y la dirección del primer elemento en la segunda fila es proporcionada por `nums[1]`. Por tanto, la variable a la que apunta `nums[0]` es `nums[0][0]` y la variable a la que apunta `nums[1]` es `nums[1][0]`. Una vez que se entiende la naturaleza de estas constantes, puede tenerse acceso a cada elemento en el arreglo al aplicar un desplazamiento apropiado al apuntador apropiado. Por tanto, las siguientes notaciones son equivalentes:

Notación de apuntador	Notación de subíndice	Valor
`*nums[0]`	`nums[0][0]`	16
`*(nums[0] + 1)`	`nums[0][1]`	18
`*(nums[0] + 2)`	`nums[0][2]`	20
`*nums[1]`	`nums[1][0]`	25
`*(nums[1] + 1)`	`nums[1][1]`	26
`*(nums[1] + 2)`	`nums[1][2]`	27

Ahora es posible avanzar aún más y reemplazar `nums[0]` y `nums[1]` con sus respectivas notaciones de apuntador, usando la dirección de la misma `nums`. Como se ilustra en la figura 12.24, la variable a la que apunta nums es `nums[0]`. Es decir, `*nums es nums[0]`.

Del mismo modo, `*(nums + 1)` es `nums[1]`. Usar estas relaciones conduce a las siguientes equivalencias:

Notación de apuntador	Notación de subíndice	Valor
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1) + 1)`	`nums[1][1]`	26
`*(*(nums + 1) + 2)`	`nums[1][2]`	27

Se aplica la misma notación cuando se transmite un arreglo bidimensional a una función. Por ejemplo, supóngase que el arreglo bidimensional `nums` se transmite a la función `calc()` usando la llamada `calc(nums);`. Aquí, como con todas las transmisiones de arreglos, se transmite una dirección. Una línea de encabezado de la función adecuada para la función `calc()` es

```
calc(int pt[2][3])
```

Como ya se ha visto, la declaración de parámetros para `pt` también puede ser

```
calc(int pt[][3])
```

Usando notación de apuntador, otra declaración adecuada es

```
calc(int (*pt)[3])
```

En esta última declaración se requieren los paréntesis interiores para crear un apuntador único para arreglos de tres números enteros. Por supuesto, cada arreglo es equivalente a una sola fila del arreglo `nums`. Al desplazar de manera adecuada el apuntador, puede tenerse acceso a cada elemento en el arreglo. Hay que observar que sin los paréntesis la declaración se vuelve

```
int *pt[3]
```

lo cual crea un arreglo de tres apuntadores, cada uno apuntando a un solo número entero.

Una vez que se hace la declaración correcta para `pt` (puede usarse cualquiera de las tres declaraciones válidas), todas las notaciones siguientes dentro de la función `calc()` son equivalentes:

Notación de apuntador	Notación de subíndice	Valor
`*(*pt)`	`pt[0][0]`	16
`*(*pt+1)`	`pt[0][1]`	18
`*(*pt+2)`	`pt[0][2]`	20
`*(*(pt+1))`	`pt[1][0]`	25
`*(*(pt+1)+1)`	`pt[1][1]`	26
`*(*(pt+1)+2)`	`pt[1][2]`	27

Las últimas dos notaciones usando apuntadores se encuentran en programas en C++ más avanzados. La primera de éstas ocurre debido a que las funciones pueden devolver cualquier tipo de datos escalares en C++ válido, incluyendo apuntadores a cualquiera de estos tipos de datos. Si una función devuelve un apuntador, el tipo de datos al que se está apuntando debe declararse en la declaración de la función. Por ejemplo, la declaración

```
int *calc()
```

declara que `calc()` devuelve un apuntador a un valor en número entero. Esto significa que se devuelve una dirección de una variable en número entero. Del mismo modo, la declaración

```
double *impuestos()
```

declara que `impuestos()` devuelve un apuntador a un valor de precisión doble. Esto significa que se devuelve una dirección de una variable de precisión doble.

Además de declarar apuntadores a números enteros, números de precisión doble y otros tipos de datos de C++, también pueden declararse apuntadores que apunten a (que contengan la dirección de) una función. Los apuntadores a funciones son posibles debido a que los nombres de función, como los nombres de arreglos, son en sí mismos constantes apuntadoras. Por ejemplo, la declaración

```
int (*calc)()
```

declara que `calc()` es un apuntador a una función que devuelve un número entero. Esto significa que `calc` contendrá la dirección de una función, y la función cuya dirección está en la variable `calc` devuelve un valor en número entero. Por ejemplo, si la función `suma()` devuelve un número entero, es válida la asignación `calc = suma;`.

Ejercicios 12.4

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

   ```
   double precios[500];
   ```

   Escriba tres encabezados diferentes para una función llamada `ordenarArreglo()` que acepte el arreglo precios como un parámetro llamado `enArreglo` y no devuelva ningún valor.

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

   ```
   char claves[256];
   ```

   Escriba tres encabezados diferentes para una función llamada `hallarClave()` que acepte el arreglo `claves` como un parámetro llamado seleccionar y no devuelva ningún valor.

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

   ```
   double tasas[256];
   ```

   Escriba tres encabezados diferentes para una función llamada `maximo()` que acepte el arreglo `tasas` como un parámetro llamado `velocidad` y devuelva un valor de precisión doble.

4. Modifique la función `hallarMax()` para localizar el valor mínimo del arreglo transmitido. Escriba la función usando sólo apuntadores.

5. En la última versión de `hallarMax()` presentada, `vals` se incrementaba dentro de la lista de alteración de la instrucción `for`. En cambio, suponga que se hace el incremento dentro de la expresión condicional de la instrucción `if` como sigue:

```
int hallarMax(int *vals, int numels)    // versión incorrecta
{
  int i, max = *vals++; // obtiene el primer elemento y se
                           incrementa

  for (i = 1; i < numels; i++)
    if (max < *vals++)
      max = *vals;
  return (max);
}
```

Esta versión produce un resultado incorrecto. Determine por qué.

6. a. Escriba un programa que tenga una declaración en `main` para almacenar los siguientes números en un arreglo llamado `tasas`: 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 tasas en un parámetro llamado `tasas` y luego despliegue los números usando la notación de apuntador `*(tasas + i)`.

b. Modifique la función `mostrar()` escrita en el ejercicio 6a para alterar la dirección en `tasas`. Use siempre la expresión `*tasas` en lugar de `*(tasas + i)` para recuperar el elemento correcto.

7. a. Determine la salida del siguiente programa:

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

const int FILAS = 2;
const int COLS = 3;

void arr(int [][COLS]);

int main()
{
  int nums[FILAS][COLS] = { {33, 16, 29},
                            {54,67,99}};
  arr(nums);

  return 0;
}
```

```
void arr(int (*val)[COLS])
{
  cout << '\n' << *(*val);
  cout << '\n' << *(*val + 1);
  cout << '\n' << *(*(val + 1) + 2);
  cout << '\n' << *(*val) + 1;

  return;
}
```

b. Dada la declaración para `val` en la función `arr()`, ¿sería válida la notación `val[1][2]` dentro de la función?

12.5 Errores comunes de programación

Al usar el material presentado en este capítulo, debe percatarse de los siguientes posibles errores.

1. Intentar almacenar una dirección en una variable que no ha sido declarada como un apuntador.
2. Usar un apuntador para tener acceso a elementos inexistentes de un arreglo. Por ejemplo, si `nums` es un arreglo de diez números enteros, la expresión `*(nums + 15)` apunta a una ubicación que está seis ubicaciones de número entero adelante del último elemento del arreglo. Debido a que C++ no hace ninguna comprobación de límites en accesos a arreglos, este tipo de error no es detectado por el compilador. Éste es el mismo error, disfrazado en forma de notación de apuntador, que ocurre cuando se usa un subíndice para tener acceso a un elemento del arreglo fuera de los límites.
3. Olvidar usar el conjunto de corchetes, `[]`, después del operador `delete` cuando se desasigna en forma dinámica la memoria que fue asignada antes usando el nuevo operador `[]`.
4. Aplicar de manera incorrecta los operadores de dirección e indirección. Por ejemplo, si `pt` es una variable apuntadora, las expresiones

   ```
   pt = &45
   pt = &(millas + 10)
   ```

   son inválidas porque intentan tomar la dirección de un valor. Sin embargo, hay que observar que la expresión `pt = &millas + 10` es válida. Aquí, 10 se suma a la dirección de `millas`. Una vez más, es responsabilidad del programador asegurar que la dirección final "apunte a" un elemento de datos válido.
5. Tomar direcciones de constantes apuntadoras. Por ejemplo, dadas las declaraciones

   ```
   int nums[25];
   int *pt;
   ```

la asignación

```
pt = &nums;
```

es inválida. La constante `nums` es una constante apuntadora que en sí misma es equivalente a una dirección. La asignación correcta es `pt = nums`.

6. Tomar direcciones de un argumento de referencia, variable de referencia o variable de registro. La razón para esto es que los argumentos y variables de referencia son en esencia lo mismo que constantes apuntadoras, en cuanto a que son valores de dirección nombrados. Del mismo modo, no puede tomarse la dirección de una variable de registro. Por tanto, para las declaraciones

   ```
   register in total;
   int *ptTot;
   ```

   la asignación

   ```
   ptTot = &total;  // INVALIDO
   ```

   es inválida. La razón es que las variables de registro se almacenan en los registros internos de la computadora, y estas áreas de almacenamiento no tienen direcciones de memoria estándar.

7. Inicializar variables apuntadoras en forma incorrecta. Por ejemplo, la inicialización

   ```
   int *pt = 5;
   ```

   es inválida. Dado que `pt` es un apuntador a un número entero, debe inicializarse con una dirección válida.

8. Confundirse respecto a si una variable *contiene* una dirección o es una dirección. Las variables apuntadoras y los argumentos apuntadores contienen direcciones. Aunque una constante apuntadora es sinónimo de una dirección, es útil tratar a las constantes apuntadoras como variables apuntadoras con dos restricciones:
   - La dirección de una constante apuntadora no puede ser toamda.
   - La dirección "contenida en" la constante apuntadora no puede ser alterada.

   Excepto por estas dos restricciones, las constantes apuntadoras y las variables apuntadoras pueden usarse en forma casi intercambiable. Por consiguiente, cuando se requiere una dirección, puede usarse cualquiera de los siguientes:
   - un nombre de variable apuntadora
   - un nombre de argumento apuntador
   - un nombre de constante apuntadora
   - un nombre de variable no apuntadora precedido por el operador de dirección (por ejemplo, `&variable`)
   - un nombre de argumento no apuntador precedido por el operador de dirección (por ejemplo, `&argumento`)

   Algo de la confusión que rodea a los apuntadores es causada por el uso indiferente de la palabra *apuntador*. Por ejemplo, el enunciado "una función requiere un argumento apuntador" se entiende con más claridad cuando uno se da cuenta que el enunciado significa en realidad "una función requiere una dirección como argumento". Del mismo modo, el enunciado "una función devuelve un apuntador" en realidad significa "una función devuelve una dirección".

Si en alguna ocasión tiene dudas respecto a lo que contiene en realidad una variable o cómo deberá ser tratada, use el objeto `cout` para desplegar el contenido de la variable, el "elemento al que apunta" o "la dirección de la variable". Poder observar lo que se despliega con frecuencia ayuda a aclarar lo que hay en realidad en la variable.

12.6 RESUMEN DEL CAPÍTULO

1. Toda variable tiene un tipo de datos, una dirección y un valor. En C++ puede obtenerse la dirección de una variable usando el operador de dirección `&`.
2. Un apuntador es una variable que se utiliza para almacenar la dirección de otra variable. Los apuntadores, como todas las variables en C++, deben declararse. El operador de indirección, `*`, se usa tanto para declarar una variable apuntadora como para tener acceso a la variable cuya dirección se almacena en un apuntador.
3. Un nombre de arreglo es una constante apuntadora. El valor de la constante apuntadora es la dirección del primer elemento en el arreglo. Por tanto, si `val` es el nombre de un arreglo, `val` y `&val[0]` pueden usarse de manera intercambiable.
4. Cualquier acceso a un elemento del arreglo usando notación de subíndice siempre puede reemplazarse usando notación de apuntador. Es decir, la notación `a[i]` siempre puede reemplazarse con la notación `*(a + i)`. Esto es cierto aun si `a` fue declarada inicialmente en forma explícita como un arreglo o como un apuntador.
5. Los arreglos pueden crearse en forma dinámica mientras se está ejecutando un programa. Por ejemplo, la secuencia de instrucciones

   ```
   cout << "Introduzca el tamanio del arreglo: ";
   cin   >> num;
   int +calif = new int[num];
   ```

   crea un nombre de arreglo `grades` del tamaño `num`. El área asignada para el nombre de arreglo puede ser destruido dinámicamente usando el operador `delete[]`. Por ejemplo, la declaración `delete[] grades;` devolvera el área asignada para el nombre de arreglo de `grades` de nuevo a la computadora.
6. Los arreglos se transmiten a las funciones como direcciones. La función llamada siempre recibe acceso directo a los elementos del arreglo declarado originalmente.
7. Cuando se transmite un arreglo unidimensional a una función, la declaración de parámetros para la función puede ser una declaración de arreglo o una declaración de apuntador. Por tanto, las siguientes declaraciones de parámetros son equivalentes:

   ```
   double a[];
   double *a;
   ```

8. Los apuntadores pueden incrementarse, disminuirse, compararse y asignarse. Los números sumados o restados de un apuntador se escalan de manera automática. El factor de escala usado es el número de bytes requeridos para almacenar el tipo de datos al que se apunta originalmente.