12 Apuntadores y arreglos dinámicos

12 Apuntadores y arreglos dinámicos

La memoria es necesaria para todas las operaciones de la razón.

BLAISE PASCAL, *Pensées*

Introducción

Un *apuntador* es una construcción que nos da más control sobre la memoria de la computadora. En este capítulo veremos cómo se usan apuntadores con arreglos y presentaremos una nueva forma de arreglo llamada *arreglo dinámico*. Los arreglos dinámicos son arreglos cuyo tamaño se determina durante la ejecución del programa, en lugar de fijarse cuando se escribe el programa.

Prerrequisitos

La sección 12.1, que cubre aspectos básicos de los apuntadores, utiliza material de los capítulos 2 al 7. Esta sección no requiere ningún material de los capítulos 8, 9, 10 u 11. Particularmente, no empleará arreglos y utilizará lo básico de las estructuras y clases.

La sección 12.2, que habla acerca de arreglos dinámicos, utiliza material de la sección 12.1, de los capítulos 2 al 7 y del capítulo 10. Esta sección no requiere ningún material de los capítulos 8, 9 u 11; en particular, sólo utilizará lo básico de las estructuras y las clases.

La sección 12.3, que habla acerca de las relaciones entre arreglos dinámicos y clases, así como de algunas propiedades dinámicas de las clases, utiliza material de las secciones 12.1 y 12.2 así como de los capítulos del 2 al 10 y de las secciones 11.1 y 11.2 del capítulo 11.

12.1 Apuntadores

No confundas el dedo que apunta con la Luna.
REFRÁN ZEN

apuntador

Un **apuntador** es la dirección en memoria de una variable. Recuerde que la memoria de una computadora se divide en posiciones de memoria numeradas (llamadas bytes), y que las variables se implementan como una sucesión de posiciones de memoria adyacentes. Recuerde también que en ocasiones el sistema C++ utiliza estas direcciones de memoria como nombres de las variables. Si una variable se implementa como tres posiciones de memoria, la dirección de la primera de esas posiciones a veces se usa como nombre para esa variable. Por ejemplo, cuando la variable se usa como argumento de llamada por referencia, es esta dirección, no el nombre del identificador de la variable, lo que se pasa a la función invocadora.

Una dirección que se utiliza para nombrar una variable de este modo (dando la dirección de memoria donde la variable inicia) se llama *apuntador* porque podemos pensar que la dirección "apunta" a la variable. La dirección "apunta" a la variable porque la identifica diciendo *dónde* está, en lugar de decir qué nombre tiene. Digamos que una variable está en la posición número 1007; podemos referirnos a ella diciendo "es la variable que está allá, en la posición 1007".

En varias ocasiones hemos empleado apuntadores. Como señalamos en el párrafo anterior, cuando una variable es un argumento de llamada por referencia en una llamada de función, la función recibe esta variable argumento en forma de un apuntador a la variable. Éstos son dos usos importantes de los apuntadores, pero el sistema C++ se encarga de ello automáticamente. En este capítulo mostraremos cómo escribir programas que manipulen apuntadores de cualquier forma que queramos, en lugar de confiar en que el sistema lo haga.

Variables de apuntador

Un apuntador se puede guardar en una variable. Sin embargo, aunque un apuntador es una dirección de memoria y una dirección de memoria es un número, no podemos guardar un apuntador en una variable de tipo `int` o `double`. Una variable que va a contener un apuntador se debe declarar como de tipo apuntador. Por ejemplo, lo que sigue declara `p` como una variable de apuntador que puede contener un apuntador que apunta a una variable de tipo `double`:

declaración de variables de apuntador

```
double *p;
```

La variable `p` puede contener apuntadores a variables de tipo `double`, pero normalmente no puede contener un apuntador a una variable de algún otro tipo, como `int` o `char`. Cada tipo de variable requiere un tipo de apuntador distinto.

En general, si queremos declarar una variable que pueda contener apuntadores a otras variables de un tipo específico, declaramos las variables de apuntador igual que declaramos una variable ordinaria de ese tipo, pero colocamos un asterisco antes del nombre de la variable. Por ejemplo, lo siguiente declara las variables `p1` y `p2` de modo que puedan contener apuntadores a variables de tipo `int`; también se declaran dos variables ordinarias `v1` y `v2` de tipo `int`:

```
int *p1, *p2, v1, v2;
```

Es necesario que haya un asterisco antes de *cada una* de las variables de apuntador. Si omitimos el segundo asterisco en la declaración anterior, `p2` no será una variable de apunta-

dor; será una variable ordinaria de tipo `int`. El asterisco es el mismo símbolo que hemos estado usando para la multiplicación, pero en este contexto tiene un significado totalmente distinto.

Declaraciones de variables de apuntador

Una variable que puede contener apuntadores a otras variables de tipo *Nombre_de_Tipo* se declara del mismo modo que una variable de tipo *Nombre_de_Tipo*, excepto que se antepone un asterisco al nombre de la variable.

Sintaxis:

Nombre_de_Tipo `*`*Nombre_de_Variable1*, `*`*Nombre_de_Variable2*, ...;

Ejemplo:

```
double * apuntador1, * apuntador2;
```

Cuando tratamos apuntadores y variables de apuntador, normalmente hablamos de *apuntar* en lugar de hablar de *direcciones*. Cuando una variable de apuntador, como `p1`, contiene la dirección de una variable, como `v1`, decimos que dicha variable *apunta a la variable* `v1` o *es un apuntador a la variable* `v1`.

Direcciones y números

Un apuntador es una dirección, una dirección es un entero, pero un apuntador no es un entero. Esto no es absurdo, ¡es abstracción! C++ insiste en que usemos un apuntador como una dirección y no como un número. Un apuntador no es un valor de tipo `int` ni de ningún otro tipo numérico. Normalmente no podemos guardar un apuntador en una variable de tipo `int`. Si lo intentamos, casi cualquier compilador de C++ generará un mensaje de error o de advertencia. Además, no podemos efectuar las operaciones aritméticas normales con apuntadores. (Podemos realizar una especie de suma y una especie de resta con apuntadores, pero no son la suma y resta normales con enteros.)

el operador &

Las variables de apuntador, como `p1` y `p2` en nuestro ejemplo de declaración, pueden contener apuntadores a variables como `v1` y `v2`. Podemos usar el operador `&` para determinar la dirección de una variable, y luego podemos asignar esa dirección a una variable de apuntador. Por ejemplo, lo siguiente asigna a la variable `p1` un apuntador que apunta a la variable `v1`:

```
p1 = &v1;
```

el operador *

desreferenciación

Ahora tenemos dos formas de referirnos a `v1`: podemos llamarla `v1` o "la variable a la que `p1` apunta". En C++ la forma de decir "la variable a la que `p1` apunta" es `*p1`. Éste es el mismo asterisco que usamos al declarar `p1`, pero ahora tiene otro significado. Cuando el asterisco se usa de esta manera se le conoce como **operador de desreferenciación**, y decimos que la variable de apuntador está **desreferenciada**.

Si armamos todas estas piezas podemos obtener algunos resultados sorprendentes. Consideremos el siguiente código:

```
v1 = 0;
p1 = &v1;
```

```
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

Este código despliega lo siguiente en la pantalla:

```
42
42
```

En tanto p1 contenga un apuntador que apunte a v1, entonces v1 y *p1 se referirán a la misma variable. Así pues, si asignamos 42 a *p1, también estamos asignando 42 a v1.

El símbolo & que usamos para obtener la dirección de una variable es el mismo símbolo que usamos en una declaración de función para especificar un parámetro de llamada por referencia. Esto no es una coincidencia. Recuerde que un argumento de llamada por referencia se implementa dando la dirección del argumento a la función invocadora. Así pues, estos dos usos del símbolo & son básicamente el mismo. Sin embargo, hay ciertas diferencias pequeñas en la forma de uso, así que los consideraremos dos usos distintos (aunque íntimamente relacionados) del símbolo &.

Los operadores * y &

El operador * antepuesto a una variable de apuntador produce la variable a la que apunta. Cuando se usa de esta forma, el operador * se llama **operador de desreferenciación**.

El operador & antepuesto a una variable ordinaria produce la dirección de esa variable; es decir, produce un apuntador que apunta a la variable. El operador & se llama simplemente **operador de dirección de**.

Por ejemplo, consideremos las declaraciones

```
double *p, v;
```

Lo siguiente establece a p de modo que apunte a la variable v:

```
p= &v;
```

*p produce la variable a la que apunta p, así que después de la asignación anterior *p y v se refieren a la misma variable. Por ejemplo, lo siguiente establece el valor de v a 9.99, aunque nunca se usa explícitamente el nombre v:

```
*p = 9.99;
```

apuntadores en instrucciones de asignación

Podemos asignar el valor de una variable de apuntador a otra variable de apuntador. Esto copia una dirección de una variable de apuntador a otra. Por ejemplo, si p1 todavía está apuntando a v1, lo que siguiente establecerá el valor de p2 de modo que también apunte a v1:

```
p2 = p1;
```

Siempre que no hayamos modificado el valor de v1, lo siguiente también desplegará 42 en la pantalla:

```
cout << *p2;
```

Asegúrese de no confundir

```
p1 = p2;
```

con

```
*p1 = *p2;
```

Cuando añadimos el asterisco, no estamos tratando con los de apuntadores `p1` y `p2`, sino con las variables a las que estos apuntan. Esto se ilustra en el cuadro 12.1.

Puesto que podemos usar un de apuntador para referirnos a una variable, el programa puede manipular variables aunque éstas carezcan de identificadores que las nombren. Podemos usar el operador *new* para crear variables sin identificadores que sean sus nombres. Hacemos referencia a estas variables sin nombre mediante apuntadores. Por ejemplo, lo siguiente crea una nueva variable de tipo *int* y asigna a la variable de apuntador `p1` la dirección de esta nueva variable (es decir, `p1` apunta a esta nueva variable sin nombre):

new

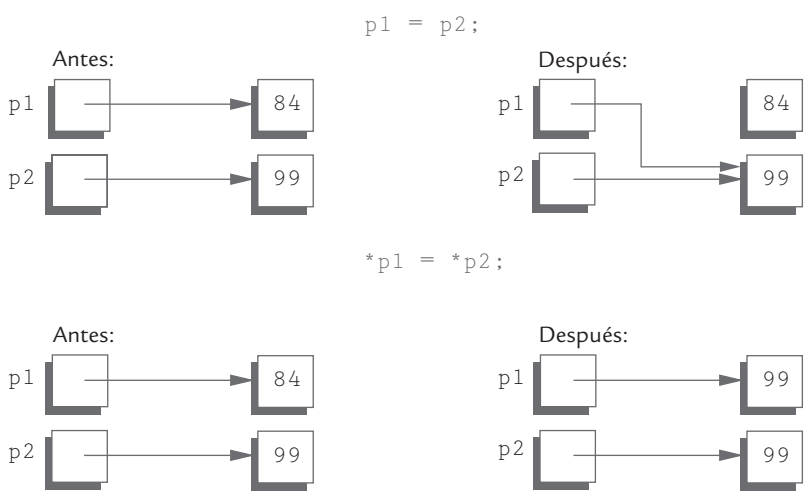
```
p1 = new int;
```

Podemos usar `*p1` para referirnos a esta nueva variable (es decir, la variable a la que `p1` apunta). Podemos hacer con esta variable sin nombre lo mismo que con cualquier otra variable de tipo *int*. Por ejemplo, lo que sigue lee un valor de tipo *int* del teclado y lo coloca en la variable sin nombre, suma `7` al valor y luego despliega el nuevo valor en la pantalla:

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

El operador *new* produce una nueva variable sin nombre y devuelve un apuntador que apunta a esta nueva variable. Especificamos el tipo de la nueva variable escribiendo el nombre

CUADRO 12.1 *Usos del operador de asignación*



variables dinámicas

del tipo después del operador *new*. Las variables que se crean con el operador *new* se llaman **variables dinámicas** porque se crean y se destruyen mientras el programa se está ejecutando. El programa del cuadro 12.2 demuestra algunas operaciones sencillas con apuntadores y variables dinámicas. El cuadro 12.3 ilustra el funcionamiento del programa del cuadro 12.2. En el cuadro 12.3, las variables se representan como cuadritos y el valor de la variable se escribe dentro del cuadrito. No hemos mostrado las direcciones numéricas reales en las variables de apuntadores. Los números reales no son importantes. Lo que es importante es que el número es la dirección de una variable dada. Así pues, en lugar de usar el número real de la dirección, sólo hemos indicado la dirección con una flecha que apunta a la variable que tiene esa dirección. Por ejemplo, en la ilustración (b) del cuadro 12.3, `p1` contiene la dirección de una variable en la que se escribió un signo de interrogación.

Uso de variables de apuntador con =

Si `p1` y `p2` son variables de apuntador, la instrucción

```
p1 = p2;
```

modificará `p1` de modo que apunte a lo mismo que `p2`.

El operador *new*

El operador *new* crea una nueva variable dinámica del tipo que se especifica y devuelve un apuntador que apunta a esta nueva variable. Por ejemplo, lo que sigue crea una variable dinámica nueva del tipo `MiTipo` y deja a la variable del apuntador `p` apuntando a esa nueva variable.

```
MiTipo *p;
p = new MiTipo;
```

Si el tipo es una clase que tiene un constructor, se invoca el constructor predeterminado para la variable dinámica recién creada. Se pueden especificar inicializadores que hagan que se invoquen otros constructores:

```
int *n;
n = new int(17);  //inicializa n con 17
MiTipo *apuntMt;
apuntMt = new MiTipo(32.0, 17);  // invoca MiTipo(double, int);
```

El estándar de C++ estipula que si no hay suficiente memoria desocupada para crear la nueva variable, la acción predeterminada del operador *new* es terminar el programa.[1]

[1]Técnicamente, el operador *new* lanza una excepción que, si no se atrapa, termina el programa. Es posible "atrapar" la excepción o instalar un `controlador new`, pero estos temas rebasan el alcance de este libro.

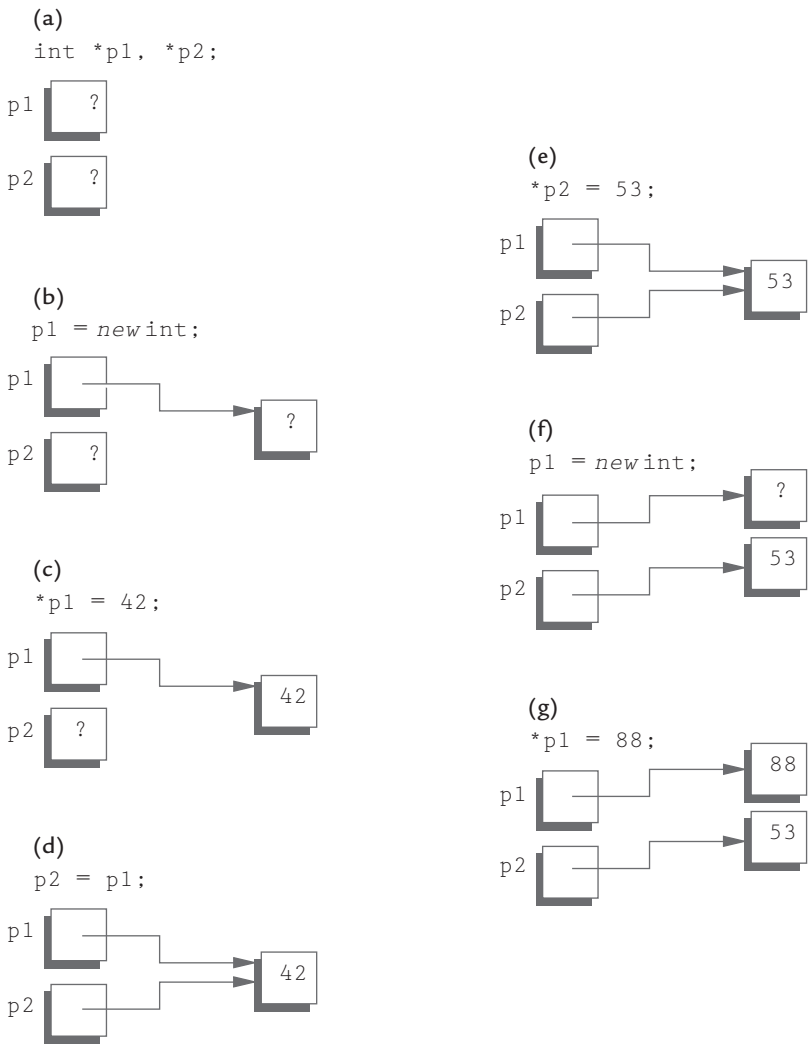
CUADRO 12.2 *Manipulaciones básicas de apuntadores*

```
//Programa para demostrar apuntadores y variables dinámicas.
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Ojala hayas entendido este ejemplo!\n";
    return 0;
}
```

Diálogo de ejemplo

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Ojala hayas entendido este ejemplo!
```

CUADRO 12.3 *Explicación del cuadro 12.2*



Ejercicios de AUTOEVALUACIÓN

1. Explique el concepto de apuntador en C++.

2. ¿Qué interpretación errónea puede ocurrir con la siguiente declaración?

```
int* apunt_int1, apunt_int2;
```

3. Mencione al menos dos usos del operador *. Indique lo que está haciendo el *, y nombre el uso de * que está presentando.

4. ¿Qué salidas produce el siguiente código?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

¿Cómo cambiarían las salidas si se sustituyera

```
*p1 = 30;
```

con lo siguiente?

```
*p2 = 30;
```

5. ¿Qué salidas produce el siguiente código?

```
int *p1 = *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; //Esto es diferente del ejercicio 4.
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Administración de memoria básica

Se reserva un área especial de la memoria, llamada **almacén libre**[2] para las variables dinámicas. Toda variable dinámica nueva creada por un programa consume parte de la memo- **almacén libre**

[2]El almacén libre también se conoce como montículo (*heap*).

ria del almacén libre. Si nuestro programa crea demasiadas variables dinámicas, consumirá toda la memoria del almacén libre. Si esto sucede, todas las llamadas subsecuentes a `new` fracasarán.

El tamaño del almacén libre varía de una implementación de C++ a otra. Por lo regular es grande y es poco probable que un programa modesto use toda la memoria del almacén libre. No obstante, incluso en los programas modestos es recomendable reciclar la memoria del almacén libre que ya no se necesite. Si nuestro programa ya no necesita una variable dinámica dada, la memoria ocupada por esa variable puede ser reciclada.

delete

El operador `delete` elimina una variable dinámica y devuelve al almacén libre la memoria que ocupaba, para poder reutilizarla. Supongamos que `p` es una variable de apuntador que está apuntando a una variable dinámica. Lo que sigue destruye la variable dinámica a la que `p` apunta y devuelve al almacén libre la memoria ocupada por esa variable dinámica:

```
delete p;
```

Después de la llamada a `delete`, el valor de `p` es indefinido y `p` será tratada como una variable no inicializada.

El operador *delete*

El operador `delete` elimina una variable dinámica y devuelve al almacén libre la memoria que esa variable ocupaba. Esa memoria puede entonces reutilizarse para crear nuevas variables dinámicas. Por ejemplo, lo que sigue elimina la variable dinámica a la que apunta la variable de apuntador `p`:

```
delete p;
```

Después de una llamada a `delete`, el valor de la variable de apuntador (`p` en nuestro ejemplo) no está definido. (Se usa una versión un poco distinta de `delete`, la cual veremos más adelante, cuando la variable dinámica es un arreglo.)

RIESGO *Apuntadores colgantes*

Apuntadores colgantes

Cuando aplicamos `delete` a una variable de apuntador, la variable dinámica a la que apunta se destruye. En ese momento, el valor de la variable de apuntador queda indefinido, lo que significa que no sabemos a dónde está apuntando ni qué valor hay en el lugar al que está apuntando. Es más, si alguna otra variable de apuntador estaba apuntando a la variable dinámica que se destruyó, esa otra variable de apuntador también quedará indefinida. Estas variables de apuntador no definidas se llaman **apuntadores colgantes**. Si `p` es un apuntador colgante y el programa aplica el operador de desreferenciación `*` a `p` (para producir la expresión `*p`), el resultado es impredecible y por lo regular desastroso. Antes de aplicar el operador de desreferenciación a una variable de apuntador debe asegurarse de que ésta apunta a alguna variable.

Variables estáticas, dinámicas y automáticas

variables dinámicas

Las variables que se crean con el operador *new* se llaman **variables dinámicas** porque se crean y destruyen mientras el programa se está ejecutando. En comparación con estas variables dinámicas, las variables ordinarias parecen estáticas, pero la terminología que usan los programadores en C++ es un poco más complicada, y las variables ordinarias no se denominan *variables estáticas*.

Las variables ordinarias que hemos estado usando en capítulos anteriores no son realmente estáticas. Si una variable es local respecto a una función, el sistema C++ crea la variable cuando se invoca la función, y la destruye cuando se completa la llamada. Puesto que la parte `main` del programa en realidad no es más que una función llamada `main`, esto se cumple incluso para las variables declaradas en la parte `main`. (Puesto que la llamada a `main` no termina hasta que el programa termina, las variables declaradas en `main` no se destruyen hasta que el programa finaliza, pero el mecanismo para manejar variables locales es el mismo para `main` que para cualquier otra función.) Las variables ordinarias que hemos estado usando (es decir, las variables declaradas dentro de `main` o dentro de alguna otra definición de función) se llaman **variables automáticas** porque sus propiedades dinámicas se controlan automáticamente; se crean de manera automática cuando se invoca la función en la que se declararon y se destruyen de igual forma cuando termina la llamada a la función. Por lo regular llamaremos a estas variables **variables ordinarias**, pero otros libros las llaman *variables automáticas*.

variables automáticas

Existe otra catégoria de variables llamada **variables globales**. Éstas son variables que se declaran fuera de cualquier definición de función (incluso fuera de `main`). Tratamos a las variables globales brevemente en el capítulo 3. Da la casualidad que no necesitamos variables globales y por ello no las hemos usado.

variables globales

TIP DE PROGRAMACIÓN

Defina tipos de apuntadores

typedef

Podemos definir un nombre de tipo de apuntador para poder declarar variables de apuntador igual que otras variables, sin tener que colocar un asterisco antes de cada variable de apuntador. Por ejemplo, lo que sigue define un tipo llamado `ApuntInt`, que es el tipo para variables de apuntador que contienen apuntadores a variables *int*:

```
typedef int* ApuntInt;
```

Así pues, estas dos declaraciones de variable de apuntador son equivalentes:

```
ApuntInt p;
```

e

```
int *p;
```

Podemos usar *typedef* para definir un alias para cualquier nombre o definición de tipo. Por ejemplo, lo que se muestra a continuación define el nombre de tipo Kilometros de modo que signifique lo mismo que el nombre de tipo *double*:

```
typedef double Kilometros;
```

Una vez dada esta definición de tipo, podemos definir una variable de tipo *double* así:

```
Kilometros distancia;
```

En ocasiones, este renombramiento de tipos existentes puede ser útil. Sin embargo, nuestro principal uso de *typedef* será en la definición de tipos para variables de apuntador.

El uso de nombres de tipo de apuntador definidos, como ApuntInt, tiene dos ventajas. Primero, evita el error de omitir un asterisco. Recuerde, si queremos que p1 y p2 sean apuntadores, lo siguiente es un error:

```
int *p1, p2;
```

Puesto que omitimos el * de p2, la variable p2 es una variable *int* ordinaria, no una variable de apuntador. Si nos confundimos y colocamos el * en *int*, el problema es el mismo pero es más difícil de detectar. C++ nos permite colocar el * junto al nombre del tipo, como *int*, de modo que lo siguiente es válido:

```
int* p1, p2;
```

Aunque lo anterior es válido, es engañoso. Pareciera que tanto p1 como p2 son variables de apuntador, pero en realidad sólo p1 lo es; p2 es una variable *int* ordinaria. En lo que al compilador de C++ concierne, el * que se anexó a la palabra *int* igualmente podría haberse anexado al identificador p1. Una forma correcta de declarar tanto p1 como p2 como variables de apuntador es

```
int *p1, *p2;
```

Una forma más fácil y menos propensa a errores para declarar tanto p1 como p2 como variables de apuntador es usar el nombre de tipo definido ApuntInt de esta manera:

```
ApuntInt p1, p2;
```

La segunda ventaja de usar un tipo de apuntador definido como ApuntInt se hace evidente cuando definimos una función con un parámetro de llamada por referencia para una variable de apuntador. Sin el nombre de tipo de apuntador definido, necesitaríamos incluir tanto un * como un & en la declaración de la función, y los detalles pueden dar pie a confusión. Si usamos un nombre de tipo para el tipo de apuntador, el uso de un parámetro de llamada por referencia para dicho tipo no tendrá complicaciones. Definiremos un parámetro de llamada por referencia para un tipo de apuntador definido de la misma manera que definimos cualquier otro parámetro de llamada por referencia. He aquí un ejemplo:

```
void funcion_muestra(ApuntInt& variable_apuntador);
```

Definiciones de tipos

Podemos asignar un nombre a una definición de tipo y luego usar el nombre de tipo para declarar variables. Esto se hace con la palabra clave *typedef*. Estas definiciones de tipos normalmente se colocan afuera del cuerpo de la parte `main` del programa (y fuera del cuerpo de otras funciones) en el mismo lugar que las definiciones de `struct` y clases. Usaremos definiciones de tipos para definir nombres de tipos de apuntadores, como se muestra en el siguiente ejemplo:

Sintaxis:

typedef Definicion_de_Tipo_Conocido Nuevo_Nombre_de_Tipo;

Ejemplo:

```
typedef int* ApuntInt;
```

El nombre de tipo `ApuntInt` se puede usar entonces para declarar apuntadores a variables dinámicas de tipo *int*, como en:

```
ApuntInt apuntador1, apuntador2;
```

Ejercicios de AUTOEVALUACIÓN

6. Suponga que se crea una variable dinámica así:

   ```
   char *p;
   p = new char;
   ```

   Suponiendo que el valor de `p` no haya cambiado (o sea, que todavía apunte a la misma variable dinámica), ¿cómo podemos destruir esta nueva variable dinámica y devolver al almacén libre la memoria que ocupa, a fin de poder reutilizarla en la creación de otras variables dinámicas nuevas?

7. Escriba una definición de un tipo llamado `ApuntNumero` que sea el tipo para variables de apuntador que contengan apuntadores a variables dinámicas de tipo *int*. También, escriba una declaración de una variable de apuntador llamada `mi_apunt`, que sea de tipo `ApuntNumero`.

8. Describa la acción del operador *new*. ¿Qué devuelve el operador *new*?

12.2 Arreglos dinámicos

En esta sección veremos que las variables de arreglo son en realidad variables de apuntador. También aprenderemos a escribir programas con arreglos dinámicos. Un **arreglo dinámico** es un arreglo cuyo tamaño no se especifica al momento de escribir el programa, sino durante la ejecución de éste.

arreglo dinámico

Variables de arreglo y variables de apuntador

En el capítulo 10 describimos la forma en que los arreglos se guardan en la memoria. Entonces no habíamos visto los apuntadores, así que tratamos los arreglos en términos de direcciones de memoria. Sin embargo, una dirección de memoria es un apuntador, así que en C++ una variable de arreglo en realidad es una variable de apuntador que apunta a la primera variable indizada del arreglo. Dadas las dos declaraciones siguientes, `p` y `a` son variables de la misma especie:

```
int a[10];
typedef int* ApuntInt;
ApuntInt p;
```

El hecho de que `a` y `p` sean variables de la misma especie se ilustra en el cuadro 12.4. Puesto que `a` es un apuntador que apunta a una variable de tipo *`int`* (a saber, la variable `a[0]`), podemos asignar el valor de `a` a la variable de apuntador `p` así:

```
p = a;
```

Después de esta asignación, `p` apunta a la misma posición de memoria que `a`. Así pues, `p[0]`, `p[1]`, ... `p[9]` se refieren a las variables indizadas `a[0]`, `a[1]`, ... `a[9]`. La notación de corchetes que hemos estado usando con arreglos aplica a las variables de apuntador en tanto éstas apunten a un arreglo en memoria. Después de la asignación anterior, podemos tratar el identificador `p` como si fuera un identificador de arreglo. También podemos tratar al identificador `a` como si fuera una variable de apuntador, con una importante diferencia. *No podemos cambiar el valor de apuntador de una variable de arreglo, como* `a`. Usted podría pensar que lo siguiente es válido, pero no es así:

```
ApuntInt p2;
...//p2 es dado por algún valor de apuntador
a = p2; //NO VÁLIDO. No podemos asignar una dirección distinta a a.
```

Cómo crear y usar arreglos dinámicos

Un problema que presentan los arreglos que hemos usado hasta ahora es que es preciso especificar el tamaño del arreglo en el momento de escribir el programa, y podría ser imposible saber de qué tamaño necesitamos un arreglo antes de ejecutar el programa. Por ejemplo, un arreglo podría contener una lista de números de identificación de estudiantes, pero el tamaño del grupo podría ser distinto cada vez que se ejecuta el programa. Con los tipos de arreglos que hemos usado hasta ahora, tendríamos que estimar el tamaño más grande que podríamos llegar a necesitar para el arreglo, y rezar porque sea suficiente. Esto implica dos problemas. Primero, podríamos hacer un estimado demasiado bajo, y entonces nuestro programa no funcionaría en todas las situaciones. Segundo, dado que el arreglo podría tener muchas posiciones desocupadas, se podría desperdiciar memoria de la computadora. Los arreglos dinámicos evitan estos problemas. Si nuestro programa usa un arreglo dinámico para los números de identificación de los estudiantes, podremos introducir el tamaño del grupo como entrada del programa y crear un arreglo dinámico exactamente de ese tamaño.

creación de un arreglo dinámico

Los arreglos dinámicos se crean con el operador *`new`*. La creación y uso de arreglos dinámicos son sorprendentemente sencillos. Puesto que las variables de arreglo son variables

CUADRO 12.4 *Arreglos y variables de apuntador*

```
//Programa para demostrar que una variable arreglo es una especie de variable
//de apuntador.
#include <iostream>
using namespace std;

typedef int* ApuntInt;

int main()
{
    ApuntInt p;
    int a[10];
    int indice;

    for (indice = 0; indice < 10; indice++)
        a[indice] = indice;
    p = a;

    for (indice = 0; indice < 10; indice++)
        cout << p[indice] << "";
    cout << endl;

    for (indice = 0; indice < 10; indice++)
        p[indice] = p[indice] + 1;

    for (indice = 0; indice < 10; indice++)
        cout << a[indice] << " ";
    cout << endl;

    return 0;
}
```

Observe que los cambios al arreglo `p` *también son cambios al arreglo* `a`.

Salida

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

de apuntador, podemos usar el operador *new* para crear variables dinámicas que sean arreglos y tratarlas como si fueran arreglos ordinarios. Por ejemplo, lo siguiente crea una variable arreglo dinámica con 10 elementos de tipo *double*.

```
typedef double* ApuntDouble;
ApuntDouble p;
p = new double[10];
```

Si queremos crear un arreglo dinámico de elementos de cualquier otro tipo, simplemente sustituimos *double* por el tipo deseado. En particular, podemos sustituir el tipo *double* por un tipo *struct* o de clase. Si queremos obtener una variable de arreglo dinámico de cualquier otro tamaño, basta con sustituir 10 por el tamaño deseado.

Este ejemplo tiene varios detalles menos obvios que es importante notar. En primer lugar, el tipo que usamos para un apuntador a un arreglo dinámico es el mismo que el tipo de apuntador que usaríamos para un solo elemento del arreglo. Por ejemplo, el tipo de apuntador para un arreglo de elementos de tipo *double* es el mismo que usaríamos para una variable simple de tipo *double*. El apuntador al arreglo en realidad es un apuntador a la primera variable indizada del arreglo. En el ejemplo anterior se crea un arreglo con 10 variables indizadas, y el apuntador p apunta a la primera de esas 10 variables indizadas.

Cómo usar un arreglo dinámico

- *Defina un tipo de apuntador:* Defina un tipo para apuntadores a variables del mismo tipo que los elementos del arreglo. Por ejemplo, si el arreglo dinámico es un arreglo de *double*, podría usar lo siguiente:

  ```
  typedef double* ApuntArregloDouble;
  ```

- *Declare una variable de apuntador:* Declare una variable de apuntador de este tipo definido. Dicha variable apuntará al arreglo dinámico en la memoria y servirá como nombre del arreglo dinámico.

  ```
  ApuntArregloDouble a;
  ```

- *Llame a new:* Cree un arreglo dinámico usando el operador *new*:

  ```
  a = new double[tamanio_arreglo];
  ```

  El tamaño del arreglo dinámico se da entre corchetes como en el ejemplo anterior. Se puede dar el tamaño usando una variable *int* o alguna otra expresión *int*. En el ejemplo anterior, tamanio_arreglo puede ser una variable de tipo *int* cuyo valor se determina mientras el programa se está ejecutando.

- *Utilice como un arreglo ordinario:* La variable de apuntador, digamos a, se usa igual que un arreglo ordinario. Por ejemplo, las variables indizadas se escriben de la forma acostumbrada, a[0], a[1], etcétera. No debemos asignar a dicha variable ningún otro valor de apuntador; debemos usarla como una variable de arreglo.

- *Llame a delete []:* Cuando el programa haya terminado de usar la variable dinámica, use *delete* y unos corchetes vacíos junto con la variable de apuntador para eliminar el arreglo dinámico y devolver al almacén libre la memoria que ocupa, a fin de reutilizarla. Por ejemplo,

  ```
  delete [] a;
  ```

Observe también que, al usar *new*, damos el tamaño del arreglo dinámico en corchetes después del tipo, que en este ejemplo es *double*. Esto le dice a la computadora cuánto espacio debe reservar para el arreglo dinámico. Si omitimos los corchetes y el 10, la computadora sólo asignará suficiente memoria para una variable de tipo *double*, no para un arreglo de 10 variables indizadas de tipo *double*. Como se ilustra en el cuadro 12.5, puede utilizar una variable de tipo *int* en lugar de una constante 10 de manera que el tamaño del arreglo pueda ser leído dentro del programa.

El cuadro 12.5 contiene un programa que ordena una lista de números. Este programa funciona con listas de cualquier tamaño porque usa un arreglo dinámico para contener los números. El tamaño del arreglo se determina cuando se ejecuta el programa. Se pregunta al usuario cuántos números habrá y entonces el operador *new* crea un arreglo dinámico de ese tamaño. El tamaño del arreglo dinámico está dado por la variable tam_arreglo.

delete []

Observe la instrucción *delete* que destruye la variable del arreglo dinámico a en el cuadro 12.5. Puesto que el programa ya estaba a punto de terminar, en realidad no era necesario incluir esta instrucción *delete*; pero si el programa fuera a hacer otras cosas con variables dinámicas es recomendable incluir esa instrucción para que la memoria ocupada por el arreglo dinámico se devuelva al almacén libre. La instrucción *delete* para un arreglo dinámico es similar a la que vimos antes, excepto que en este caso es preciso incluir un par de corchetes, así:

```
delete [] a;
```

Los corchetes le dicen a C++ que se está eliminando una variable de arreglo dinámico, así que el sistema verifica el tamaño del arreglo y elimina ese número de variables indizadas. Si omitimos los corchetes le estaríamos diciendo a la computadora que sólo eliminara una variable de tipo *int*. Por ejemplo,

```
delete a;
```

no es válido, pero ningún compilador que el autor conozca detecta el error. El Estándar ANSI C++ dice que lo que sucede cuando se hace esto "no está definido". Eso significa que el autor del compilador puede decidir que se haga lo que crea más conveniente (para el autor del compilador, no para nosotros). Aunque se haga algo útil, no hay garantía de que la siguiente versión de ese compilador, o cualquier otro compilador que usemos para compilar este código, hará la misma cosa. Moraleja: siempre use la sintaxis

```
delete [] apuntArreglo;
```

al eliminar memoria que se asignó con, por ejemplo,

```
apuntArreglo = new MiTipo[37];
```

Creamos un arreglo dinámico con una llamada a *new* usando un apuntador, como el apuntador a del cuadro 12.5. Después de la llamada a *new*, no deberemos asignar ningún otro valor de apuntador a esta veriable apuntador, pues ello podría confundir al sistema cuando la memoria del arreglo dinámico se devuelva al almacén libre con una llamada a *delete*.

Los arreglos dinámicos se crean usando *new* y una variable de apuntador. Cuando nuestro programa termine de usar un arreglo dinámico, es muy recomendable devolver al almacén libre la memoria que ocupaba, con una llamada a *delete*. Por lo demás, un arreglo dinámico se puede usar como cualquier otro arreglo.

CUADRO 12.5 *Un arreglo dinámico (parte 1 de 2)*

```
//Ordena una lista de números introducidos con el teclado.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* ApuntArregloInt;

void llenar_arreglo(int a[], int tamanio);
//Precondición: tamanio es el tamaño del arreglo a.
//Postcondición: a[0] hasta a[tamanio-1] se han
//llenado con valores leídos del teclado.

void ordenar(int a[], int tamanio);
//Precondición: tamanio es el tamaño del arreglo a.
//Los elementos a[0] hasta a[tamanio-1] tienen valores.
//Postcondición: Los valores de a[0] a a[tamanio-1] se reacomodaron
//de modo que a[0] <= a[1] <= ... <= a[tamanio-1].

int main()
{
    using namespace std;
    cout << "Este programa ordena numeros de menor a mayor.\n";

    int tamanio_arreglo;
    cout << "Cuantos numeros se ordenaran? ";
    cin >> tamanio_arreglo;

    ApuntArregloInt a;
    a = new int[tamanio_arreglo];

    llenar_arreglo(a, tamanio_arreglo);
    ordenar(a, tamanio_arreglo);
```

Parámetros de arreglos ordinarios.

CUADRO 12.5 *Un arreglo dinámico (parte 2 de 2)*

```
    cout << "De menor a mayor los numeros son:\n";
    for (int indice = 0; indice < tamanio_arreglo; indice++)
        cout << a[indice] << " ";
    cout << endl;

    delete [] a;

    return 0;
}

//Usa la biblioteca iostream:
void llenar_arreglo(int a[], int tamanio)
{
    using namespace std;
    cout << "Escriba " << tamanio << " enteros.\n";
    for (int indice = 0; indice < tamanio; indice++)
        cin >> a[indice];
}

void ordenar(int a[], int tamanio)
```

El arreglo dinámico a *se usa como un arreglo ordinario.*

<Se puede usar cualquier implementación de `ordenar`. Esto podría requerir la definición de funciones adicionales. La implementación ni siquiera necesita saber que `ordenar` se invocará para un arreglo dinámico. Por ejemplo, podemos usar la implementación del cuadro 10.12 (con ajustes apropiados a los nombres de los parámetros).>

Ejercicios de AUTOEVALUACIÓN

9. Escriba una definición de tipo para variables de apuntador que se usarán para apuntar a arreglos dinámicos. Los elementos de los arreglos serán de tipo `char`. Llame al tipo `ArregloChar`.

10. Suponga que su programa contiene código para crear un arreglo dinámico de esta manera:

```
int *dato;
dato = new int[10];
```

de modo que la variable de apuntador `dato` apunte a este arreglo dinámico. Escriba código para llenar este arreglo con 10 números que se introducen desde el teclado.

11. Suponga que su programa contiene código para crear un arreglo dinámico como en el ejercicio anterior, y que no se ha modificado el valor (de apuntador) de la variable de apuntador `dato`. Escriba código que destruya este nuevo arreglo dinámico y devuelva al almacén libre la memoria que ocupa.

12. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int a[10];
int *p = a;
int i;
for (i = 0; i < 10; i++)
   a[i] = i;
for (i = 0; i < 10; i++)
   cout << p[i] << " ";
cout << endl;
```

13. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int *p = a;
int i;
for (i = 0; i < tamanio_arreglo; i++)
   a[i] = i;
p[0] = 10;
for (i = 0; i < tamanio_arreglo; i++)
   cout << a[i] << " ";
cout << endl;
```

Aritmética de apuntadores (*Opcional*)

Existe un tipo de aritmética que podemos efectuar con apuntadores, pero es una aritmética de direcciones, no de números. Por ejemplo, supongamos que nuestro programa contiene el siguiente código:

```
typedef double* ApuntDouble;
ApuntDouble d;
d = new double[10];
```

direcciones, no números

Después de estas instrucciones, `d` contiene la dirección de la variable indizada `d[0]`. La evaluación de la expresión `d + 1` da la dirección de `d[1]`, `d + 2` es la dirección de `d[2]`, etcétera. Observe que aunque el valor de `d` es una dirección y una dirección es un número, `d + 1` no se limita a sumar `1` al número que está en `d`. Si una variable de tipo *double* requiere ocho bytes (ocho posiciones de memoria) y `d` contiene la dirección `2001`, entonces la evaluación de `d + 1` da la dirección de memoria `2009`. Desde luego, podemos sustituir el tipo *double* por cualquier otro tipo, y entonces la suma de apuntadores se efectuará en unidades de variables de ese tipo.

Esta aritmética de apuntadores nos ofrece otra forma de manipular arreglos. Por ejemplo, si `tamanio_arreglo` es el tamaño del arreglo dinámico al que apunta `d`, lo que sigue desplegará en la pantalla el contenido del arreglo dinámico:

```
for (int i = 0; i < tamanio_arreglo; i++)
    cout << *(d + i) << " ";
```

Lo anterior equivale a:

```
for (int i = 0; i < tamanio_arreglo; i++)
    cout << d[i] << " ";
```

No se permite efectuar multiplicación o división con apuntadores. Lo único que puede hacerse es sumar un entero a un apuntador, restar un entero a un apuntador, o restar dos apuntadores del mismo tipo. Cuando restamos dos apuntadores el resultado es el número de variables indizadas que hay entre dos direcciones. Recuerde, para poder restar dos valores de apuntadores los valores deben apuntar al mismo arreglo. No tiene sentido restar un apuntador que apunta a un arreglo a otro apuntador que apunta a un arreglo distinto. Podemos usar los operadores de incremento y decremento ++ y −−. Por ejemplo, `d++` aumenta el valor de `d` de modo que contenga la dirección de la siguiente variable indizada, y `d−−` modifica `d` de modo que contenga la dirección de la variable indizada anterior.

++ y −−

Ejercicios de AUTOEVALUACIÓN

Estos ejercicios aplican para la sección opcional aritmética de apuntadores.

14. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int i;
for (i = 0; i < tamanio_arreglo; i++)
   *(a + i) = i;
for (i = 0; i < tamanio_arreglo; i++)
   cout << a[i] << " ";
cout << endl;
```

15. ¿Qué salida producirá el siguiente código incrustado en un programa completo?

```
int tamanio_arreglo = 10;
int *a;
a = new int[tamanio_arreglo];
int i;
for (i = 0; i < tamanio_arreglo; i++)
   a[i] = i;
while (*a < 9)
{
    a++;
    cout << *a << " ";
}
cout << endl;
```

Arreglos dinámicos multidimensionales (*Opcional*)

Podemos tener arreglos dinámicos multidimensionales. Sólo recuerde que los arreglos multidimensionales son arreglos de arreglos, o arreglos de arreglos de arreglos, etcétera. Por ejemplo, para crear un arreglo dinámico bidimensional, debemos recordar que éste es un arreglo de arreglos. Para crear un arreglo bidimensional de enteros, debemos crear un arreglo dinámico unidimensional de apuntadores de tipo `int*`, el cual es el tipo para los arreglos unidimensionales de tipo `int`. Después deberemos crear un arreglo dinámico de `int`s para cada variable indizada del arreglo de apuntador.

Una definición de tipo puede ayudar a mantener las cosas en orden. La siguiente variable de tipo es un arreglo dinámico unidimensional ordinario de `int`s:

```
typedef int* ApuntArregloInt;
```

Para obtener un arreglo de enteros de 3-por-4, necesitará un arreglo cuyo tipo base sea `ApuntArregloInt`. Por ejemplo:

```
ApuntArregloInt *m = new ApuntArregloInt[3];
```

Éste es un arreglo de tres apuntadores, cada uno de los cuales puede nombrar a un arreglo dinámico de `int`s, como sigue:

```
for (int i = 0; i < 3; i++)
m[i] = new int[4];
```

El arreglo resultante `m` es un arreglo dinámico de 3 por 4. Un programa simple para ilustrar esto se da en el cuadro 12.6.

`delete[]`

Asegúrese de observar el uso de `delete` en el cuadro 12.6. Dado que el arreglo dinámico m es un arreglo de arreglos, cada uno de los arreglos creados con new `int` en el ciclo `for` debe ser devuelto al controlador del almacén libre con una llamada a `delete[]`; entonces el arreglo `m` debe ser devuelto al almacén libre con otra llamada `delete[]` debe haber una llamada a `delete[]` por cada llamada a new que creó un arreglo. (Dado que el programa termina justo después de las llamadas a `delete[]` podríamos omitir estas llamadas pero queríamos demostrar su uso.)

12.3 Clases y arreglos dinámicos

Con todos los adminículos y recursos de sobra.

William Shakespeare, Rey *Enrique IV, parte III*

Un arreglo dinámico puede tener un tipo base que sea una clase. Una clase puede tener una variable miembro que sea un arreglo dinámico. Podemos combinar las técnicas que aprendimos para las clases y las que aprendimos para los arreglos dinámicos de prácticamente cualquier manera. Hay unas cuantas cosas más de las cuales preocuparse cuando se usan clases y arreglos dinámicos, pero las técnicas básicas son las que ya hemos usado. Comencemos con un ejemplo.