CAPÍTULO 13
Estructuras

TEMAS

Una estructura es un vestigio histórico de C. Desde la perspectiva del programador, una estructura puede considerarse como una clase que tiene todas las variables de instancia pública y no tiene métodos miembros. En las aplicaciones comerciales a la estructura se le denomina como, y es lo mismo que un registro. En C y C++, una estructura proporciona un medio para almacenar valores que tienen diferentes tipos de datos, como un número de parte entero, un tipo parte en carácter y un suministro de voltaje en número de precisión doble.

Por ejemplo, supóngase que un fabricante de circuitos integrados (CI) mantiene un resumen de información para cada uno de los circuitos que fabrica. Los elementos de datos conservados para cada circuito se ilustran en la figura 13.1.

Número de parte:
Familia de circuitos integrados:
Tipo de función:
Suministro de voltaje:
Unidades en existencia:

Figura 13.1 Un registro de inventario.

Cada uno de los elementos de datos individuales enlistados en la figura 13.1 es una entidad en sí mismo que se conoce como un ***campo de datos****. En conjunto, todos los campos de datos forman una sola unidad que se denomina* ***estructura****.*

Aunque el fabricante de circuitos integrados podría dar seguimiento a cientos de componentes, la forma de cada estructura de carácter es idéntica. Al tratar con estructuras es importante distinguir entre la forma de una estructura y su contenido.

La forma de una estructura consiste en los nombres simbólicos, tipos de datos y el orden de los campos de datos individuales en la estructura. El contenido de la estructura se refiere a los datos reales almacenados en los nombres simbólicos. La figura 13.2 muestra contenidos aceptables para la forma de estructura ilustrada en la figura 13.1.

Número de parte: 23421
Familia de circuitos integrados: TTL
Tipo de función: AND
Suministro de voltaje: 6.0
Unidades en existencia: 345

Figura 13.2 La forma y contenido de un registro.

En este capítulo se describen las instrucciones de C++ que se requieren para crear, llenar, usar y transmitir estructuras entre funciones.

13.1 ESTRUCTURAS SENCILLAS

Crear y usar una estructura requiere los mismos dos pasos necesarios para crear y usar cualquier variable. Primero debe declararse la estructura del registro. Luego pueden asignarse valores específicos a los elementos individuales de la estructura. Declarar una estructura requiere enlistar los tipos de datos, los nombres de datos y el orden de los elementos de datos. Por ejemplo, la definición

```
struct
{
  int mes;
  int dia;
  int anio;
} nacimiento;
```

da la forma de una estructura llamada `nacimiento` y reserva almacenamiento para los elementos de datos individuales enlistados en la estructura. La estructura `nacimiento` consiste en tres elementos de datos o campos, los cuales se denominan miembros de la estructura.

Asignar valores de datos reales a los elementos de datos de una estructura se conoce como *poblar la estructura* y es un procedimiento relativamente sencillo. Se tienen acceso a cada miembro de una estructura dando tanto el nombre de la estructura como el nombre del elemento de datos individual, separado por un punto. Por tanto, `nacimiento-.mes` se refiere al primer miembro de la estructura `nacimiento, nacimiento.dia` se refiere al segundo miembro de la estructura y `nacimiento.anio` se refiere al tercer miembro. El programa 13.1 ilustra la asignación de valores a los miembros individuales de la estructura `nacimiento`.

Program 13.1

```
// a program that defines and populates a record
#include <iostream>
using namespace std;

int main()
{
  struct
  {
    int month;
    int day;
    int year;
  } birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 86;

  cout << "My birth date is "
       << birth.month << '/'
       << birth.day   << '/'
       << birth.year  << endl;
```

La salida producida por el programa 13.1 es

```
Mi fecha de nacimiento es 12/28/86
```

Como en la mayor parte de las instrucciones en C++, el espaciamiento en la definición de una estructura no es rígido. Por ejemplo, la estructura nacimiento podría haberse definido también como

```
struct {int mes; int dia; int anio;} nacimiento;
```

Además, como con todas las definiciones de instrucciones en C++, pueden definirse múltiples variables en la misma instrucción. Por ejemplo, la instrucción de definición

```
struct
{
  int mes;
  int dia;
  int anio;
} nacimiento, actual;
```

crea dos variables de estructura que tienen la misma forma. Se hace referencia a los miembros de la primera estructura con los nombres individuales `nacimiento.mes`, `nacimiento.dia` y `nacimiento.anio`, mientras que se hace referencia a los miembros de la segunda estructura con los nombres `actual.mes, actual.dia` y `actual.anio`. Hay

que observar que la forma de esta instrucción de definición de una estructura particular es idéntica a la forma usada al definir cualquier variable de programa: el tipo de datos es seguido por una lista de nombres de variables.

Una modificación útil y muy usada para definir tipos de estructura es enlistar la forma de la estructura sin que le sigan los nombres de variables. En este caso, sin embargo, la lista de los miembros de la estructura debe ser precedida por un nombre de tipo de datos seleccionado por el usuario. Por ejemplo, en la declaración

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

el término `Fecha` es un nombre de tipo de estructura: define un tipo de datos nuevo que es una estructura de datos de la forma declarada.[1] Por convención la primera letra de un nombre de tipo de datos seleccionado por el usuario se pone en mayúscula, como en el nombre `Fecha`, lo cual ayuda a identificarla cuando se usa en instrucciones de declaración subsiguientes. Aquí, la declaración para la estructura `Fecha` crea un tipo de datos nuevo sin reservar en realidad ninguna ubicación de almacenamiento. Como tal no es una instrucción de definición. Tan sólo declara un tipo de estructura `Fecha` y describe cómo se ordenan los elementos de datos individuales dentro de la estructura. El almacenamiento real para los miembros de la estructura sólo se reserva cuando se asignan nombres de variable específicos. Por ejemplo, la instrucción de definición

```
Fecha nacimiento, actual;
```

reserva almacenamiento para dos variables de la estructura `Fecha` llamadas `nacimiento` y `actual`, respectivamente. Cada una de estas estructuras individuales tiene la forma declarada con anterioridad para la estructura `Fecha`.

La declaración de los tipos de datos de una estructura, como todas las declaraciones, puede ser global o local. El programa 13.2 ilustra la declaración global de un tipo de datos de `Fecha`. Dentro de `main()`, la variable `nacimiento` se define como una variable local del tipo `Fecha`.

La salida producida por el programa 13.2 es idéntica a la salida producida por el programa 13.1.

La inicialización de estructuras sigue las mismas reglas que para la inicialización de arreglos: las estructuras globales y locales pueden inicializarse al colocar después de la definición una lista de inicializadores. Por ejemplo, la instrucción de definición

```
Fecha nacimiento = {12, 28, 86};
```

puede usarse para reemplazar las primeras cuatro instrucciones internas en `main()` en el programa 13.2. Hay que observar que los inicializadores están separados por comas, no por puntos y comas.

---

[1] Como complemento debe mencionarse que una estructura en C++ también puede ser declarada como una clase sin funciones miembro y todos los miembros de datos públicos. Del mismo modo, una clase en C++ puede ser declarada como una `struct` que tiene todos los miembros de datos privados y todas las funciones miembro públicas. Por tanto, C++ proporciona dos sintaxis tanto para estructuras como para clases. La convención, sin embargo, es no mezclar las notaciones y siempre usar estructuras para crear tipos de registro y clases para proporcionar información verdadera y ocultar la implementación.

Program 13.2

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

struct Date    // this is a global declaration
{
  int month;
  int day;
  int year;
};

int main()
{
  Date birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 86;

  cout << "My birth date is " << birth.month << '/'
       << birth.day   << '/'
       << birth.year  << endl;
```

Los miembros individuales de una estructura no están restringidos a tipos de datos en números enteros, como lo ilustra la estructura `Fecha`. Puede utilizarse cualquier tipo de datos válido en C++. Por ejemplo, considérese un registro de empleados consistente en los siguientes elementos de datos:

```
Nombre:
Número de identificación:
Tarifa de pago normal:
Tarifa de pago de tiempo extra:
```

Una declaración adecuada para estos elementos de datos es

```
struct Reg_pago
{
  string nombre;
  int num_id;
  double tarifa_normal;
  double tarifa_textra;
};
```

Una vez que se ha declarado el tipo de datos `Reg_pago`, puede definirse e inicializarse una variable de estructura específica usando este tipo. Por ejemplo, la definición

```
Reg_pago empleado = {"H. Price", 12387, 15.89, 25.50};
```

crea una estructura llamada `empleado` del tipo de datos de `Reg_pago`. Los miembros individuales de `empleado` se inicializan con los datos respectivos enlistados entre llaves en la instrucción de definición.

Hay que observar que una estructura sencilla tan sólo es un método conveniente para combinar y almacenar elementos relacionados bajo un nombre común. Aunque una estructura sencilla es útil para identificar de manera explícita la relación entre sus miembros, los miembros individuales podrían definirse como variables separadas. Una de las ventajas reales de usar estructuras sólo se nota cuando el mismo tipo de datos se usa muchas veces en una lista. La creación de listas con el mismo tipo de datos es el tema de la siguiente sección.

Antes de dejar las estructuras sencillas, vale la pena señalar que los miembros individuales de una estructura pueden ser cualquier tipo de datos válido en C++, incluyendo arreglos y estructuras. Se usó un arreglo de caracteres como un miembro de la estructura empleado definida con anterioridad. Para tener acceso a un elemento de un arreglo miembro se requiere dar el nombre de la estructura, seguido por un punto, seguido por la designación del arreglo.

Incluir una estructura dentro de una estructura sigue las mismas reglas para incluir cualquier tipo de datos en una estructura. Por ejemplo, supóngase que una estructura consiste en un nombre y una fecha de nacimiento, donde se ha declarado una estructura `Fecha` como

```
struct Datos
{
  int mes;
  int dia;
  int anio;
};
```

Una definición adecuada de una estructura que incluya un `nombre` y una estructura de `Fecha` es

```
struct
{
struct Datos
  string nombre;
  Fecha nacimiento;
} persona;
```

Hay que observar que al declarar la estructura `Fecha`, el término `Fecha` es el nombre de un tipo de datos; por tanto aparece antes de las llaves en la instrucción de declaración. Al definir la variable de estructura para persona, `persona` es un nombre de variable; por tanto es el nombre de una estructura específica. Lo mismo sucede con la variable llamada `nacimiento`. Éste es el nombre de una estructura `Fecha` específica. Se tiene acceso a los miembros individuales en la estructura `persona` precediendo al miembro deseado con el nombre de la estructura seguido por un punto. Por ejemplo, `persona.nacimiento.mes` se refiere a la variable `mes` en la estructura `nacimiento` contenida en la estructura `persona`.

Ejercicios 13.1

1. Declare un tipo de estructura de datos llamado `Temp_e` para cada uno de los siguientes registros:
   a. el registro de un estudiante que consiste en un número de identificación del estudiante, número de créditos completados y el promedio acumulado de puntos de calificación.
   b. el registro de un estudiante que consiste en el nombre de un estudiante, la fecha de nacimiento, el número de créditos completados y el promedio acumulado de puntos de calificación.
   c. un registro de inventario que consiste en los elementos ilustrados antes en la figura 13.1.
   d. un registro de acciones que consiste en el nombre de la acción, el precio de la acción y la fecha de compra.
   e. un registro de inventario que consiste en un número de parte entero, la descripción de la parte, el número de partes en el inventario y un número entero para reordenar

2. Para los tipos de datos individuales declarados en el ejercicio 1, defina un nombre de variable de estructura adecuado e inicialice cada estructura con los siguientes datos apropiados:

   a. Número de identificación: 4672
      Número de créditos completados: 68
      Promedio de puntos de calificación: 3.01
   b. Nombre: Rhona Karp
      Fecha de nacimiento: 8/4/60
      Número de créditos completados: 96
      Promedio de puntos de calificación: 3.89
   c. Número de parte: 54002
      Familia de CI: ECL
      Tipo de función: NAND
      Suministro de voltaje: –5
      Unidades en existencia: 123
   d. Acción: IBM
      Precio de compra: 134.5
      Fecha de compra: 10/1/86
   e. Número de parte: 16879
      Descripción: Batería
      Número en existencia: 10
      Número de reorden: 3

3. a. Escriba un programa en C++ que pida a un usuario que introduzca el mes, día y año actuales. Almacene los datos introducidos en un registro definido de manera adecuada y despliegue la fecha de una manera apropiada.
   b. Modifique el programa escrito en el ejercicio 3a para usar un registro que acepte el tiempo actual en horas, minutos y segundos.

Point of Information

Homogeneous and Heterogeneous Data Structures

Both arrays and structures are structured data types. The difference between these tures is the types of elements they contain. An array is a *homogeneous* data structu

4. Escriba un programa en C++ que use una estructura para almacenar el nombre de una acción, sus ganancias estimadas por acción y la proporción estimada de costo-utilidad. Haga que el programa pida al usuario que introduzca estos elementos para cinco acciones diferentes, usando cada vez la misma estructura para almacenar los datos introducidos. Cuando se hayan introducido los datos para una acción particular, haga que el programa calcule y despliegue el precio de la acción anticipado basado en los valores de las ganancias y costo-utilidad introducidos. Por ejemplo, si un usuario introdujo los datos XYZ 1.56 12, el precio anticipado para una acción XYZ es (1.56)*(12) = $18.72.

5. Escriba un programa en C++ que acepte un tiempo introducido por el usuario en horas y minutos. Haga que el programa calcule y despliegue el tiempo un minuto después.

6. **a.** Escriba un programa en C++ que acepte una fecha introducida por el usuario. Haga que el programa calcule y despliegue la fecha del día siguiente. Para propósitos de este ejercicio, suponga que todos los meses constan de 30 días.
   b. Modifique el programa escrito en el ejercicio 6a para que dé cuenta del número real de días en cada mes.

13.2 ARREGLOS DE ESTRUCTURAS

El poder verdadero de las estructuras se comprende cuando la misma estructura se usa para listas de datos. Por ejemplo, suponga que los datos mostrados en la figura 13.3 deben ser procesados. Es evidente que los números de empleado pueden almacenarse juntos en un arreglo de números enteros, los nombres en un arreglo de cadenas y las tarifas de pago en un arreglo de números de precisión doble. Al organizar los datos de esta forma, cada columna en la figura 13.3 es considerada como una lista separada, la cual es almacenada en su propio arreglo. La correspondencia entre elementos para cada empleado individual se mantiene almacenando los datos de un empleado en la misma posición en cada arreglo.

Employee Number	Employee Name	Employee Pay Rate
32479	Abrams, B.	6.72
33623	Bohm, P.	7.54
34145	Donaldson, S.	5.56
35987	Ernst, T.	5.43
36203	Gwodz, K.	8.72
36417	Hanson, H.	7.64
37634	Monroe, G.	5.29

Figura 13.3 *Una lista de datos de empleados.*

La separación de la lista completa en tres arreglos individuales es desafortunada, porque todos los elementos que se relacionan con un solo empleado constituyen una organización natural de los datos en estructuras, como se ilustra en la figura 13.4. Usando una estructura, la integridad de la organización de los datos como un registro puede ser mantenida y reflejada por el programa. Bajo este enfoque, la lista ilustrada en la figura 13.4 puede procesarse como un arreglo sencillo de diez estructuras.

	Employee Number	Employee Name	Empl Pay I
1st structure →	32479	Abrams, B.	6.72
2nd structure →	33623	Bohm, P.	7.54
3rd structure →	34145	Donaldson, S.	5.56
4th structure →	35987	Ernst, T.	5.43
5th structure →	36203	Gwodz, K.	8.72
6th structure →	36417	Hanson, H.	7.64
7th structure →	37634	Monroe, G.	5.29

Figura 13.4 *Una lista de estructuras.*

Declarar un arreglo de estructuras es lo mismo que declarar un arreglo de cualquier otro tipo de variable. Por ejemplo, si el tipo de datos `Reg_pago` es declarado como

```
struct Reg_pago {int num_id; string nombre; double tarifa;};
```

entonces un arreglo de diez estructuras de éstas puede definirse como

```
Reg_pago empleado[10];
```

Esta instrucción de definición construye un arreglo de diez elementos, cada uno de los cuales es una estructura del tipo de datos `Reg_pago`. Hay que observar que la creación de un arreglo de diez estructuras tiene la misma forma que la creación de cualquier otro arreglo. Por ejemplo, crear un arreglo de diez números enteros llamada `empleado` requiere la declaración

```
int empleado[10];
```

En esta declaración el tipo de datos es un número entero, mientras en la declaración anterior para empleado el tipo de datos es `Reg_pago`.

Una vez que se ha declarado un arreglo de estructuras, se referencia un elemento de datos particular dando la posición de la estructura deseada en el arreglo seguida por un punto y el miembro de la estructura apropiado. Por ejemplo, la variable `empleado-[0].tarifa` hace referencia al miembro `tarifa` de la primera estructura `empleado` en el arreglo `empleado`. Incluir estructuras como elementos de un arreglo permite que se procese una lista de estructuras usando técnicas de programación de arreglos estándares. El programa 13.3 despliega los registros de los primeros cinco empleados ilustrados en la figura 13.4.

Program 13.3

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

const int NUMRECS = 5;   // maximum number of records

struct PayRec   // this is a global declaration
{
  int id;
  string name;
  double rate;
};

int main()
{
  int i;
  PayRec employee[NUMRECS] = {
                              { 32479, "Abrams, B.", 6.72 },
                              { 33623, "Bohm, P.", 7.54},
                              { 34145, "Donaldson, S.",  5.56},
                              { 35987, "Ernst, T.", 5.43 },
```

(Continued)

```
cout << endl;   // start on a new line
cout << setiosflags(ios::left);  // left justify the output
for ( i = 0; i < NUMRECS; i++)
  cout << setw(7)  << employee[i].id
       << setw(15) << employee[i].name
       << setw(6)  << employee[i].rate << endl;
```

La salida desplegada por el programa 13.3 es:

```
32479  Abrams, B.     6.72
33623  Bohm, P.       7.54
34145  Donaldson, S.  5.56
35987  Ernst, T.      5.43
36203  Gwodz, K.      8.72
```

Al revisar el programa 13.3 se puede observar la inicialización del arreglo de estructuras. Aunque los inicializadores para cada estructura se han encerrado en llaves interiores, éstas no son estrictamente necesarias porque todos los miembros se han inicializado. Como con todas las variables externas y estáticas, en ausencia de inicializadores explícitos, los elementos numéricos de los arreglos o estructuras estáticos y externos se inicializan en cero y sus elementos de carácter se inicializan en `NULL`. El manipulador `setiosflags (ios::left)` incluido en el flujo del objeto `cout` obliga a que cada nombre sea desplegado justificado a la izquierda en su ancho de campo designado.

Ejercicios 13.2

1. Defina arreglos de 100 estructuras para cada uno de los tipos de datos descritos en el ejercicio 1 de la sección anterior.
2. **a.** Usando el tipo de datos

```
struct MesDias
{
  string nombre;
  int dias;
};
```

defina un arreglo de 12 estructuras del tipo `MesDias`. Nombre al arreglo `convertir[]` e inicialice el arreglo con los nombres de los 12 meses en un año y el número de días en cada mes.

b. Incluya el arreglo creado en el ejercicio 2a en un programa que despliegue los nombres y número de días en cada mes.

3. Usando el tipo de datos declarado en el ejercicio 2a, escriba un programa en C++ que acepte un mes de un usuario en forma numérica y despliegue el nombre del mes y el número de días en el mes. Por tanto, en respuesta a una entrada de 3, el programa desplegará `Marzo tiene 31 dias`.

4. a. Declare una estructura sencilla con un tipo de datos adecuado para una estructura de empleados del tipo ilustrado a continuación:

Number	Name	Rate	I
3462	Jones	4.62	4
6793	Robbins	5.83	3
6985	Smith	5.22	4

b. Usando el tipo de datos declarado en el ejercicio 4a, escriba un programa en C++ que acepte de manera interactiva los datos anteriores en un arreglo de seis estructuras. Una vez que se han introducido los datos, el programa deberá crear un reporte de nómina enlistando el nombre, número y salario bruto de cada empleado. Incluya el pago bruto total de todos los empleados al final del reporte.

5. a. Declare una estructura de datos sencilla con un tipo de datos adecuado para una estructura de automóvil del tipo ilustrado:

Car Number	Miles Driven	Gallons Used
25	1450	62
36	3240	136

b. Usando el tipo de datos declarado para el ejercicio 5a, escriba un programa en C++ que acepte en forma interactiva los datos anteriores en un arreglo de cinco estructuras. Una vez que se hayan introducido los datos, el programa deberá crear un reporte mostrando el número de automóvil y las millas por galón logradas por cada automóvil. Al final del reporte incluya las millas por galón promedio logradas por la flotilla completa de automóviles.

13.3 ESTRUCTURAS COMO ARGUMENTOS DE FUNCIÓN

Los miembros individuales de estructura pueden transmitirse a una función de la misma manera que cualquier variable escalar. Por ejemplo, dada la definición de estructura

```
struct Empleado
{
  int num_id;
  double tarifaPago;
  double horas;
} empl;
```

la instrucción

```
desplegar(empl.num_id);
```

transmite una copia del miembro de la estructura `empl.num_id` a una función llamada `desplegar()`. Del mismo modo, la instrucción

```
calcPago(empl.tarifaPago,empl.horas);
```

transmite copias de los valores almacenados en los miembros de la estructura `empl.tarifaPago` y `empl.horas` a la función `calcPago()`. Ambas funciones, `desplegar()` y `calcPago()`, deben declarar los tipos de datos correctos para sus respectivos argumentos.

También pueden transmitirse copias completas de todos los miembros de una estructura a una función al incluir el nombre de la estructura como un argumento para la función llamada. Por ejemplo, la llamada a la función

```
calcNeto(empl);
```

transmite una copia de la estructura `empl` completa a `calcNeto()`. Dentro de `calcNeto()` debe hacerse una declaración apropiada para recibir la estructura. El programa 13.4 declara un tipo de datos global para una estructura de empleados. Este tipo es usado entonces por las funciones `main()` y `calcNeto()` para definir estructuras específicas con los `nombres empl` y `temp`, respectivamente.

La salida producida por el programa 13.4 es:

```
El pago neto para el empleado 6782 es $361.66
```

Al revisar el programa 13.4, se observará que tanto `main()` como `calcNeto()` usan el mismo tipo de datos para definir sus variables individuales de estructura. La variable de estructura definida en `main()` y la variable de estructura definida en `calcNeto()` son dos estructuras completamente diferentes. Los cambios hechos a la variable local `temp` en `calcNeto()` no se reflejan en la variable `empl` de `main()`. De hecho, dado que ambas variables de estructura son locales para sus respectivas funciones, podría haberse usado el mismo nombre de variable de estructura en ambas funciones sin ambigüedad.

Programa 13.4

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

struct Empleado        // declara un tipo global
{
  int num_id;
  double tarifaPago;
  double horas;
};

double calcNeto(Empleado);     // prototipo de la funcion

int main()
{
  Empleado empl = {6782, 8.93, 40.5};
  double pagoNeto;

  pagoNeto = calcNeto(empl);     // transmite copias de los valores en empl
    // establece formatos de salida
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "El pago neto para el empleado " << empl.num_id
       << " es $" << pagoNeto << endl;

  return 0;
}

double calcNeto(Empleado temp) // temp es del tipo de datos Empleado
{
  return temp.tarifaPago * temp.horas;
}
```

Cuando `calcNeto()` es llamado por `main()`, se transmiten copias de los valores de la estructura `empl` a la estructura `temp`. Entonces `calcNeto()` usa dos de los valores miembro transmitidos para calcular un número, el cual es devuelto a `main()`. Dado que `calcNeto()` devuelve un número que no es entero, el tipo de datos del valor devuelto debe incluirse en todas las declaraciones para `calcNeto()`.

Una alternativa para la llamada a la función con transmisión por valor ilustrada en el programa 13.4, en el cual la función llamada recibe una copia de una estructura, es una transmisión por referencia, la que transmite una referencia a una estructura. Hacerlo así permite que la función llamada tenga acceso directo a la variable de estructura en la fun-

ción que llama y alterar sus valores. Por ejemplo, haciendo referencia al programa 13.4, el prototipo de `calcNeto()` puede modificarse a

```
double calcNeto(Empleado &);
```

Si se usa este prototipo de la función y la función `calcNeto()` se vuelve a escribir para ajustarse a él, la función `main()` en el programa 13.4 puede usarse como está. El programa 13.4a ilustra estos cambios dentro del contexto de un programa completo.

Program 13.4a

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

struct Employee       // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

double calcNet(Employee&);   // function prototype

int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(emp);       // pass a reference

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" <<  netPay << endl;

  return 0;
}

double calcNet(Employee& temp)   // temp is a reference variable
{
```

El programa 13.4a produce la misma salida que el programa 13.4, excepto que la función `calcNeto()` en el programa 13.4a recibe acceso directo a la estructura `empl` en lugar de una copia de ellas. Esto significa que el nombre de la variable `temp` dentro de `calcNeto` es un nombre alterno para la variable `empl` en `main()` y los cambios a `temp` son cambios directos a `empl`. Aunque en ambos programas se hace la misma llamada a la función, `calcNeto(empl)`, la llamada en el programa 13.4a transmite una referencia, mientras que en el programa 13.4 transmite valores.

Transmisión de un apuntador

En lugar de transmitir una referencia, puede usarse un apuntador. Usar un apuntador requiere, además de modificar el prototipo de la función y la línea de encabezado, que la llamada a `calcNeto()` en el programa 13.4 se modifique a

```
calcNeto(&empl);
```

Aquí la llamada a la función indica con claridad que se está transmitiendo una dirección (lo cual no sucede en el programa 13.4a). La desventaja, sin embargo, está en la notación de desreferenciación requerida en forma interna en la función. No obstante, como los apuntadores se usan en forma amplia en la práctica, vale la pena familiarizarse con la notación empleada.

Para almacenar en forma correcta la dirección transmitida, `calcNeto()` debe declarar sus parámetros como un apuntador. Una definición de función adecuada para `calcNeto()` es

```
calcNeto(Empleado *pt)
```

Aquí, la declaración para `pt` declara este parámetro como un apuntador a una estructura de tipo `Empleado`. El apuntador `pt` recibe la dirección inicial de una estructura siempre que se llame a `calcNeto()`. Dentro de `calcNeto()`, este apuntador se usa para hacer referencia directa a cualquier miembro en la estructura. Por ejemplo `(*pt).num_id` se refiere al miembro `num_id` de la estructura `(*pt).tarifaPago` se refiere al miembro `tarifaPago` de la estructura y `(*pt).horas` se refiere al miembro `horas` de la estructura. Estas relaciones se ilustran en la figura 13.5.

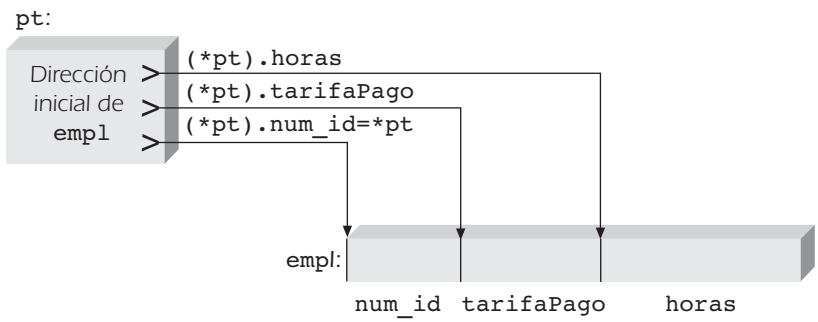


Figura 13.5 *Se puede utilizar un apuntador para tener acceso a los miembros de la estructura.*

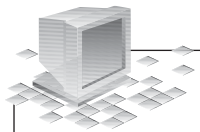
Los paréntesis que rodean a la expresión `*pt` en la figura 13.5 son necesarios para tener acceso inicial a "la estructura cuya dirección está en `pt`". Esto es seguido por un identificador para tener acceso al miembro deseado dentro de la estructura. En ausencia de paréntesis, el operador `.` del miembro de la estructura toma precedencia sobre el operador de indirección. Por tanto, la expresión `*pt.horas` es otra forma de escribir `*(pt.horas)`, lo cual se referiría a "la variable cuya dirección está en la variable `pt.horas`". Es evidente que esta última expresión no tiene sentido porque no hay una estructura llamada `pt` y `horas` no contiene una dirección. Como se ilustra en la figura 13.5, la dirección inicial de la estructura `empl` también es la dirección del primer miembro de la estructura.

El uso de apuntadores de esta manera es tan común que existe una notación especial para ello. La expresión general `(*apuntador).miembro` siempre puede reemplazarse con la notación `apuntador->miembro`, donde el operador `->` se construye usando un signo de menos seguido por una flecha que apunta a la derecha (símbolo mayor que). Puede usarse cualquier expresión para localizar el miembro deseado. Por ejemplo, las siguientes expresiones son equivalentes:

`(*pt).num_id`	puede reemplazarse con	`pt->num_id`
`(*pt).tarifaPago`	puede reemplazarse con	`pt->tarifaPago`
`(*pt).horas`	puede reemplazarse con	`pt->horas`

El programa 13.5 ilustra la transmisión de la dirección de una estructura y el uso de un apuntador con la nueva notación para hacer referencia directa a la estructura.

El nombre del parámetro apuntador declarado en el programa 13.5 es seleccionado, por supuesto, por el programador. Cuando se llama a `calcNeto()`, la dirección inicial de `empl` es transmitida a la función. Usando esta dirección como punto de partida, se tiene acceso a los miembros individuales de la estructura al incluir sus nombres con el apuntador.



Program 13.5

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

struct Employee   // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

double calcNet(Employee *);   //function prototype

int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp);       // pass an address

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee *pt)   // pt is a pointer to a
{                                 // structure of Employee type
```

Como con todas las expresiones en C++ que tienen acceso a una variable, también pueden aplicarse a ellas los operadores de incremento y decremento. Por ejemplo, la expresión

```
++pt->horas
```

añade uno al miembro `horas` de la estructura `empl`. Dado que el operador `->` tiene una prioridad mayor que el operador de incremento, primero se tiene acceso al miembro `horas` y luego se aplica el incremento. De manera alternativa, la expresión `(++pt)->horas` usa el prefijo del operador de incremento para aumentar la dirección en `pt` antes que se tenga acceso al miembro `horas`. Del mismo modo, la expresión `(pt++)->horas` usa el posfijo del operador de incremento para aumentar la dirección en `pt` después que se tie-

ne acceso al miembro `horas`. En ambos casos, sin embargo, debe haber suficientes estructuras definidas para asegurar que los apuntadores incrementados en realidad apuntan a estructuras legítimas.

Como ejemplo, la figura 13.6 ilustra un arreglo de tres estructuras de tipo `empleado`. Suponiendo que la dirección de `empl[1]` está almacenada en la variable apuntadora `pt`, la expresión `++pt` cambia la dirección en `pt` a la dirección inicial de `empl[2]`, mientras la expresión `--pt` cambia la dirección para apuntar a `empl[0]`.

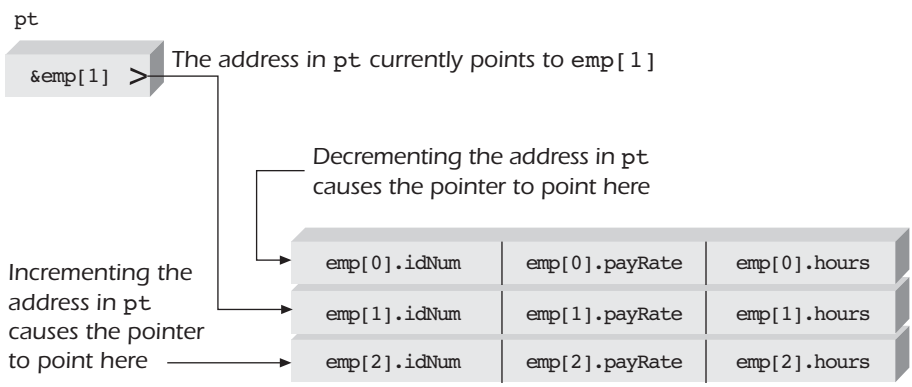


Figura 13.6 *Cambiar direcciones en el apuntador.*

Devolución de estructuras

En la práctica, la mayor parte de las funciones que manejan estructuras reciben acceso directo a una estructura al recibir una referencia o dirección a una estructura. Entonces pueden hacerse los cambios a la estructura en forma directa desde dentro de la función. Sin embargo, si se desea hacer que una función devuelva una estructura separada, deben seguirse los mismos procedimientos para devolver estructuras de datos completas que para devolver valores escalares. Estos procedimientos incluyen declarar la función de manera apropiada y alertar a cualquier función que llame del tipo de estructura de datos que se está devolviendo. Por ejemplo, la función `obtenerValores()` en el programa 13.6 devuelve una estructura completa a `main()`.

Program 13.6

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

struct  Employee       // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

Employee getVals();   // function prototype

int main()
{
  Employee emp;

  emp = getVals();
  cout << "\nThe employee id number is " << emp.idNum
       << "\nThe employee pay rate is $" << emp.payRate
       << "\nThe employee hours are " << emp.hours << endl;

  return 0;
}

Employee getVals() // return an employee structure
{
  Employee next;

  next.idNum = 6789;
  next.payRate = 16.25;
  next.hours = 38.0;
```

La siguiente salida se despliega cuando se ejecuta el programa 13.6:

```
El número de identificación del empleado es 6789
La tarifa de pago del empleado es $16.25
Las horas del empleado son 38
```

Dado que la función `obtenerValores()` devuelve una estructura, el encabezado de la función para `obtenerValores()` debe especificar el tipo de estructura que se está devolviendo. Debido a que `obtenerValores()` no recibe ningún argumento, el encabezado de la función no tiene declaraciones de parámetros y consiste en la línea

```
Empleado obtenerValores();
```

Dentro de `obtenerValores()`, la variable `siguiente` se define como una estructura del tipo que se va a devolver. Después que se han asignado valores a la estructura `siguiente`, los valores de la estructura son devueltos al incluir el nombre de la estructura dentro de los paréntesis de la instrucción de devolución.

En el lado receptor, debe alertarse a `main()` que la función `obtenerValores()` devolverá una estructura. Esto se maneja incluyendo una declaración de función para `obtenerValores()` en `main()`. Hay que observar que estos pasos para devolver una estructura a partir de una función son idénticos a los procedimientos normales para devolver tipos de datos escalares descritos antes en el capítulo 6.

Ejercicios 13.3

1. Escriba una función en C++ llamada `dias()` que determine el número de días desde el inicio del siglo para cualquier fecha transmitida como una estructura. Use la estructura `Fecha`

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

   Al escribir la función `dias()` use la convención de que todos los años tienen 360 días y cada mes consiste en 30 días. La función deberá devolver el número de días para cualquier estructura `Fecha` que se le transmita.

2. Escriba una función en C++ llamada `dif_dias()` que calcule y devuelva la diferencia entre dos fechas. Cada fecha es transmitida a la función como una estructura usando el siguiente tipo global:

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

   La función `dif_dias()` deberá hacer dos llamadas a la función `dias()` escrita para el ejercicio 1.

3. **a.** Vuelva a escribir la función `dias()` redactada para el ejercicio 1 para recibir una referencia a una estructura `Fecha`, en lugar de una copia de la estructura completa.
   b. Repita el ejercicio 3a usando un apuntador en lugar de una referencia.

4. **a.** Escriba una función en C++ llamada `mayor()` que devuelva la fecha más reciente de cualquier par de fechas que se le transmitan. Por ejemplo, si se transmiten las fechas 10/9/2005 y 11/3/2005 a `mayor()`, será devuelta la segunda fecha.
   b. Incluya la función `mayor()` que se escribió para el ejercicio 4a en un programa completo. Almacene la estructura `Fecha` devuelta por `mayor()` en una estructura `Fecha` separada y despliegue los valores miembros de la `Fecha` devuelta.

5. a. En dos dimensiones, un vector matemático es un par de números que representan flechas dirigidas en un plano, como lo muestran los vectores matemáticos v_1 y v_2 en la figura 13.7.

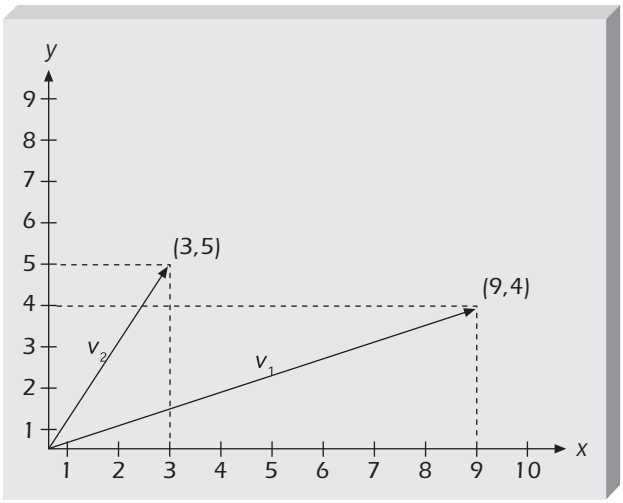


Figura 13.7 Gráfica para el ejercicio 5.

Los vectores matemáticos bidimensionales pueden escribirse en la forma (a,b), donde a y b son llamados los componentes x y y del vector, respectivamente. Por ejemplo, para los vectores ilustrados en la figura 13.7, $v_1 = (9, 4)$ y $v_2 = (3, 5)$. Para los vectores se aplican las siguientes operaciones:

Si $v_1 = (a,b)$ y $v_2 = (c,d)$
$v_1 + v_2 = (a,b) + (c,d) = (a + c, b + d)$
$v_1 - v_2 = (a,b) - (c,d) = (a - c, b - d)$

Usando esta información, escriba un programa en C++ que defina un arreglo de dos registros de vectores, donde cada registro consista en dos componentes de precisión doble a y b. Su programa deberá permitir a un usuario introducir dos vectores, llamar a dos funciones que devuelvan la suma y la diferencia de los vectores introducidos, y desplegar los resultados calculados por estas funciones.

b. Además de las operaciones definidas en el ejercicio 5a, dos operaciones adicionales con vectores son la negación y el valor absoluto. Para un vector v_1 con componentes (a,b) estas operaciones se definen como sigue:

negación: $-v_1 = -(a,b) = (-a,-b)$
valor absoluto: $| v_1 |$ = sqrt $(a * a + b * b)$

Usando esta información, modifique el programa que escribió para el ejercicio 5a para desplegar la negación y los valores absoluto de ambos vectores introducidos por un usuario al igual que la negación y el valor absoluto de la suma de los dos vectores introducidos.

13.4 LISTAS VINCULADAS

Un problema clásico del manejo de datos es hacer adiciones o eliminaciones a estructuras existentes que se mantienen en un orden específico. Esto se ilustra mejor al considerar la lista alfabética de teléfonos que se muestra en la figura 13.8. Empezando con este conjunto inicial de nombres y números telefónicos, se desea agregar estructuras nuevas a la lista en la secuencia alfabética apropiada, y eliminar estructuras existentes, de tal forma que se elimine el almacenamiento para las estructuras eliminadas.

Amador, Samuel
(555) 898-2392
Domínguez, Edith
(555) 682-3104
López, Juan
(555) 718-4581
Martínez, Esteban
(555) 382-7070
Zapata, Horacio
(555) 219-9912

Figura 13.8 *Una lista telefónica en orden alfabético.*

Aunque la inserción o eliminación de estructuras ordenadas puede lograrse usando un arreglo de estructuras, estos arreglos no son representaciones eficientes para agregar o eliminar estructuras internas en el arreglo. Los arreglos son fijos y tienen un tamaño especificado con anterioridad. Eliminar una estructura de un arreglo crea un espacio vacío que requiere una marcación especial o mover hacia arriba todos los elementos debajo de la estructura eliminada para cerrar el espacio vacío. Del mismo modo, agregar una estructura al cuerpo de un arreglo de estructuras requiere que todos los elementos debajo de la adición se muevan hacia abajo para hacer espacio a la nueva entrada, o el nuevo elemento podría agregarse al final del arreglo existente y luego tendría que reordenarse el arreglo para restablecer el orden apropiado de las estructuras. Por tanto, agregar o eliminar registros de una lista así por lo general requiere reestructurar y volver a escribir la lista, una práctica molesta, tardada e ineficiente.

Una lista vinculada proporciona un método conveniente para mantener una lista que cambia en forma constante, sin necesidad de reordenar y reestructurar de manera continua la lista completa. Una lista vinculada es tan sólo un conjunto de estructuras en el que cada estructura contiene al menos un miembro cuyo valor es la dirección de la siguiente estructura ordenada lógicamente en la lista. En lugar de requerir que cada registro esté almacenado en forma física en el orden apropiado, cada estructura nueva se agrega de manera física siempre que la computadora tiene espacio libre en su área de almacenamiento. Los registros se "vinculan" incluyendo la dirección del siguiente registro en el registro que lo precede inmediatamente. Desde el punto de vista de la programación, la estructura actual que se está procesando contiene la dirección del siguiente registro, sin importar dónde esté almacenada en realidad la siguiente estructura.

El concepto de lista vinculada se ilustra en la figura 13.9. Aunque los datos reales para la estructura `López` ilustrada en la figura pueden estar almacenados físicamente en cualquier parte en la computadora, el miembro adicional incluido al final de la estructura `Domínguez` mantiene el orden alfabético apropiado. Este miembro proporciona la dirección inicial de la ubicación donde está almacenado el registro López. Como podría esperarse, este miembro es un apuntador.

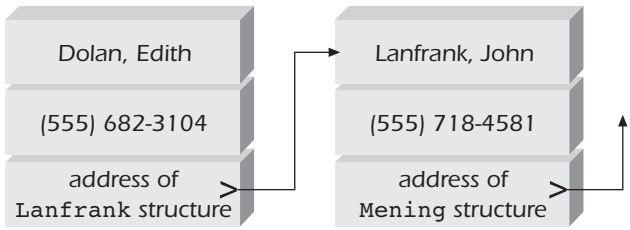


Figura 13.9 *Uso de apuntadores para vincular estructuras.*

Para poder observar la utilidad del apuntador en la estructura Domínguez, se agregará un número telefónico para Julia Hernández en la lista alfabética mostrada en la figura 13.8. Los datos para Julia Hernández se almacenan en una estructura de datos empleando el mismo tipo utilizado para las estructuras existentes. Para asegurar que el número telefónico para Hernández se despliega de manera correcta después del número telefónico para Domínguez, la dirección en la estructura Domínguez debe alterarse para que apunte a la estructura Hernández, y la dirección en la estructura Hernández debe establecerse para que apunte a la estructura López. Esto se ilustra en la figura 13.10. Hay que observar que el apuntador en cada estructura tan sólo apunta a la ubicación de la siguiente estructura ordenada, aun si esa estructura no se localiza físicamente en el orden correcto.

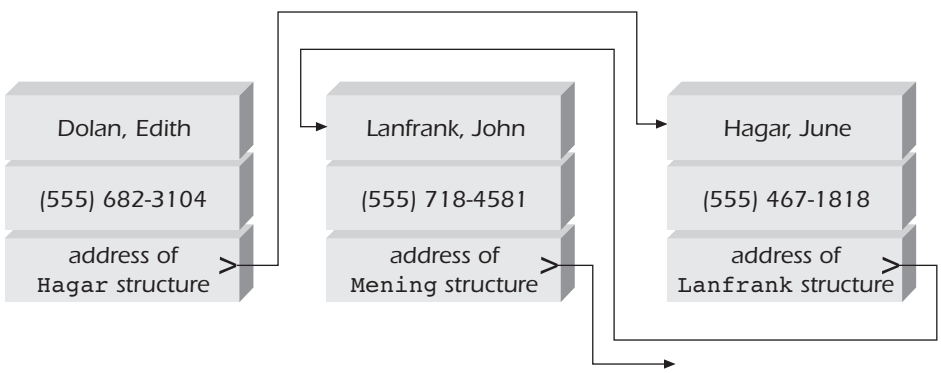


Figura 13.10 *Ajustar direcciones para que apunten a estructuras apropiadas.*

La eliminación de una estructura de la lista ordenada es el proceso inverso de agregar un registro. El registro real se elimina lógicamente de la lista con sólo cambiar la dirección en la estructura que la precede para que apunte a la que sigue de inmediato al registro eliminado.

Cada estructura en una lista vinculada tiene el mismo formato; sin embargo, es claro que el último registro no puede tener un valor apuntador válido que apunte a otro registro, debido a que no hay ninguno. C++ proporciona un valor apuntador especial llamado `NULL` que actúa como un centinela o bandera para indicar cuando se ha procesado el último registro. El valor apuntador `NULL`, como su contraparte de fin de cadena, tiene un valor numérico de cero.

Además de un valor centinela de fin de lista, también debe proporcionarse un apuntador especial para almacenar la dirección de la primera estructura en la lista. La figura 13.11 ilustra el conjunto completo de apuntadores y estructuras para una lista consistente en tres nombres.

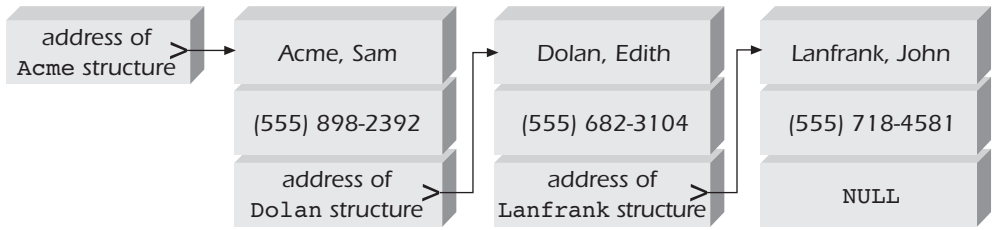


Figura 13.11 *Uso de los valores apuntadores inicial y final.*

La inclusión de un apuntador en una estructura no debería ser sorprendente. Como se descubrió en la sección 13.1, en C++ una estructura puede contener cualquier tipo de datos. Por ejemplo, la declaración de estructura

```
struct Prueba
{
  int num_id;
  double *ptPago
};
```

declara un tipo de estructura consistente en dos miembros. El primer miembro es una variable en número entero llamada `num_id` y la segunda variable es un apuntador llamado `ptPago`, el cual es un apuntador a un número de precisión doble. El programa 13.7 ilustra que el miembro apuntador de una estructura se usa como cualquier otra variable apuntadora.

La salida producida al ejecutar el programa 13.7 es

```
El empleado numero 12345 recibio un pago de $456.20
```

La figura 13.12 ilustra la relación entre los miembros de la estructura `empl` definida en el programa 13.7 y la variable llamada pago. El valor asignado a `empl.num_id` es el número 12345 y el valor asignado a pago es 456.20. La dirección de la variable `pago` es asignada al miembro de la estructura `empl.ptPago`. Dado que este miembro se ha definido como un apuntador a un número de precisión doble, colocar la dirección de la variable de precisión doble `pago` en él es un uso correcto de este miembro. Por último, dado que el operador de miembro `.` tiene una precedencia mayor que el operador de indirección `*`, la expresión usada en la instrucción `cout` en el programa 13.7 es correcta. La expresión `*empl.ptPago` es equivalente a la expresión `*(empl.ptPago)`, la cual se traduce como "la variable cuya dirección está contenida en el miembro `empl.ptPago`".

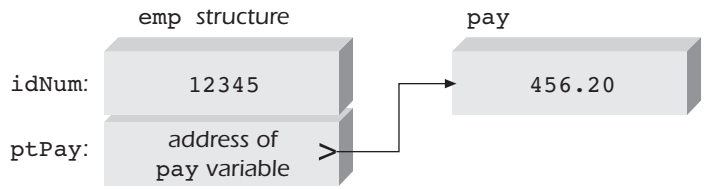


Figura 13.12 *Almacenamiento de una dirección en un miembro de una estructura.*

Program 13.7

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

struct Test
{
  int idNum;
  double *ptPay;
};

int main()
{
  Test emp;
  double pay = 456.20;

  emp.idNum = 12345;
  emp.ptPay = &pay;
    // set output formats
  cout << setw(6)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "\nEmployee number " << emp.idNum << " was paid $"
       << *emp.ptPay << endl;
```

Aunque el apuntador definido en el programa 13.7 se ha usado en una forma bastante trivial, el programa ilustra el concepto de incluir un apuntador en una estructura. Este concepto puede extenderse con facilidad para crear una lista vinculada de estructuras adecuada para almacenar los nombres y números telefónicos mostrados en la figura 13.8. La siguiente declaración crea un tipo para una estructura así:

```
struct Tipo_tel
{
  string nombre;
  string num_telef;
  Tipo_tel *siguientedir;
};
```

El último miembro en esta estructura es un apuntador adecuado para almacenar la dirección de una estructura del tipo `Tipo_tel`.

El programa 13.8 ilustra el uso del tipo `Tipo_tel` al definir de manera específica tres estructuras que tienen esta forma. Las tres estructuras se nombran `t1, t2` y `t3`, respectivamente, y los miembros de nombre y número telefónico de cada una de estas estructuras se inicializan cuando se definen las estructuras, usando los datos mostrados en la figura 13.8.

Program 13.8

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

struct TeleType
{
  string name;
  string phoneNo;
  TeleType *nextaddr;
};

int main()
{
  TeleType t1 = {"Acme, Sam","(555) 898-2392"};
  TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
  TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
  TeleType *first;     // create a pointer to a structure

  first = &t1;         // store t1's address in first
  t1.nextaddr = &t2;   // store t2's address in t1.nextaddr
  t2.nextaddr = &t3;   // store t3's address in t2.nextaddr
  t3.nextaddr = NULL;  // store a NULL address in t3.nextaddr

  cout << endl << first->name
       << endl << t1.nextaddr->name
       << endl << t2.nextaddr->name
       << endl;
```

La salida producida al ejecutar el programa 13.8 es

```
Amador, Samuel
Domínguez, Edith
López, Juan
```

El programa 13.8 demuestra el uso de apuntadores para tener acceso a miembros de estructura sucesivos. Como se ilustra en la figura 13.13, cada estructura contiene la dirección de la siguiente estructura en la lista.

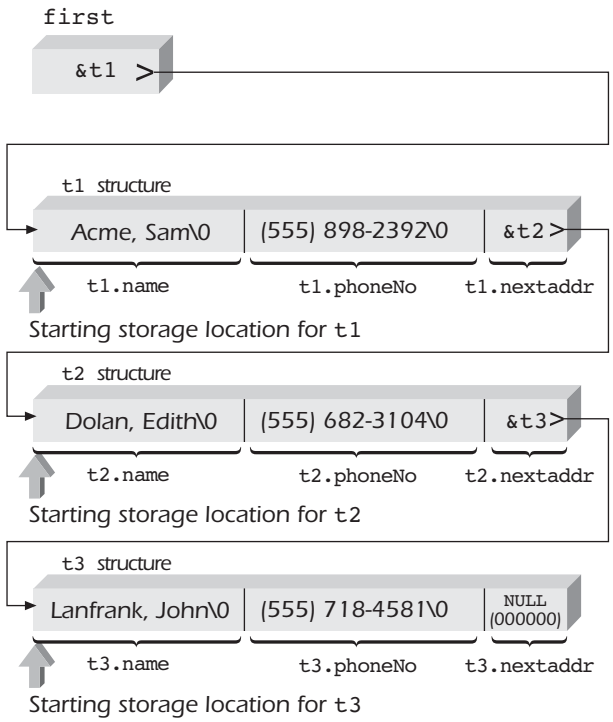


Figura 13.13 *La relación entre estructuras en el programa 13.8.*

La inicialización de los nombres y números telefónicos para cada una de las estructuras definidas en el programa 13.9 es sencilla. Aunque cada estructura consiste en tres miembros, sólo se inicializan los primeros dos miembros de cada estructura. Como ambos miembros son arreglos de caracteres, pueden inicializarse con cadenas. El miembro restante de cada estructura es un apuntador. Para crear una lista vinculada, cada apuntador a la estructura debe tener asignada la dirección de la siguiente estructura en la lista.

Las cuatro instrucciones de asignación en el programa 13.8 ejecutan las asignaciones correctas. La expresión `primero = &t1` almacena la dirección de la primera estructura en la lista en la variable apuntadora llamada primero. La expresión `t1.siguientedir = &t2` almacena la dirección inicial de la estructura `t2` en el miembro apuntador de la estructura `t1`. Del mismo modo, la expresión `t2.siguientedir = &t3` almacena la dirección inicial de la estructura `t3` en el miembro apuntador de la estructura `t2`. Para terminar la lista, el valor del apuntador `NULL`, el cual es cero, se almacena en el miembro apuntador de la estructura `t3`.

Una vez que se han asignado valores a cada miembro de la estructura y se han almacenado direcciones correctas en los apuntadores apropiados, las direcciones en los apuntadores se usan para tener acceso al miembro nombre de cada estructura. Por ejemplo, la expresión `t1.siguientedir->nombre` se refiere al miembro nombre de la estructura cuya dirección está en el miembro `siguientedir` de la estructura `t1`. Las precedencias del operador de miembro y el operador de apuntador de estructura `->` son iguales y se evalúan de izquierda a derecha. Por tanto, la expresión `t1.siguientedir->nombre` se evalúa como `(t1.siguientedir)->nombre`. Dado que `t1.siguientedir` contiene la dirección de la estructura `t2`, se tiene acceso al nombre apropiado.

La expresión `t1.siguientedir->nombre` puede reemplazarse, por supuesto, por la expresión equivalente `(*t1.siguientedir).nombre`, la cual usa el operador de indirección más convencional. Esta expresión también se refiere a "el miembro nombre de la variable cuya dirección está en `t1.siguientedir`".

Las direcciones en una lista de estructuras vinculada pueden usarse para recorrer en un ciclo la lista completa. Conforme se tiene acceso a cada estructura puede examinarse para seleccionar un valor específico o para imprimir una lista completa. Por ejemplo, la función `desplegar()` en el programa 13.9 ilustra el uso de un ciclo `while`, el cual usa la dirección en cada miembro apuntador de la estructura para recorrer en un ciclo toda la lista y desplegar en forma sucesiva los datos almacenados en cada estructura.

La salida producida por el programa 13.9 es

```
Amador, Samuel          (555) 898-2392
Domínguez, Edith        (555) 682-3104
López, Juan             (555) 718-4581
```

El concepto importante ilustrado por el programa 13.9 es el uso de la dirección en una estructura para tener acceso a los miembros de la siguiente estructura en la lista. Cuando se invoca a la función `desplegar()`, se le transmite el valor almacenado en la variable llamada `primero`. Dado que `primero` es una variable apuntadora, el valor real que se transmite es una dirección (la dirección de la estructura `t1`). La función `desplegar()` acepta el valor transmitido en el argumento llamado `contenido`. Para almacenar en forma correcta la dirección transmitida, `contenido` se declara como un apuntador a una estructura del tipo `Tipo_tel`. Dentro de `desplegar()`, se usa un ciclo `while` para recorrer las estructuras vinculadas, empezando con la estructura cuya dirección está en `contenido`. La condición probada en la instrucción `while` compara el valor en `contenido`, el cual es una dirección, con el valor `NULL`. Para cada dirección válida se desplegarán los miembros de nombre y número telefónico de la estructura direccionada. La dirección en `contenido` se actualiza entonces con la dirección en el miembro apuntador de la estructura actual. Entonces se vuelve a evaluar la dirección en `contenido`, y el proceso continúa mientras la dirección en contenido no es igual al valor `NULL`. La función `desplegar()` no "sabe" nada respecto a los nombres de las estructuras declaradas en `main()` o ni siquiera cuántas estructuras existen. Tan sólo recorre la lista vinculada, estructura por estructura, hasta que encuentra la dirección `NULL` de fin de la lista. Dado que el valor de `NULL` es cero, la condición probada puede ser reemplazada por la expresión equivalente `contenido`.

Una desventaja del programa 13.9 es que se definen exactamente tres estructuras en `main()` por nombre y el almacenamiento para ellas se reserva en tiempo de compilación. Si se requiriera una cuarta estructura, la estructura adicional tendría que ser declarada y volver a compilar el programa. En la siguiente sección se muestra cómo hacer que la computadora asigne y libere en forma dinámica almacenamiento para estructuras en tiempo de ejecución, conforme se requiera el almacenamiento. Sólo cuando se va a agregar una estructura nueva a la lista, y mientras el programa esté en ejecución, se crea almacenamiento para la estructura nueva. Del mismo modo, cuando ya no se necesita una estructura y puede eliminarse de la lista, se renuncia al almacenamiento para el registro eliminado y se devuelve a la computadora.

Program 13.9

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

struct TeleType
{
  string name;
  string phoneNo;
  TeleType *nextaddr;
};

void display(TeleType *);       // function prototype

int main()
{
  TeleType t1 = {"Acme, Sam","(555) 898-2392"};
  TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
  TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
  TeleType *first;    // create a pointer to a structure

  first = &t1;        // store t1's address in first
  t1.nextaddr = &t2;  // store t2's address in t1.nextaddr
  t2.nextaddr = &t3;  // store t3's address in t2.nextaddr
  t3.nextaddr = NULL; // store the NULL address in t3.nextaddr

  display(first);     // send the address of the first structure

  return 0;
}

void display(TeleType *contents) // contents is a pointer to a stru
{                                 // of type TeleType
  while (contents != NULL)   // display till end of linked list
  {
    cout << endl << setiosflags(ios::left)
         << setw(30) << contents->name
         << setw(20) << contents->phoneNo ;
    contents = contents->nextaddr;     // get next address
  }
  cout << endl;

  return;
```

Ejercicios 13.4

1. Modifique el programa 13.9 para solicitar al usuario un nombre. Haga que el programa busque el nombre introducido en la lista existente. Si el nombre está en la lista, despliegue el número telefónico correspondiente; de lo contrario despliegue este mensaje:
   `El nombre no está en el directorio telefónico actual`
2. Escriba un programa en C++ que contenga una lista vinculada de diez números enteros. Haga que el programa despliegue los números en la lista.
3. Usando la lista de estructuras vinculada ilustrada en la figura 13.13, escriba la secuencia de pasos necesaria para eliminar de la lista el registro de Edith Domínguez.
4. Generalice la descripción obtenida en el ejercicio 3 para describir la secuencia de pasos necesaria para eliminar la *n*-ésima estructura de una lista de estructuras vinculadas. La *n*-ésima estructura es precedida por la estructura $(n - 1)$ y es seguida por la estructura $(n + 1)$. Asegúrese de almacenar en forma correcta todos los valores apuntadores.
5. **a.** Una lista con vinculación doble es una lista en la que cada estructura contiene un apuntador tanto a la estructura siguiente como a la anterior en la lista. Defina un tipo apropiado para una lista con doble vinculación de nombres y números telefónicos.
   b. Usando el tipo definido en el ejercicio 5a, modifique el programa 13.9 para enlistar los nombres y números telefónicos en orden inverso.

13.5 ASIGNACIÓN DINÁMICA DE ESTRUCTURAS DE DATOS

Ya se ha encontrado el concepto de asignar y desasignar espacio de memoria en forma explícita usando los operadores new y delete (véase la sección 12.2). Por comodidad se repite la descripción de estos operadores en la tabla 13.1.

Tabla 13.1

Operator Name	Description
`new`	Reserves the number of bytes required by the requested data type. Returns the address of the first reserved location or `NULL`

Esta asignación dinámica de memoria es útil en especial cuando se aborda una lista de estructuras debido a que permite expandir la lista conforme se agregan nuevos registros y contraerla conforme se eliminan registros.

Al solicitar espacio de almacenamiento adicional, el usuario debe proporcionar a la función nueva una indicación de la cantidad de almacenamiento necesaria. Esto se hace solicitando suficiente espacio para un tipo de datos particular. Por ejemplo, la expresión `new(int)` o `new int` (las dos formas pueden usarse de manera intercambiable) solicita

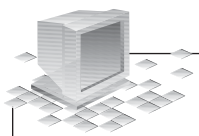
suficiente espacio de almacenamiento para almacenar un número entero. Una solicitud de almacenamiento suficiente para una estructura de datos se hace de la misma forma. Por ejemplo, usando la declaración

```
struct Tipo_tel
{
  string nombre;
  string num_telef;
};
```

las expresiones `newTipo_tel` y `new(Tipo_tel)` reservan suficiente almacenamiento para una estructura de datos `Tipo_tel`.

Al asignar almacenamiento en forma dinámica, no se tiene indicación anticipada del lugar en que el sistema de cómputo reservará físicamente el número de bytes solicitado, y no se tiene ningún nombre explícito para tener acceso a las ubicaciones de almacenamiento recién creadas. Para proporcionar acceso a estas ubicaciones, `new` devuelve la dirección de la primera ubicación que se ha reservado. Esta dirección debe ser asignada, por supuesto, a un apuntador. La devolución de un apuntador por `new` es útil en especial para crear una lista vinculada de estructuras de datos. Conforme se crea cada estructura nueva, el apuntador devuelto por `new` a la estructura puede asignarse a un miembro de la estructura previa en la lista.

El programa 13.10 ilustra el uso de `new` para crear una estructura en forma dinámica en respuesta a una solicitud de entrada del usuario.



Program 13.10

```
// a program illustrating dynamic structure allocation

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

struct TeleType
{
  string name;
  string phoneNo;
};

void populate(TeleType *);  // function prototype needed by main(
void dispOne(TeleType *);   // function prototype needed by main(
```

(Continued)

```
int main()
{
  char key;
  TeleType *recPoint;   // recPoint is a pointer to a
                        // structure of type TeleType

  cout << "Do you wish to create a new record (respond with y or n): ";
  key = cin.get();
  if (key == 'y')
  {
    key = cin.get();      // get the Enter key in buffered input
    recPoint = new TeleType;
    populate(recPoint);
    dispOne(recPoint);

  }
  else
    cout << "\nNo record has been created.";

  return 0;
}

  // input a name and phone number
void populate(TeleType *record) // record is a pointer to a
  {                             // structure of type TeleType
    cout << "Enter a name: ";
    getline(cin,record->name);
    cout << "Enter the phone number: ";
    getline(cin,record->phoneNo);

    return;
  }
  // display the contents of one record
void dispOne(TeleType *contents)  // contents is a pointer to a
{                                 // structure of type TeleType
  cout << "\nThe contents of the record just created is:"
       << "\nName: " << contents->name
       << "\nPhone Number: " << contents->phoneNo << endl;

  return;
}
```

Una sesión de muestra producida por el programa 13.10 es

```
Desea crear un registro nuevo (responda con s o n): s
Introduzca un nombre: Muñoz, Jaime
Introduzca el número telefónico: (555) 617-1817

El contenido del registro que se acaba de crear es:
Nombre: Muñoz, Jaime
Número telefónico: (555) 617-1817
```

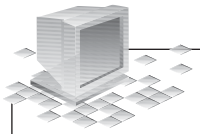
Al revisar el programa 13.10 se nota que sólo se hacen dos declaraciones de variables en `main()`. La variable clave se declara como una variable de carácter y la variable `regApunt` se declara como un apuntador a una estructura del tipo `Tipo_tel`. Debido a que la declaración para el tipo `Tipo_tel` es global, `Tipo_tel` puede usarse dentro de `main()` para definir `regApunt` como un apuntador a una estructura del tipo `Tipo_tel`.

Si un usuario introduce `s` en respuesta a la primera petición en `main()`, se hace una llamada a `new` por la memoria requerida para almacenar la estructura designada. Una vez que se ha cargado `regApunt` con la dirección apropiada, esta dirección puede usarse para tener acceso a la estructura recién creada. La función `poblar()` se usa para pedir al usuario los datos necesarios para llenar la estructura y almacenar los datos introducidos por el usuario en los miembros correctos de la estructura. El argumento transmitido a `poblar()` en `main()` es el apuntador `regApunt`. Como todos los argumentos transmitidos, el valor contenido en `regApunt` es transmitido a la función. Dado que el valor en `regApunt` es una dirección, `poblar()` recibe la dirección de la estructura recién creada y puede tener acceso directo a los miembros de la estructura.

Dentro de `poblar()`, el valor recibido se almacena en el argumento llamado `registro`. Dado que el valor que se almacenará en `registro` es la dirección de una estructura, `registro` debe declararse como un apuntador a una estructura. Esta declaración es proporcionada por la instrucción `Tipo_tel *registro;`. Las instrucciones dentro de `poblar()` usan la dirección en `registro` para localizar a los miembros respectivos de la estructura.

La función en `desplUno()` en el programa 13.10 se usa para desplegar el contenido de la estructura recién creada y poblada. La dirección transmitida a `desplUno()` es la misma dirección que se transmitió a `poblar()`. Debido a que este valor transmitido es la dirección de una estructura, el nombre del argumento usado para almacenar la dirección se declara como un apuntador al tipo de estructura correcto.

Una vez que se entiende el mecanismo de la llamada a `new`, puede usarse esta función para construir una lista de estructuras vinculadas. Como se describió en la sección anterior, las estructuras usadas en una lista vinculada deben contener al menos un miembro apuntador. La dirección en el miembro apuntador es la dirección inicial de la siguiente estructura en la lista. Además, debe reservarse un apuntador para la dirección de la primera estructura, y al miembro apuntador de la última estructura en la lista se le da una dirección `NULL` para indicar que no se apunta a más miembros. El programa 13.11 ilustra el uso de `new` para construir una lista vinculada de nombres y números telefónicos. La función `poblar()` usada en el programa 13.11 es la misma función usada en el programa 13.10, mientras que la función `desplegar()` es la misma función usada en el programa 13.9.



Program 13.11

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

const int MAXRECS = 3;   // maximum no. of records

struct TeleType
{
  char name;
  char phoneNo;
  TeleType *nextaddr;
};

void populate(TeleType *);   // function prototype needed by main(
void display(TeleType *);    // function prototype needed by main(

int main()
{
  int i;
  TeleType *list, *current; // two pointers to structures of
                            // type TeleType

    // get a pointer to the first structure in the list
  list = new TeleType;
  current = list;

    // populate the current structure and create the remaining str
  for(i = 0; i < MAXRECS - 1; i++)
  {
    populate(current);
    current->nextaddr = new TeleType;
    current = current->nextaddr;
  }

  populate(current);         // populate the last structure
  current->nextaddr = NULL;  // set the last address to a NULL add
  cout << "\nThe list consists of the following records:\n";
  display(list);   // display the structures

  return 0;
}
```

(Continued)

```
  // input a name and phone number
void populate(TeleType *record) // record is a pointer to a
{                               // structure of type TeleType

  cout << "Enter a name: ";
  getline(cin,record->name);
  cout << "Enter the phone number: ";
  getline(cin,record->phoneNo);

  return;
}

void display(TeleType *contents) // contents is a pointer to a
{                                      // structure of type TeleType
  while (contents != NULL)        // display till end of linked list
  {
    cout << endl << setiosflags(ios::left)
        << setw(30) << contents->name
        << setw(20) << contents->phoneNo;
    contents = contents->nextaddr;
  }
  cout << endl;

  return;
}
```

La primera vez que se llama a `new` en el programa 13.11 se usa para crear la primera estructura en la lista vinculada. Como tal, la dirección devuelta por `new` se almacena en la variable apuntadora llamada `lista`. La dirección en `lista` se asigna luego al apuntador llamado actual. Esta variable apuntadora siempre es usada por el programa para apuntar a la estructura actual. Debido a que la estructura actual es la primera estructura creada, la dirección en el apuntador llamado `lista` es asignada al apuntador llamado `actual`.

Dentro del ciclo `for` de `main()`, los miembros nombre y número telefónico de la estructura recién creada son poblados llamando a `poblar()` y transmitiendo la dirección de la estructura actual a la función. Con la devolución de `poblar()`, se asigna una dirección al miembro apuntador de la estructura `actual`. Esta dirección es la dirección de la siguiente estructura en la lista, la cual se obtiene de `new`. La llamada a `new` crea la siguiente estructura y devuelve su dirección al miembro apuntador de la estructura `actual`. Esto completa la población del miembro `actual`. La instrucción final en el ciclo `for` reinicia la dirección en el apuntador `actual` con la dirección de la siguiente estructura en la lista.

Después que se ha creado la última estructura, las instrucciones finales en `main()` pueblan esta estructura, asignando una dirección `NULL` al miembro apuntador, y llama a `desplegar()` para desplegar todas las estructuras en la lista. A continuación se proporciona una muestra de la ejecución del programa 13.11:

```
Introduzca un nombre: Amador, Samuel
Introduzca el número telefónico: (555) 898-2392
Introduzca un nombre: Domínguez, Edith
Introduzca el número telefónico: (555) 682-3104
Introduzca un nombre: López, Juan
Introduzca el número telefónico: (555) 718-4581

La lista consiste en los siguientes registros:

Amador, Samuel                (555) 898-2392
Domínguez, Edith              (555) 682-3104
López, Juan                   (555) 718-4581
```

Así como `new` crea almacenamiento en forma dinámica mientras se ejecuta un programa, la función `delete` regresa un bloque de almacenamiento a la computadora mientras el programa se ejecuta. El único argumento requerido por `delete` es la dirección inicial de un bloque de almacenamiento que fue asignado en forma dinámica. Por tanto, cualquier dirección devuelta por `new` puede transmitirse después a `delete` para regresar la memoria reservada a la computadora. La función `delete` no altera la dirección que se le transmite, sino tan sólo elimina el almacenamiento al que hace referencia la dirección.

Ejercicios 13.5

1. Como se describió en la tabla 13.1, el operador `new` devuelve la dirección de la primera área de almacenamiento nueva asignada o `NULL` si no se dispone de almacenamiento suficiente. Modifique el programa 13.11 para comprobar que se ha devuelto una dirección válida antes que se haga una llamada a `poblar()`. Despliegue un mensaje apropiado si no se dispone de almacenamiento suficiente.

2. Escriba una función en C++ llamada `eliminar()` que elimina una estructura existente de la lista de estructuras vinculadas creada por el programa 13.11. El algoritmo para eliminar una estructura vinculada deberá seguir la secuencia desarrollada para eliminar una estructura desarrollada en el ejercicio 4 en la sección 13.4. El argumento transmitido a `eliminar()` deberá ser la dirección de la estructura que precede al registro que se eliminará. En la función de eliminación, asegúrese que el valor del apuntador en la estructura eliminada reemplaza al valor del miembro apuntador de la estructura precedente antes que se elimine la estructura.

3. Escriba una función llamada `insertar()` que inserte una estructura en la lista de estructuras vinculadas creada en el programa 13.11. El algoritmo para insertar una estructura en una lista vinculada deberá seguir la secuencia para insertar un registro ilustrada antes en la figura 13.10. El argumento transmitido a `insertar()` deberá ser la dirección de la estructura que precede a la estructura que se va a insertar. La estructura insertada deberá seguir a esta estructura actual. La función `insertar()` deberá crear una estructura nueva en forma dinámica, llamar a la función `poblar` usada en el programa 13.11 y ajustar todos los valores apuntadores en forma apropiada.

4. Se desea insertar una estructura nueva en la lista de estructuras vinculadas creada por el programa 13.11. La función desarrollada para hacer esto en el ejercicio 3 suponía que se conocía la dirección de la estructura precedente. Escriba una función llamada `hallarReg()` que devuelva la dirección de la estructura inmediatamente precedente al punto en que se va a insertar la estructura nueva. (*Sugerencia:* `hallarReg()` debe solicitar el nuevo nombre como entrada y comparar el nombre introducido con los nombres existentes para determinar dónde colocar el nuevo nombre.)

5. Escriba una función en C++ llamada `modificar()` que pueda utilizarse para modificar los miembros nombre y número telefónico de una estructura del tipo creado en el programa 13.11. El argumento transmitido a `modificar()` deberá ser la dirección de la estructura que se va a modificar. La función `modificar()` deberá desplegar primero el nombre y número telefónico existentes en la estructura seleccionada y luego solicitar datos nuevos para estos miembros.

6. **a.** Escriba un programa en C++ que presente inicialmente un menú de opciones para el usuario. El menú deberá consistir en las siguientes opciones:

   Crear una lista vinculada inicial de nombres y números telefónicos
   Insertar una estructura nueva en la lista vinculada
   Modificar una estructura existente en la lista vinculada
   Eliminar de la lista una estructura existente
   Salir del programa

   Con la selección del usuario, el programa deberá ejecutar las funciones apropiadas para satisfacer la solicitud.

   b. ¿Por qué la creación original de una lista vinculada por lo general es realizada por un programa y las opciones para agregar, modificar o eliminar una estructura en la lista es proporcionada por un programa diferente?

13.6 UNIONES[2]

Una unión es un tipo de datos que reserva la misma área en la memoria para dos o más variables, cada una de las cuales puede ser un tipo de datos diferente. Una variable que se declara como un tipo de datos de unión puede usarse para contener una variable de carácter, una variable en número entero, una variable de precisión doble o cualquier otro tipo de datos válido en C++. Cada uno de estos tipos, pero sólo uno a la vez, puede asignarse en realidad a la variable unión.

La definición de una unión tiene la misma forma que una definición de estructura, con la palabra clave `union` usada en lugar de la palabra clave `struct`. Por ejemplo, la declaración

```
union
{
  char clave;
  int num;
  double voltios;
} val;
```

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

crea una variable de tipo unión llamada `val.` Si `val` fuera una estructura consistiría en tres miembros individuales. Sin embargo, como una unión, `val` contiene un solo miembro que puede ser una variable de carácter llamada `clave`, una variable en número entero llamada `num` o una variable en precisión doble llamada `voltios`. En efecto, una unión reserva suficientes ubicaciones de memoria para acomodar su tipo de datos del miembro más grande. Este mismo conjunto de ubicaciones es referenciado luego por diferentes nombres de variables, dependiendo del tipo de datos del valor que reside en la actualidad en las ubicaciones reservadas. Cada valor almacenado sobrescribe el valor anterior, usando tantos bytes del área de memoria reservada como sea necesario.

Los miembros individuales de la unión se referencian usando la misma notación que para los miembros de la estructura. Por ejemplo, si la unión `val` se está usando en la actualidad para almacenar un carácter, el nombre correcto de la variable para tener acceso al carácter almacenado es `val.clave`. Del mismo modo, si la unión se usa para almacenar un número entero, se tiene acceso al valor con el nombre `val.num` y se tiene acceso a un valor de precisión doble con el nombre `val.voltios`. Al usar miembros de unión, es responsabilidad del programador asegurar que se use el nombre de miembro correcto para el tipo de datos que reside en la actualidad en la unión.

Por lo general se usa una segunda variable para seguirle la pista al tipo de datos actual almacenado en la unión. Por ejemplo, podría usarse el siguiente código para seleccionar el miembro apropiado de `val` para desplegarlo. Aquí el valor en la variable `tipo_u` determina el tipo de datos almacenado en la actualidad en la unión `val`.

```
switch (tipo_u)
{
  case ‘c’: cout << val.clave;
            break;
  case ‘d’: cout << val.voltios;
            break;
  case 'd': cout << val.volts;
            break;
  default : cout << "Tipo inválido en tipo_u : " << tipo_u;
}
```

Como sucede con las estructuras, un tipo de datos puede asociarse con una unión. Por ejemplo, la declaración

```
union Fecha_tiempo
{
  int dias;
  double tiempo;

};
```

proporciona un tipo de datos unión sin reservar en realidad ninguna ubicación de almacenamiento. Este tipo de datos puede usarse entonces para definir cualquier número de variables. Por ejemplo, la definición

```
Fecha_tiempo primero, segundo, *pt;
```

crea una variable de unión llamada `primero`, una variable de unión llamada `segundo` y un apuntador que puede usarse para almacenar la dirección de cualquier unión que tenga la forma `Fecha_tiempo`. Una vez que se ha declarado un apuntador a una unión, la mis-

ma notación usada para tener acceso a los miembros de una estructura puede usarse para tener acceso a los miembros de la unión. Por ejemplo, si se hace la asignación `pt = &primero;`, entonces `pt->fecha` hace referencia al miembro `fecha` de la unión llamada primero.

Las uniones en sí pueden ser miembros de estructuras o arreglos. Estructuras, arreglos y apuntadores también pueden ser miembros de uniones. En cada caso, la notación usada para tener acceso a un miembro debe ser consistente con la anidación empleada. Por ejemplo, en la estructura definida por

```
struct
{
  char tipo_u;
  union
  {
    char *texto;
    float tasa;
  } impuesto_u;
}  bandera;
```

se hace referencia a la variable tasa como

```
bandera.impuesto_u.tasa
```

Del mismo modo, se hace referencia al primer carácter de la cadena cuya dirección está almacenada en el apuntador `texto` como

```
bandera.impuesto_u.texto
```

Ejercicios 13.6

1. Suponga que se ha hecho la siguiente definición

   ```
   union
   {
     double tasa;
     double impuestos;
     int num;
   } bandera;
   ```

   Para esta unión escriba activaciones de flujo `cout` apropiadas para desplegar los diversos miembros de la unión.

2. Defina una variable de unión llamada `auto` que contenga un número entero llamado `anio`, un arreglo de 10 caracteres llamado `nombre` y un arreglo de 10 caracteres llamado `modelo`.

3. Defina una variable de unión llamada `lang` que permitiría que se haga referencia a un número de precisión doble con los nombres de variables `voltios` y `fem`.

4. Declare un tipo de datos de unión llamado `cant` que contenga una variable en número entero llamada `cant_int`, una variable en precisión doble llamada `cant_doble` y un apuntador a un carácter llamado `ptClave`.

5. a. ¿Qué piensa que se desplegará con la siguiente sección de código?

```
union
{
  char caracter;
  double tipob;
} alt;
alt.caracter = 's';
cout << alt.tipob;
```

b. Incluya el código presentado en el ejercicio 5a en un programa y ejecute el programa para verificar su respuesta al ejercicio 5a.

13.7 ERRORES COMUNES DE PROGRAMACIÓN

Con frecuencia se cometen tres errores comunes cuando se usan estructuras o uniones. El primer error ocurre debido a que las estructuras y uniones, como entidades completas, no pueden usarse en expresiones relacionales. Por ejemplo, aun si `Tipo_tel` y `Tipo_aparato` son dos estructuras del mismo tipo, la expresión `Tipo_tel == Tipo_aparato` es inválida. Por supuesto, los miembros individuales de una estructura o unión pueden compararse si son del mismo tipo de datos, usando cualquiera de los operadores relacionales de C++.

El segundo error común en realidad es una extensión de un error de apuntador en su relación con las estructuras y uniones. Siempre que se usa un apuntador para "apuntar a" cualquiera de estos tipos de datos, o siempre que un apuntador es en sí mismo un miembro de una estructura o una unión, tenga cuidado de usar la dirección en el apuntador para tener acceso al tipo de datos apropiado. Si no se tiene claro a qué está apuntando, recuerde: "Si tiene dudas, imprímalo".

El error final se relaciona de manera específica con las uniones. Dado que una unión puede almacenar sólo uno de sus miembros a la vez, debe tener cuidado de seguirle la pista a la variable almacenada en la actualidad. Almacenar un tipo de datos en una unión y tener acceso a él con el nombre de variable equivocado puede producir un error que es particularmente problemático de localizar.

13.8 RESUMEN DEL CAPÍTULO

1. Una estructura permite que se agrupen variables individuales bajo un nombre de variable común. Se tiene acceso a cada variable en una estructura por su nombre de variable de estructura, seguido por un punto, seguido por su nombre de variable individual. Otro término para una estructura de datos es registro. Una forma para declarar una estructura es:

```
struct
{
  declaraciones de miembros individuales;
} nombreEstructura;
```

2. Puede crearse un tipo de datos a partir de una estructura usando la forma de declaración

```
struct TipoDatos
{
  declaraciones de miembros individuales;
};
```

   Las variables de estructura individuales pueden definirse entonces como este `TipoDatos`. Por convención, la primera letra del nombre `TipoDatos` siempre se escribe con mayúsculas.

3. Las estructuras son útiles en particular como elementos de arreglos. Usadas de esta manera, cada estructura se convierte en un registro en una lista de registros.

4. Pueden usarse estructuras completas como argumentos de función, en cuyo caso la función llamada recibe una copia de cada elemento en la estructura. También puede transmitirse la dirección de una estructura, ya sea como una referencia o como un apuntador, lo cual le proporciona a la función llamada acceso directo a la estructura.

5. Los miembros de la estructura puede ser cualquier tipo de datos válido en C++, incluyendo otras estructuras, uniones, arreglos y apuntadores. Cuando se incluye un apuntador como un miembro de la estructura, puede crearse una lista vinculada. Dicha lista usa el apuntador en una estructura para "apuntar a" (contener la dirección de) la siguiente estructura lógica en la lista.

6. Las uniones se declaran de la misma manera que las estructuras. La definición de una unión crea un área de memoria superpuesta, con cada miembro de la unión usando las mismas ubicaciones de almacenamiento de memoria. Por tanto, sólo un miembro de una unión puede estar activo a la vez.