15 Apuntadores y listas enlazadas

15 Apuntadores y listas enlazadas

Si hubiera la posibilidad de que alguien
Me amara en forma verdadera
Mi corazón me lo señalaría
Y yo te lo señalaría a ti.

GILBERT Y SULLIVAN, *Ruddigore*

Introducción

Una *lista enlazada* es una lista que se construye mediante el uso de apuntadores. No tiene un tamaño fijo, sino que puede aumentar y disminuir mientras un programa se ejecuta. En este capítulo le mostraremos cómo definir y manipular listas enlazadas, lo cual le servirá como presentación para una nueva forma de usar los apuntadores.

Prerrequisitos

En este capítulo se utiliza material de los capítulos 2 al 12.

15.1 Nodos y listas enlazadas

Es muy raro que las variables dinámicas útiles sean de un tipo simple como *int* o *double*, sino más bien son de algún tipo complejo como un arreglo, un tipo *struct* o un tipo de clase. Ya vimos que las variables dinámicas de un tipo de arreglo pueden ser útiles. Las variables dinámicas de un tipo *struct* o de clase también pueden ser útiles, pero de una manera distinta. Las variables dinámicas que son tipo *struct* o clases por lo general tienen una o más variables miembro que son variables de apuntador, con las que se conectan a otras variables dinámicas. Por ejemplo, una de esas estructuras (que contiene una lista de compras) se muestra en forma de diagrama en el cuadro 15.1.

Nodos

estructuras de nodos

Una estructura como la que se muestra en el cuadro 15.1 consiste de elementos que hemos dibujado como cuadros conectados por flechas. Los cuadros se llaman **nodos** y las flechas representan apuntadores. Cada uno de los nodos del cuadro 15.1 contiene una cadena, un entero y un apuntador que puede apuntar hacia otros nodos del mismo tipo. Observe que los apuntadores apuntan hacia todo el nodo completo, no hacia los elementos individuales (como `10` o `"rollos"`) que se encuentran dentro del nodo.

definición de tipo de nodo

En C++, los nodos se implementan como tipos *struct* o clases. Por ejemplo, las definiciones de tipo *struct* para un nodo del tipo que se muestra en el cuadro 15.1, junto con la definición del tipo para un apuntador hacia dichos nodos, puede ser la siguiente:

```
struct NodoLista
{
    string elemento;
    int cuenta;
    NodoLista *enlace;
};

typedef  NodoLista*  NodoListaPtr;
```

El orden de las definiciones de tipo es importante. La definición de `NodoLista` debe ir primero, ya que se utiliza en la definición de `NodoListaPtr`.

El cuadro etiquetado como `cabeza` en el cuadro 15.1 no es un nodo, sino una variable de apuntador que puede apuntar hacia un nodo. La variable de apuntador `cabeza` se declara de la siguiente manera:

```
NodoListaPtr  cabeza;
```

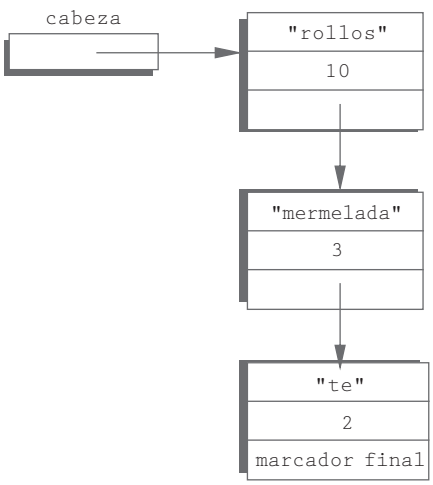
Aunque hemos ordenado las definiciones de tipos para evitar ciertas formas ilegales de circularidad, es evidente que la definición del `NodoLista` tipo *struct* sigue siendo circular. La definición del tipo `NodoLista` utiliza el nombre de tipo `NodoLista` para definir la variable miembro `enlace`. No hay nada malo con esta circularidad específica, y está permitida en C++. Una indicación de que esta definición no es inconsistente en forma lógica es el hecho de que podemos hacer dibujos, como el cuadro 15.1, que representan a dichas estructuras.

modificación de los datos de un nodo

Ahora tenemos apuntadores dentro de tipos *struct* y hacemos que apunten hacia tipos *struct* que contienen apuntadores, y así sucesivamente. En tales situaciones la sintaxis puede ser un poco complicada, pero en todos los casos sigue las reglas que hemos descrito para los apuntadores y los tipos *struct*. Por ejemplo, supongamos que las declaraciones son como las anteriores, que la situación es igual que el diagrama del cuadro 15.1 y queremos modificar el número en el primer nodo de `10` a `12`. Una manera de lograr esto es con la siguiente instrucción:

```
(*cabeza).cuenta = 12;
```

CUADRO 15.1 *Nodos y apuntadores*



La expresión del lado izquierdo del operador de asignación podría requerir un poco de explicación. La variable `cabeza` es una variable apuntador. Por lo tanto, la expresión `*cabeza` es el objeto al cual apunta, a saber el nodo (variable dinámica) que contiene `"rollos"` y el entero `10`. Este nodo (al cual se hace referencia como `*cabeza`) es un tipo *`struct`* y la variable miembro de esta *`struct`*, que contiene un valor de tipo *`int`*, se llama `cuenta`; por lo tanto, `(*cabeza).cuenta` es el nombre de la variable *`int`* en el primer nodo. Los paréntesis alrededor de `*cabeza` no son opcionales. Queremos que el operador de desreferencia `*` se aplique antes que el operador punto. No obstante, el operador punto tiene mayor precedencia que el operador de desreferencia `*`, por lo que sin los paréntesis se aplicaría primero el operador punto (y eso produciría un error). En el siguiente párrafo describiremos una notación abreviada que puede evitar el tener que preocuparnos por los paréntesis.

el operador −>

C++ cuenta con un operador que puede usarse con un apuntador para simplificar la notación para especificar los miembros de una *`struct`* o una clase. El **operador flecha −>** combina las acciones de un operador de desreferencia y de un operador punto para especificar un miembro de un tipo *`struct`* dinámico u objeto que está siendo apuntado por un apuntador dado. Por ejemplo, la instrucción de asignación anterior para modificar el número en el primer nodo podría escribirse de una manera más simple como:

```
cabeza->cuenta = 12;
```

Esta instrucción de asignación y la anterior significan lo mismo, pero ésta es la forma que se utiliza con más frecuencia.

La cadena en el primer nodo puede modificarse de "rollos" a "bagels" mediante la siguiente instrucción:

```
cabeza->elemento = "bagels";
```

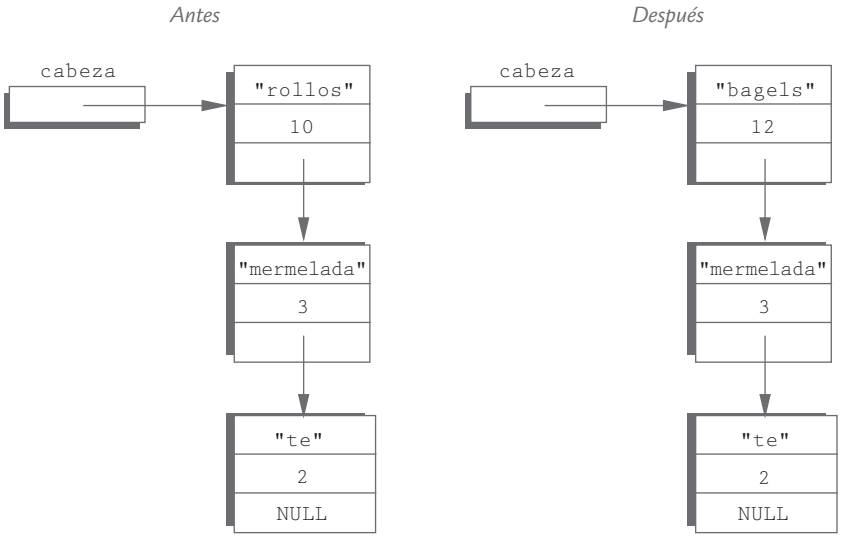
El resultado de estas modificaciones al primer nodo en la lista se muestra como diagrama en el cuadro 15.2

Vea el miembro apuntador en el último nodo de las listas que se muestran en el cuadro 15.2 Este último nodo tiene la palabra NULL escrita en donde debería estar un apuntador. En el cuadro 15.1 llenamos esta ubicación con la frase "marcador final", pero ésta no es una expresión de C++. En los programas de C++ utilizamos la constante NULL como un marcador final para indicar la expresión final. NULL es una constante definida especial que forma parte del lenguaje C++ (se proporciona como parte de las bibliotecas requeridas de C++).

Por lo general, NULL se utiliza para dos propósitos distintos (pero que a menudo coinciden). Se utiliza para proporcionar un valor a una variable apuntador que de otra forma no tendría ningún valor. Esto evita una referencia inadvertida a la memoria, ya que NULL no es la dirección de ninguna ubicación de memoria. La segunda categoría de uso es la de un marcador final. Un programa puede recorrer la lista de nodos como se muestra en el cuadro 15.2, y cuando llegue al nodo que contiene NULL, sabrá que ha llegado al final de la lista.

CUADRO 15.2 *Acceso a los datos de un nodo*

```
head->count = 12;
cabeza->elemento = "bagels";
```



El operador flecha ->

El operador flecha -> especifica a un miembro de un tipo *struct* (o a un miembro del objeto de una clase) que es apuntado por una variable apuntador. La sintaxis es la siguiente:

Variable_apuntador->Nombre_miembro

Lo anterior se refiere a un miembro del tipo *struct* u objeto al que apunta *Variable_apuntador*. *Nombre_miembro* indica a cuál miembro hace referencia. Por ejemplo, suponga que tiene la siguiente definición:

```
struct Registro
{
   int numero;
   char calificacion;
};
```

El siguiente código crea una variable dinámica de tipo `Registro` y asigna los valores `2001` y `'A'` a las variables de la variable *struct* dinámica:

```
Registro *p;

p = new Registro;
p->numero = 2001;
p->calificacion = 'A';
```

NULL es 0

La constante `NULL` es en realidad el número 0, pero es preferible pensar en ella y escribirla como `NULL`. Esto hace más evidente que nos estamos refiriendo a este valor de propósito especial que podemos asignar a variables tipo apuntador. La definición del identificador `NULL` se encuentra en varias de las bibliotecas estándar, como `<iostream>` y `<cstddef>`, por lo que debemos utilizar una directiva `include` ya sea con `<iostream>` o con `<cstddef>` (o con cualquier otra biblioteca adecuada) cuando utilicemos `NULL`. No se necesita ninguna directiva *using* para poder hacer que `NULL` esté disponible para el código de nuestros programas. En especial no se requiere *using namespace* `std;`, aunque es probable que otros elementos en el código requieran algo como *using namespace* `std;`.[1]

A un apuntador se le puede asignar `NULL` mediante el operador de asignación, como en el siguiente código, que declara una variable apuntador llamada `there` y la inicializa con `NULL`:

```
double *there = NULL;
```

Se puede asignar la constante `NULL` a una variable apuntador de cualquier tipo de apuntador.

NULL

`NULL` es un valor constante especial que se utiliza para proporcionar un valor a una variable apuntador que, de otra forma, no tendría uno. `NULL` puede asignarse a una variable apuntador de cualquier tipo. El identificador `NULL` está definido en varias bibliotecas, incluyendo la biblioteca con el archivo de encabezado `<cstddef>` y la biblioteca con el archivo de encabezado `<iostream>`. La constante `NULL` es en realidad el número 0, pero es preferible pensar en ella y escribirla como `NULL`.

[1] Los detalles son los siguientes: la definición de `NULL` se maneja mediante el preprocesador C++, la cual sustituye a `NULL` con `0`. Así, como el compilador en realidad nunca ve `"NULL"` no hay problemas con el espacio de nombres, por lo cual no se necesita la directiva *using*.

Ejercicios de AUTOEVALUACIÓN

1. Suponga que su programa contiene las siguientes definiciones de tipos:

```
struct Caja
{
     string nombre;
     int numero;
     Caja *siguiente;
};

typedef Caja* CajaPtr;
```

¿Cuál es la salida que produce el siguiente código?

```
CajaPtr cabeza;
cabeza = new Caja;
cabeza->nombre = "Sally";
cabeza->numero = 18;
cout << (*cabeza).nombre << endl;
cout << cabeza->nombre << endl;
cout << (*cabeza).numero << endl;
cout << cabeza->numero << endl;
```

2. Suponga que su programa contiene las definiciones de tipos y el código que se muestra en el ejercicio de autoevaluación 1. Ese código crea un nodo que contiene el objeto `string "Sally"` y el número `"18"`. ¿Qué código agregaría para poder asignar `NULL` al valor de la variable miembro `siguiente` de este nodo?

3. Suponga que su programa contiene las definiciones de tipos y el código que se muestra en el ejercicio de autoevaluación 1. Si suponemos que el valor de la variable apuntador `cabeza` no se ha modificado, ¿cómo podemos destruir la variable dinámica a la que apunta `cabeza` y devolver la memoria que utiliza al almacén de memoria libre, para que pueda reutilizarse para crear nuevas variables dinámicas?

4. Dada la siguiente definición de una estructura:

```
struct NodoLista
{
    string elemento;
    int cuenta;
    NodoLista *enlace;
};
NodoLista *cabeza = new NodoLista;
```

Escriba un código para asignar la cadena "Orville el hermano de Wilbur" al miembro `elemento` del nodo al que apunta `cabeza`.

Listas enlazadas

lista enlazada

Las listas como las que se muestran en el cuadro 15.2 se llaman *listas enlazadas*. Una **lista enlazada** es una lista de nodos en la que cada nodo tiene una variable miembro que es un apuntador, el cual apunta al siguiente nodo en la lista. El primer nodo en una lista enlazada se llama **cabeza**; ésta es la razón por la que la variable apuntador que apunta al primer nodo se llama `cabeza`. Hay que tener en cuenta que el apuntador llamado `cabeza` no es en

cabeza

sí la cabeza de la lista, sino que sólo apunta a la cabeza de la lista. El último nodo no tiene un nombre especial, pero sí una propiedad especial. El último nodo tiene NULL como el valor de su variable apuntador miembro. Para verificar si un nodo es el último, sólo hay que ver si la variable apuntador en el nodo es igual a NULL.

definición de tipo de nodo

Nuestro objetivo en esta sección es escribir algunas funciones básicas para manipular listas enlazadas. Por cuestión de variedad y para simplificar la notación, utilizaremos un tipo de nodo más simple que el del cuadro 15.2. Estos nodos sólo contendrán un entero y un apuntador. Las definiciones de nodo y de tipo de apuntador que utilizaremos son las siguientes:

```
struct Nodo
{
    int datos;
    Nodo *enlace;
}

typedef Nodo* NodoPtr;
```

una lista enlazada de un nodo

Como ejercicio de calentamiento, veamos cómo podríamos construir el inicio de una lista enlazada con nodos de este tipo. Primero vamos a declarar una variable apuntador llamada cabeza, que apunte a la cabeza de nuestra lista enlazada:

```
NodoPtr cabeza;
```

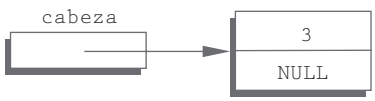
Para crear nuestro primer nodo utilizamos el operador *new* para crear una nueva variable dinámica que se convertirá en el primer nodo de nuestra lista enlazada.

```
cabeza = new Nodo;
```

Ahora asignamos valores a las variables miembro de este nuevo nodo:

```
cabeza->datos = 3;
cabeza->enlace = NULL;
```

Observe que el miembro apuntador de este nodo se hace igual a NULL. Esto se debe a que este nodo es el último de la lista (así como el primer nodo en la lista). En esta etapa, nuestra lista enlazada se ve así:



Esta lista de un nodo se creó de manera intencional. Para tener una lista más grande, nuestro programa debe ser capaz de agregar nodos de una manera sistemática. A continuación describiremos una forma de insertar nodos en una lista enlazada.

Inserción de un nodo en la cabeza de una lista

En esta subsección supondremos que nuestra lista enlazada ya contiene uno o más nodos, y desarrollaremos una función para agregar otro nodo. El primer parámetro para la función de inserción será un parámetro de llamada por referencia para una variable apuntador que apunta a la cabeza de la lista enlazada, es decir, una variable apuntador que apunta al pri-

mer nodo en la lista enlazada. El otro parámetro proporcionará el número que se almacenará en el nuevo nodo. La declaración de nuestra función de inserción es la siguiente:

```
void insercion_cabeza(NodoPtr& cabeza, int el_numero);
```

Listas enlazadas como argumentos

Siempre hay que mantener una variable apuntador que apunte a la cabeza de una lista enlazada. Esta variable apuntador es una forma de nombrar la lista enlazada. Cuando se escribe una función que toma una lista como argumento, podemos usar este apuntador (que apunta a la cabeza de la lista enlazada) como argumento de lista enlazada.

Para insertar un nuevo nodo en la lista enlazada, nuestra función utilizará el operador *new* para crear un nuevo nodo. A continuación los datos se copiarán en el nuevo nodo y éste se insertará en la cabeza de la lista. Cuando insertamos nodos de esta forma, el nuevo nodo pasa a ser el primer nodo en la lista (es decir, el nodo cabeza) en vez de ser el último nodo. Como las variables dinámicas no tienen nombres, debemos utilizar una variable apuntador local para apuntar a este nodo. Si a esta variable apuntador local la llamamos `temp_ptr`, se puede hacer referencia al nuevo nodo como `*temp_ptr`. El proceso completo puede resumirse de la siguiente manera:

algoritmo

Pseudocódigo para la función `insercion_cabeza`

1. Crear una nueva variable dinámica; `temp_ptr` va a apuntar hacia esta variable. (Esta nueva variable dinámica será el nuevo nodo. Podemos referirnos a este nuevo nodo como `*temp_ptr`.)
2. Colocar los datos en este nuevo nodo.
3. Hacer que el miembro `enlace` de este nuevo nodo apunte hacia el nodo cabeza (primer nodo) de la lista enlazada original.
4. Hacer que la variable apuntador llamada `cabeza` apunte hacia el nuevo nodo.

El cuadro 15.3 contiene un diagrama de este algoritmo. Los pasos 2 y 3 del diagrama pueden expresarse mediante estas instrucciones de asignación en C++:

```
temp_ptr->enlace = cabeza;
cabeza = temp_ptr;
```

En el cuadro 15.4 se proporciona la definición de la función completa.

Es conveniente tener la posibilidad de que una lista no contenga nada. Por ejemplo, una lista de compras podría no contener nada debido a que no hay nada que comprar esta semana. Una lista que no contiene nada se llama **lista vacía**. Para nombrar una lista enlazada se nombra un apuntador que apunta hacia la cabeza de la lista, pero una lista vacía no tiene nodo cabeza. Para especificar una lista vacía se utiliza el apuntador `NULL`. Si la variable apuntador `cabeza` debe apuntar hacia el nodo cabeza de una lista enlazada y queremos indicar que la lista está vacía, entonces hay que establecer el valor de la `cabeza` de la siguiente manera:

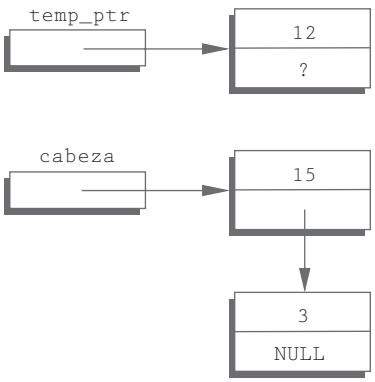
lista vacía

```
cabeza = NULL;
```

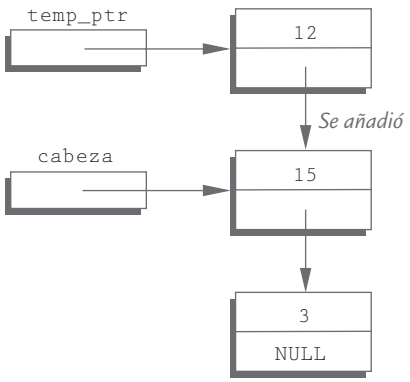
Cada vez que diseñe una función para manipular una lista enlazada, siempre deberá comprobar si funciona con la lista vacía. Si no funciona, tal vez pueda agregar un caso

CUADRO 15.3 *Cómo agregar un nodo a una lista enlazada*

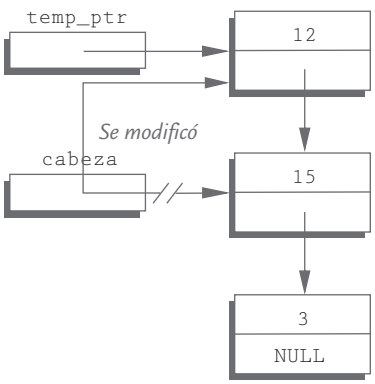
1. Se configura el nuevo nodo



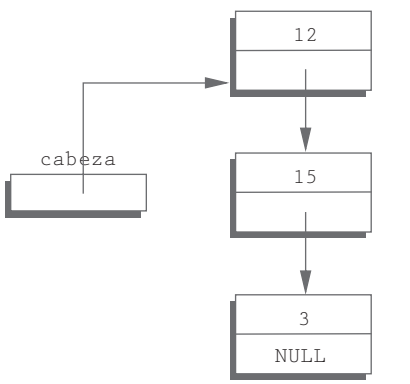
2. temp_ptr->enlace = cabeza;



3. cabeza = temp_ptr;



4. Después de la llamada a la función



CUADRO 15.4 *Cómo agregar un nodo a una lista enlazada*

Declaración de la función

```
struct Nodo
{
      int datos;
      Nodo *enlace;
};

typedef Nodo* NodoPtr;

void insercion_cabeza(NodoPtr& cabeza, int el_numero);
//Precondición: la variable apuntador cabeza apunta hacia
//la cabeza de una lista enlazada.
//Postcondición: se agregó un nuevo nodo que contiene
//el_numero a la cabeza de la lista enlazada.
```

Definición de la función

```
void insercion_cabeza(NodoPtr& cabeza, int el_numero)
{
      NodoPtr temp_ptr;
      temp_ptr = new Nodo;

      temp_ptr->datos = el_numero;

      temp_ptr->enlace = cabeza;
      cabeza = temp_ptr;

}
```

especial para la lista vacía. Si no puede diseñar la función para aplicarla a la lista vacía, entonces debe diseñar su programa de tal forma que maneja las listas vacías de alguna otra manera o que las evite por completo. Por fortuna, la lista vacía puede tratarse casi siempre como cualquier otra lista. Por ejemplo, la función `insercion_cabeza` del cuadro 15.4 se diseñó con listas no vacías como modelo, pero una comprobación nos mostrará que también funciona para la lista vacía.

RIESGO *Pérdida de nodos*

Tal vez se vea tentado a escribir la definición de la función `insercion_cabeza` (cuadro 15.4) mediante el uso de la variable apuntador cabeza para crear el nuevo nodo, en vez de usar la variable apuntador local `temp_ptr`. Si tratara de hacer esto, podría empezar la función de la siguiente manera:

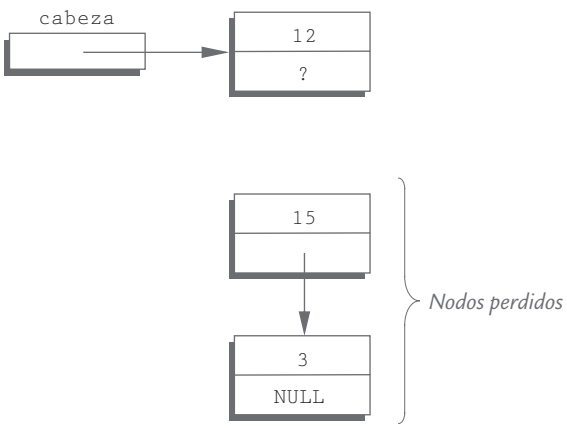
```
cabeza = new Nodo;
cabeza->datos = el_numero;
```

En este punto se crea el nuevo nodo, el cual contiene los datos correctos y la cabeza apuntador apunta a él, como todo se supone que debe ser. Todo lo que queda por hacer es unir el resto de la lista a este nodo mediante la configuración del miembro apuntador que se proporciona a continuación, de manera que apunte hacia lo que antes era el primer nodo de la lista:

```
cabeza->enlace
```

El cuadro 15.5 muestra la situación cuando el valor de los nuevos datos es `12`. Esa ilustración revela el problema. Si fuera a proceder de esta manera, no habría nada apuntando hacia el nodo con el valor de `15`. Ya que no hay un apuntador con nombre que apunte a este nodo (o a una cadena de apuntadores que terminen con ese nodo), no hay manera en que el programa pueda hacer referencia a él. El nodo que está por debajo de este nodo también se perderá. Un programa no puede hacer que un apuntador apunte hacia uno de estos nodos, ni puede acceder a los datos en estos nodos, ni puede hacer cualquier otra cosa con estos nodos. En resumen, no tiene manera de hacer referencia a estos nodos.

CUADRO 15.5 *Nodos perdidos*



Una situación así retiene memoria durante el tiempo que se ejecuta el programa. Algunas veces se dice que un programa que pierde nodos tiene una "fuga de memoria". Una fuga considerable de memoria puede hacer que el programa se quede sin memoria, lo cual produce una terminación anormal. Lo que es peor, una fuga de memoria (nodos perdidos) en un programa de usuario común puede hacer que el sistema operativo falle. Para evitar los nodos perdidos, un programa siempre debe mantener un apuntador que apunte hacia la cabeza de la lista, que por lo general viene siendo un apuntador en una variable apuntador tal como `cabeza`.

Búsqueda en una lista enlazada

Ahora diseñaremos una función para buscar por orden en una lista enlazada, para localizar un nodo específico. Utilizaremos el mismo tipo de nodo (llamado `Nodo`) que en las subsecciones anteriores. (En el cuadro 15.4 se proporciona la definición del nodo y los tipos de apuntadores.) La función que diseñaremos tendrá dos argumentos: la lista enlazada y el entero que deseamos localizar. La función devolverá un apuntador que apunte hacia el primer nodo que contenga ese entero. Si ningún nodo contiene el entero, la función devolverá el apuntador `NULL`. De esta forma, nuestro programa puede comprobar si el entero está en la lista al ver si la función devuelve un valor de apuntador que sea distinto de `NULL`. La declaración de la función y el comentario del encabezado para nuestra función es el siguiente:

```
NodoPtr busca(NodoPtr cabeza, int objetivo);
//Precondición: la cabeza apuntador apunta hacia la cabeza de
//una lista enlazada. La variable apuntador en el último nodo
//es NULL. Si la lista está vacía, entonces la cabeza es NULL.
//Devuelve un apuntador que apunta hacia el primer nodo que
//contiene el objetivo. Si ningún nodo contiene el objetivo,
//la función devuelve NULL.
```

Emplearemos una variable apuntador local llamada `aqui` para desplazarnos a través de la lista y buscar el `objetivo`. La única forma de desplazarse alrededor de una lista enlazada, o de cualquier otra estructura compuesta de nodos y apuntadores, es mediante el seguimiento de los apuntadores. Por lo tanto, comenzaremos haciendo que `aqui` apunte hacia el primer nodo y desplazaremos el apuntador de nodo en nodo, siguiendo el apuntador al salir de cada nodo. En el cuadro 15.6 se muestra un diagrama de esta técnica. Como las listas vacías presentan ciertos problemas pequeños que complicarían nuestra discusión, al principio supondremos que la lista enlazada contiene por lo menos un nodo. Después regresaremos para asegurarnos que el algoritmo funcione también para la lista vacía. Esta técnica de búsqueda produce el siguiente algoritmo:

algoritmo

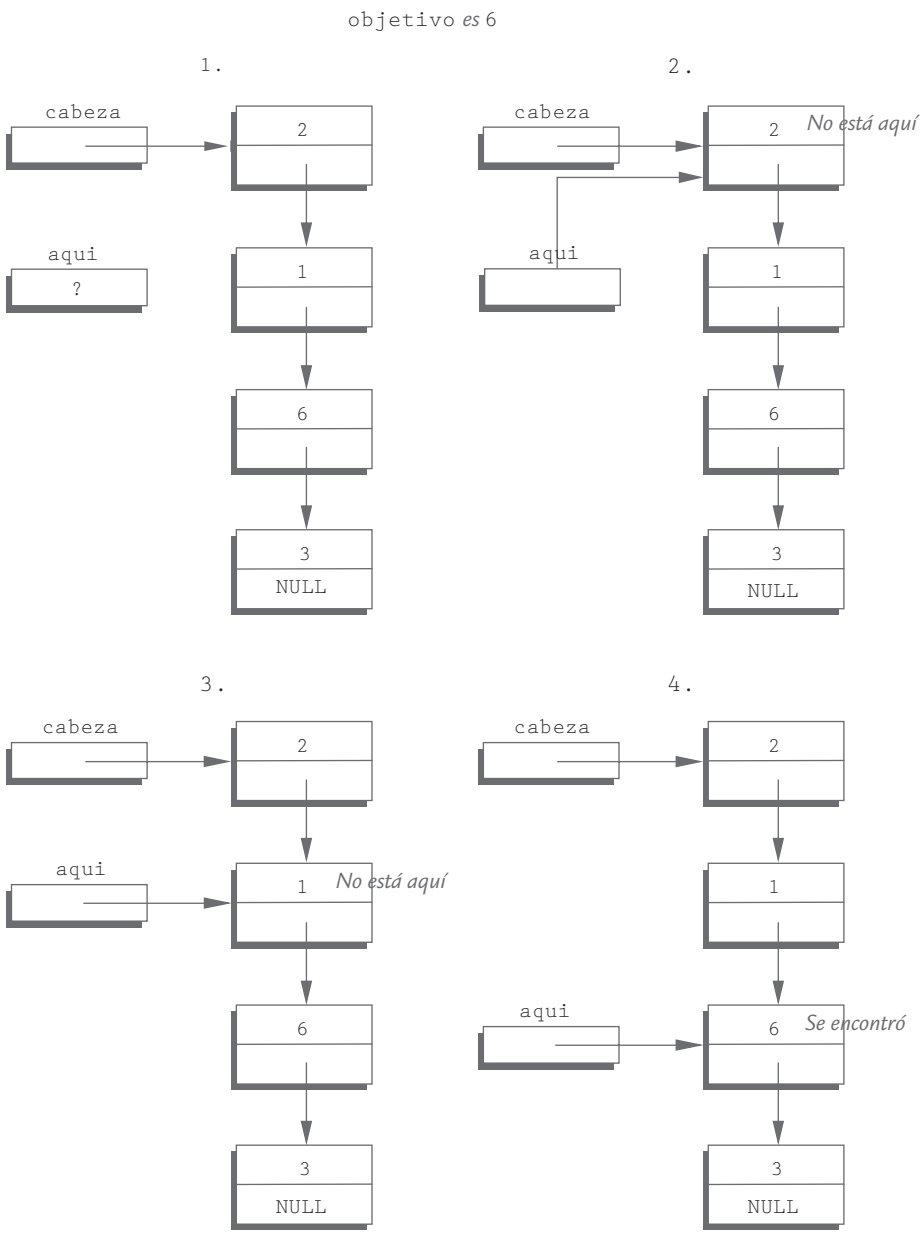
Pseudo-código para la función `busca`

1. Asegurarse que la variable apuntador `aqui` apunte hacia el nodo cabeza (es decir al primer nodo) de la lista enlazada.

```
while (aqui no esté apuntando a un nodo que contenga objetivo
                                  y aqui no esté apuntando al último nodo)
{
    Hacer que aqui apunte al siguiente nodo en la lista.
}

if (el nodo al que apunta aqui contiene objetivo)
    return aqui;
else
    return NULL;
```

CUADRO 15.6 *Búsqueda en una lista enlazada*



Para poder desplazar el apuntador `aqui` hacia el siguiente nodo, debemos pensar en términos de los apuntadores con nombre que tenemos disponibles. El siguiente nodo es al que apunta el miembro apuntador del nodo al que `aqui` apunta en ese momento. El miembro apuntador del nodo al que apunta `aqui` en ese momento se proporciona mediante la expresión

```
aqui->enlace
```

Para desplazar `aqui` al siguiente nodo tenemos que modificar `aqui` de manera que apunte al nodo que está siendo apuntado por la variable apuntador (miembro) con el nombre que se menciona arriba. Por ende, el siguiente código desplazará `aqui` hacia el siguiente nodo en la lista:

```
aqui = aqui->enlace;
```

Si unimos estas piezas se produce la siguiente refinación del pseudo-código del algoritmo:

refinación del algoritmo

```
Versión preliminar del código para la función busca

    aqui = cabeza;
    while (aqui->datos != objetivo &&
                                aqui->enlace != NULL)
        aqui = aqui->enlace;
    if (aqui->datos == objetivo)
        return aqui;
    else
        return NULL;
```

Observe la expresión Booleana en la instrucción *while*. Para evaluar si `aqui` no apunta al último nodo debemos evaluar si la variable miembro `aqui->enlace` es igual a `NULL`.

lista vacía

Aún debemos regresar y hacernos cargo de la lista vacía. Si revisamos nuestro código encontraremos que hay un problema con la lista vacía. Si la lista está vacía, entonces `aqui` es igual a `NULL` y, por lo tanto, las siguientes expresiones están indefinidas:

```
aqui->datos
aqui->enlace
```

Cuando `aqui` es `NULL` no está apuntando a ningún nodo, por lo que no hay un miembro llamado `datos` ni un miembro llamado `enlace`. Por lo tanto, podemos hacer de la lista vacía un caso especial. En el cuadro 15.7 se da la definición completa de la función.

Los apuntadores como iteradores

iterador

Un **iterador** es una construcción que nos permite desplazarnos en forma cíclica a través de los elementos de datos almacenados en una estructura de datos, de manera que podamos realizar cualquier tipo de acción sobre cada elemento de datos. Un iterador puede ser un objeto de alguna clase de iterador o algo más simple, como un índice de arreglo o un apuntador. Los apuntadores son un ejemplo simple de un iterador. De hecho, un apuntador es el ejemplo prototípico de un iterador. Las ideas básicas pueden verse con facilidad en el contexto de las listas enlazadas. Podemos utilizar un apuntador como iterador si lo despla-

CUADRO 15.7 *Función para localizar un nodo en una lista enlazada*

Declaración de la función

```
struct Nodo
{
      int datos;
      Nodo *enlace;
};

typedef Nodo* NodoPtr;

NodoPtr busca(NodoPtr cabeza, int objetivo);
//Precondición: El apuntador cabeza apunta a la cabeza de
//una lista enlazada. La variable apuntador en el último nodo
//es NULL. Si la lista está vacía, entonces cabeza es NULL.
//Devuelve un apuntador que apunta al primer nodo que
//contiene el objetivo. Si ninguno de los nodos contiene el objetivo,
//la función devuelve NULL.
```

Definición de la función

```
//Usa cstddef;
NodoPtr busca(NodoPtr cabeza, int objetivo)
{
      NodoPtr aqui = cabeza;

      if (aqui == NULL)
      {
          return NULL;
      }
      else
          while (aqui->datos != objetivo &&
                                    aqui->enlace != NULL)
             aqui = aqui->enlace;

          if (aqui->datos == objetivo)
             return aqui;
       else
             return NULL;
      }
}
```

Caso de lista vacía

zamos a través de la lista enlazada un nodo a la vez, empezando en la cabeza de la lista y recorriendo en forma cíclica todos los nodos de la misma. El bosquejo general sería el siguiente:

```
Tipo_Nodo *iter;
for (iter = Cabeza;  iter != NULL;  iter = iter->Enlace)
   Haga lo que desee con el nodo al que apunta iter;
```

en donde *Cabeza* es un apuntador hacia el nodo cabeza de la lista enlazada y *Enlace* es el nombre de la variable miembro de un nodo que apunta al siguiente nodo en la lista.

Por ejemplo, para mostrar los datos en todos los nodos de una lista enlazada del tipo que hemos estado viendo, podríamos utilizar

```
NodoPtr iter; //Equivalente a: Nodo *iter;
for (iter = cabeza;  iter != NULL;  iter = iter->Enlace)
    cout << (iter->datos);
```

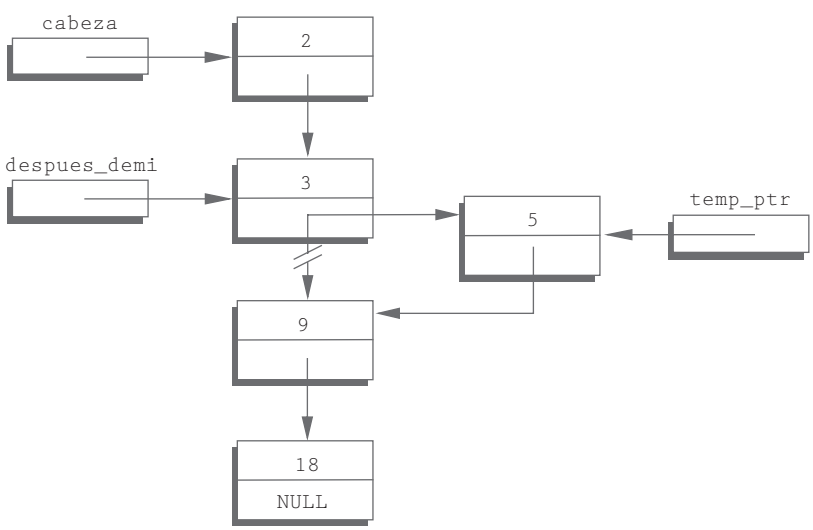
Las definiciones de `Nodo` y `NodoPtr` se proporcionan en el cuadro 15.7.

Inserción y remoción de nodos dentro de una lista

inserción en medio de una lista

A continuación diseñaremos una función para insertar un nodo en una posición específica de una lista enlazada. Si deseamos los nodos en cierto orden tal como el orden numérico o alfabético, no podemos sólo insertar el nodo al principio o al final de la lista. Por lo tanto diseñaremos una función para insertar un nodo después de cierto nodo específico en la lista enlazada. Supongamos que cierta función o parte del programa ha colocado en forma correcta un apuntador llamado `despues_demi` que apunta a cierto nodo en la lista enlazada. Queremos que el nuevo nodo se coloque después del nodo al que apunta `despues_demi`, como se muestra en el cuadro 15.8. La misma técnica funciona para los nodos con cualquier tipo de datos, pero para ser concretos utilizaremos el mismo tipo de

CUADRO 15.8 *Inserción en medio de una lista enlazada*



nodos que en las subsecciones anteriores. En el cuadro 15.7 se proporcionan las definiciones de tipo. La declaración de la función que deseamos definir es:

```
void inserta(NodoPtr despues_demi, int el_numero);
//Precondición: despues_demi apunta a un nodo en una lista enlazada.
//Postcondición: se ha agregado un nuevo nodo que contiene el_numero
//después del nodo al que apunta despues_demi.
```

Un nuevo nodo se establece de la misma forma que en la función `insercion_cabeza` del cuadro 15.4. La diferencia entre esta función y la del cuadro 15.4 es que ahora deseamos insertar el nodo no en la cabeza de la lista, sino después del nodo al que apunta `despues_demi`. En el cuadro 15.8 se muestra cómo realizar la inserción, que se expresa en código de C++ a continuación:

```
//agrega un enlace del nuevo nodo a la lista:
temp_ptr->enlace = despues_demi->enlace;
//agrega un enlace de la lista al nuevo nodo:
despues_demi->enlace = temp_ptr;
```

El orden de estas dos instrucciones de asignación es crucial. En la primera asignación queremos el valor del apuntador `despues_demi->enlace` *antes de que se modifique*. En el cuadro 15.9 se muestra la función completa.

inserción en los extremos

Si analiza el código para la función `inserta`, descubrirá que funciona en forma correcta aún si el nodo al que apunta `despues_demi` es el último nodo en la lista. No obstante, `inserta` no funcionará para insertar un nodo al principio de una lista enlazada. La función `insercion_cabeza` que se muestra en el cuadro 15.4 puede usarse para insertar un nodo al principio de una lista.

comparación con los arreglos

Mediante el uso de la función `inserta` podemos mantener una lista enlazada en orden numérico, alfabético o cualquier otro tipo de orden. Podemos insertar un nodo en la posición correcta con sólo ajustar dos apuntadores. Esto es cierto sin importar qué tan extensa sea la lista enlazada o en dónde se deseen insertar los nuevos datos. Si utilizáramos un arreglo en vez de una lista enlazada, la mayoría (y en casos extremos todo) del arreglo tendría que copiarse para poder hacer espacio para un nuevo valor en el lugar correcto. A pesar de la sobrecarga implicada en la operación de posicionar el apuntador `despues_demi`, por lo general la inserción en una lista enlazada es más eficiente que la inserción en un arreglo.

remoción de un nodo

También es muy sencillo remover un nodo de una lista enlazada. El cuadro 15.10 muestra el método. Una vez que se han posicionado los apuntadores `antes` y `descarta`, todo lo que se requiere para remover el nodo es la siguiente instrucción:

```
antes->enlace = descarta->enlace;
```

Esto es suficiente para remover el nodo de la lista enlazada. Ahora que, si no planea utilizar este nodo para algo más, debe destruirlo y regresar la memoria que utiliza al almacén de memoria libre; puede hacer esto mediante una llamada a `delete`, como se muestra a continuación:

```
delete descarta;
```

CUADRO 15.9 *Función para agragar un nodo en medio de una lista enlazada*

Declaración de la función

```
struct Nodo
{
    int datos;
    Nodo *enlace;
};

typedef Nodo* NodoPtr;

void inserta(NodoPtr despues_demi, int el_numero);
//Precondición: despues_demi apunta a un nodo en una lista
//enlazada.
//Postcondición: se ha agregado un nuevo nodo que contiene
//el_numero después del nodo al que apunta despues_demi.
```

Definición de la función

```
void inserta(NodoPtr despues_demi, int el_numero);
{
    NodoPtr temp_ptr;
    temp_ptr = new Nodo;

    temp_ptr->datos = el_numero;

    temp-ptr->enlace = despues_demi->enlace;
    despues_demi->enlace = temp_ptr;
}
```

RIESGO *Uso del operador de asignación con estructuras dinámicas de datos*

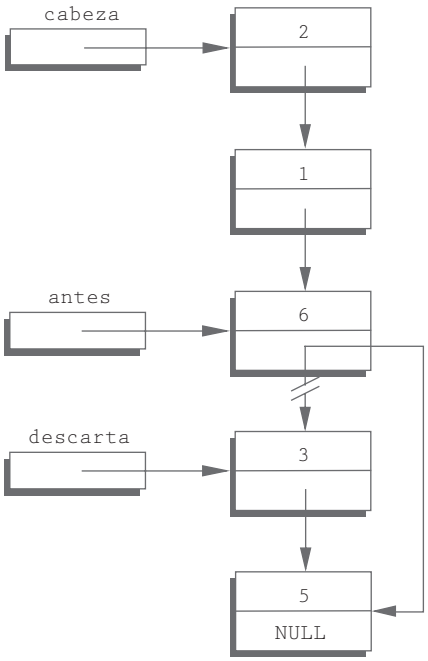
Si `cabeza1` y `cabeza2` son variables apuntadores y `cabeza1` apunta al nodo cabeza de una lista enlazada, la siguiente instrucción hará que `cabeza2` apunte al mismo nodo cabeza y por ende a la misma lista enlazada:

```
cabeza2 = cabeza1;
```

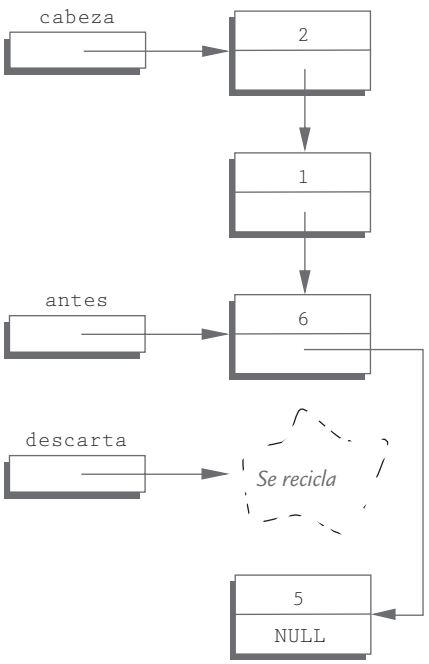
No obstante, debe recordar que sólo hay una lista enlazada y no dos. Si modifica la lista enlazada a la que apunta `cabeza1`, entonces también modificará la lista enlazada a la que apunta `cabeza2` ya que son la misma lista enlazada.

CUADRO 15.10 *Remoción de un nodo*

1. Posicione el apuntador `descarta` de manera que apunte al nodo que se va a eliminar y posicione el apuntador `antes` de manera que apunte al nodo que está antes del que se va a eliminar.
2. `antes->enlace = descarta->enlace;`



3. `delete descarta;`



Si `cabeza1` apunta a una lista enlazada y usted desea que `cabeza2` apunte a una segunda copia idéntica de esta lista, la instrucción de asignación anterior no funcionará. En vez de ello debe copiar toda la lista enlazada, nodo por nodo. Otra opción sería sobrecargar el operador de asignación = de manera que signifique lo que usted requiera. En la subsección "Sobrecarga del operador de asignación" del capítulo 12 hablamos acerca de la sobrecarga del operador =.

Ejercicios de AUTOEVALUACIÓN

5. Escriba las definiciones de tipos para los nodos y apuntadores en una lista enlazada. Utilice el nombre `TipoNodo` para el tipo nodo y `TipoApuntador` para el tipo apuntador. Las listas enlazadas serán listas de letras.

6. Por lo general, para obtener una lista enlazada se proporciona un apuntador que apunte al primer nodo en la lista, pero una lista vacía no tiene un primer nodo. ¿Qué valor de apuntador se utiliza por lo general para representar una lista vacía?

7. Suponga que su programa contiene las siguientes definiciones de tipos y declaraciones de variables apuntadores:

```
struct Nodo
{
    double datos;
    Nodo *siguiente;
};

typedef Nodo* Apuntador;
Apuntador p1, p2;
```

   Suponga que `p1` apunta a un nodo de este tipo que se encuentra en una lista enlazada. Escriba código que haga que `p1` apunte al siguiente nodo en esta lista enlazada. (El apuntador `p2` es para el siguiente ejercicio y no tiene nada que ver con este ejercicio.)

8. Suponga que su programa contiene definiciones de tipos y declaraciones de variables apuntadores, como en el ejercicio de autoevaluación 7. Suponga además que `p2` apunta a un nodo de tipo `Nodo`, que se encuentra en una lista enlazada y no es el último nodo en la lista. Escriba código que elimine el nodo que se encuentre *después* del nodo al que apunta `p2`. Después de ejecutar este código, la lista enlazada deberá ser la misma, sólo que habrá un nodo menos en ella. *Sugerencia:* Sería conveniente que declarara otra variable apuntador para usarla.

9. Seleccione una respuesta y explíquela.

   Para un arreglo extenso y una lista extensa que contiene objetos del mismo tipo, la inserción de un nuevo objeto en una ubicación conocida en medio de una lista enlazada, en comparación con la inserción en un arreglo, es

   a) Más eficiente.

   b) Menos eficiente.

   c) Casi lo mismo.

   d) Dependiente del tamaño de las dos listas.

Variaciones en listas enlazadas

En esta subsección le daremos una sugerencia acerca de las diversas estructuras de datos que pueden crearse mediante el uso de nodos y apuntadores. Describiremos brevemente dos estructuras de datos adicionales: la lista doblemente enlazada y el árbol binario.

Una lista enlazada ordinaria nos permite desplazarnos por ella en sólo una dirección (siguiendo los enlaces). Un nodo en una **lista doblemente enlazada** tiene dos enlaces, uno que apunta al siguiente nodo y otro que apunta al nodo anterior. En el cuadro 15.11 se muestra el diagrama de una lista doblemente enlazada.

lista doblemente enlazada

La clase de nodo para una lista doblemente enlazada podría ser la siguiente:

```
struct Nodo
{
    int datos;
    Nodo *enlace_delantero;
    Nodo *enlace_posterior;
};
```

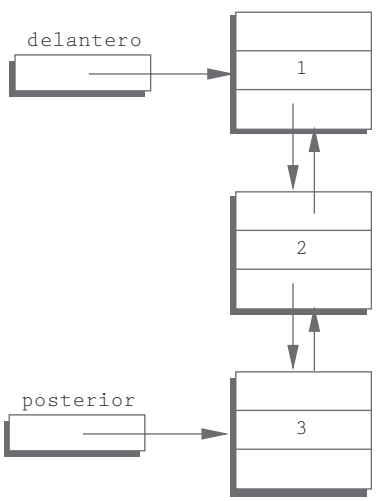
En vez de un solo apuntador hacia el nodo cabeza, una lista doblemente enlazada tiene un apuntador hacia cada uno de los dos nodos de los extremos. Podemos llamar a estos apuntadores `delantero` y `posterior`, aunque la elección de cuál es `delantero` y cuál `posterior` es arbitraria.

Las definiciones de los constructores y de algunas de las funciones en la clase de lista doblemente enlazada tendrán que modificarse (del caso de la lista con un solo enlace) para dar cabida al enlace adicional.

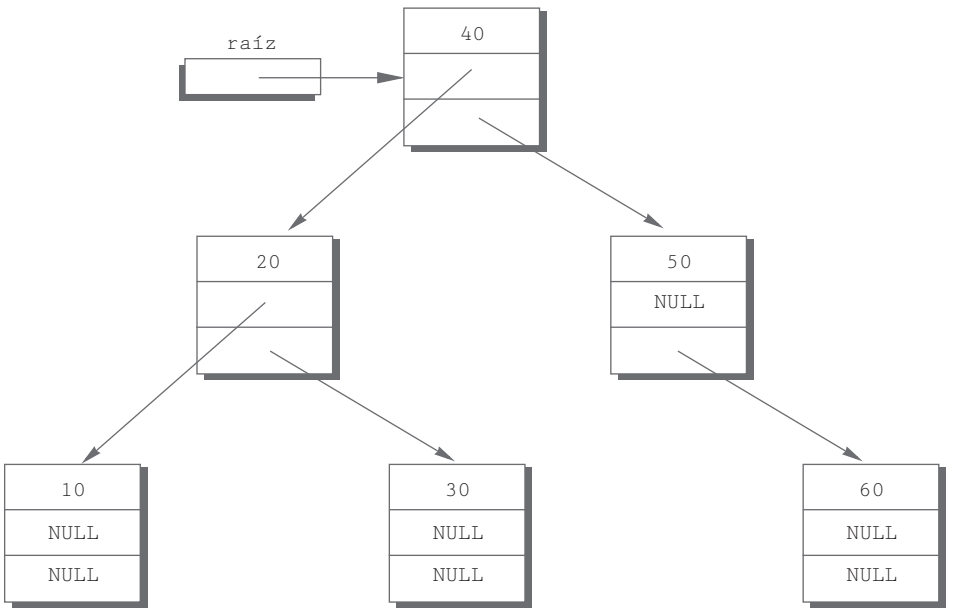
Un **árbol** es una estructura de datos que se estructura como se muestra en el cuadro 15.12. En especial, en un árbol podemos llegar a cualquier nodo desde el nodo superior (raíz) mediante cierta ruta que sigue los enlaces. Observe que no hay ciclos en un árbol. Si

árbol

CUADRO 15.11 *Una lista doblemente enlazada*



CUADRO 15.12 *Un árbol binario*



árbol binario

sigue los enlaces, en cierto momento llegará a un "final". Observe que cada nodo tiene dos enlaces que apuntan a otros nodos (o al valor `NULL`). Este tipo de árbol se llama **árbol binario**, ya que cada nodo tiene exactamente dos enlaces. Hay otros tipos de árboles con distintos números de enlaces en los nodos, pero el árbol binario es el más común.

Un árbol no es una forma de lista enlazada, pero utiliza enlaces (apuntadores) en formas similares a las listas enlazadas. La definición del tipo de nodo para un árbol binario es en esencia la misma que para una lista doblemente enlazada, pero por lo general los dos enlaces se nombran mediante el uso de alguna forma de las palabras *izquierda* y *derecha*. A continuación se muestra un tipo de nodo que puede usarse para construir un árbol binario:

```
struct NodoArbol
{
    int datos;
    NodoArbol *enlace_izquierdo;
    NodoArbol *enlace_derecho;
};
```

nodo raíz

En el cuadro 15.12, el apuntador llamado `raiz` apunta al **nodo raíz** ("nodo superior"). El nodo raíz tiene un propósito similar al del nodo cabeza en una lista enlazada ordinaria (cuadro 15.10). Se puede llegar a cualquier nodo en el árbol desde el nodo raíz si se siguen los enlaces.

El término árbol podría parecer equivocado. La raíz se encuentra en la parte superior del árbol y la estructura de ramificaciones parece más una estructura de ramificación de una raíz que una estructura de ramificación de un árbol. El secreto de la terminología es voltear la imagen (cuadro 15.12) hacia abajo. De esta forma la imagen se asemeja a la estructura de ramificación de un árbol, y el nodo raíz está en donde empezaría la raíz del árbol. Los nodos en los extremos de las ramas que tienen `NULL` en ambas variables de instancia de enlace se conocen como **nodos hoja**, una terminología que ahora sí puede tener sentido.

nodos hoja

Aunque no tenemos espacio en este libro para profundizar sobre el tema, los árboles binarios pueden usarse para almacenar y recuperar datos en forma eficiente.

15.2 Pilas y colas

Pero muchos que son primeros ahora serán los últimos,
y muchos que son los últimos ahora serán los primeros.

Mateo 19:30

Las listas enlazadas tienen muchas aplicaciones. En esta sección proporcionaremos dos muestras de su uso. Utilizaremos las listas enlazadas para proporcionar implementaciones de dos estructuras de datos conocidas como *pila* y *cola*. En esta sección siempre utilizaremos listas enlazadas regulares y no listas doblemente enlazadas.

Pilas

Una *pila* es una estructura de datos que recupera datos en orden inverso al que están almacenados. Suponga que coloca las letras `'A'`, `'B'` y después `'C'` en una pila. Cuando saque estas letras de la pila, se extraerán en el orden `'C'`, `'B'` y después `'A'`. En el cuadro 15.13 se muestra el diagrama de este uso de una pila. Como se muestra ahí, podemos considerar a una pila como un hoyo en la tierra. Para poder sacar algo de la pila, primero debe extraer los elementos que están encima del que usted desea. Por esta razón a la pila se le conoce como estructura de datos *último en entrar, primero en salir* (UEPS).

último en entrar, primero en salir

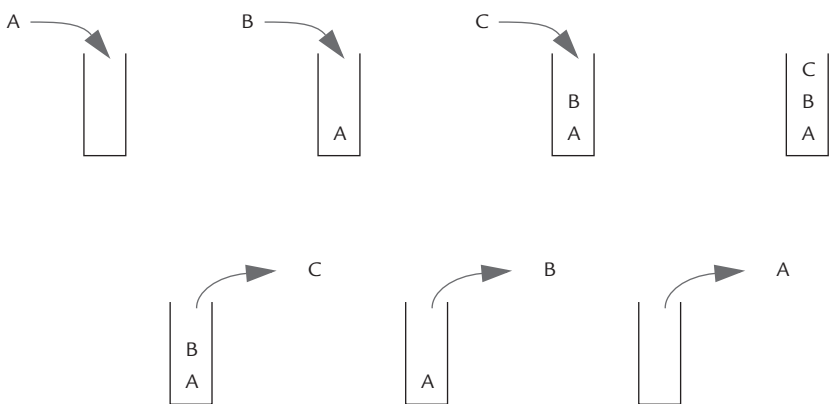
Las pilas se utilizan para muchas tareas de procesamiento de lenguajes. En el capítulo 13 vimos cómo el sistema computacional utiliza una pila para llevar el registro de las llamadas a funciones de C++. Sin embargo, aquí sólo realizaremos una aplicación muy simple. Nuestro objetivo en este ejemplo es mostrarle cómo puede usar las técnicas de la lista enlazada para implementar estructuras de datos específicas; una pila es un ejemplo sencillo del uso de listas enlazadas. No necesita leer el capítulo 13 para comprender este ejemplo.

EJEMPLO DE PROGRAMACIÓN
Una clase tipo pila

La interfaz para nuestra clase tipo pila se muestra en el cuadro 15.14. Esta pila se utiliza para almacenar datos de tipo `char`. Puede definir una pila similar para almacenar datos de cualquier otro tipo. Existen dos operaciones básicas que se pueden realizar en una pila: agregar un elemento y extraer un elemento. A la operación de agregar un elemento se le conoce como *meter* el elemento en la pila, por lo cual a la función miembro que realiza esta operación le llamamos `mete`. A la operación de extraer un elemento se le conoce como *sacar* el elemento de la pila, por lo cual a la función miembro que realiza esta operación le llamamos `saca`.

interfaz

CUADRO 15.13 *Una pila*



`mete` **y** `saca`

Los nombres `mete` y `saca` se derivan de otra forma de visualizar una pila. Una pila es análoga a un mecanismo que se utiliza algunas veces para almacenar los platos en una cafetería. El mecanismo almacena los platos en un hoyo en la encimera. Hay un resorte debajo de los platos, cuya tensión está ajustada de manera que sólo el plato superior sobresalga de la encimera. Si este tipo de mecanismo se utilizara como una estructura de datos tipo pila, los datos se escribirían en los platos (lo cual podría violar algunas leyes de sanidad, pero de todas formas es una buena analogía). Para agregar un plato a la pila sólo hay que colocarlo encima de los otros platos y el peso de este nuevo plato *mete* el resorte hacia abajo. Cuando se quita un plato, el plato debajo de éste *sale* a la vista.

programa de aplicación

El cuadro 15.15 muestra un programa simple que ilustra el uso de la clase tipo pila. Este programa lee una palabra letra por letra y coloca las letras en una pila. Después el programa extrae las letras una a una y las escribe en la pantalla. Como los datos se extraen de la pila en el orden inverso en el que entran, la salida muestra la palabra escrita al revés.

implementación

Como se muestra en el cuadro 15.16, nuestra clase tipo pila se implementa como una lista enlazada, en la que la cabeza de la lista sirve como la parte superior de la pila. La variable miembro `superior` es un apuntador que apunta hacia la cabeza de la lista enlazada.

La escritura de la definición de la función miembro `mete` corresponde al ejercicio de autoevaluación 10. No obstante, ya hemos proporcionado el algoritmo para esta tarea. El código de la función miembro `mete` es en esencia el mismo que la función `insercion_cabeza` que se muestra en el cuadro 15.4, sólo que en la función miembro `mete` utilizamos un apuntador llamado `superior` en vez de uno llamado `cabeza`.

constructor predeterminado

Una pila vacía es sólo una lista enlazada vacía, por lo que para implementarla se asigna `NULL` al apuntador `superior`. Una vez que sabemos que `NULL` representa a la pila vacía, las implementaciones del constructor predeterminado y de la función miembro `vacia` son obvias.

CUADRO 15.14 *Archivo de interfaz para una clase de tipo pila*

```
//Este es el archivo de encabezado pila.h. Es la interfaz para la clase Pila,
//que es una clase para una pila de símbolos.
#ifndef PILA_H
#define PILA_H
namespace pilasavitch
{
    struct PilaEstructura
    {
        char datos;
        PilaEstructura *enlace;
    };

    typedef PilaEstructura* PilaEstructuraPtr;

    class Pila
    {
    public:
        Pila( );
        //Inicializa el objeto como una pila vacía.

        Pila(const Pila& una_pila);
        //Constructor de copia.

        ~Pila( );
        //Destruye la pila y devuelve toda la memoria al almacén de memoria libre.

        void mete(char el_simbolo);
        //Postcondición: se ha agregado el_simbolo a la pila.

        char saca( );
        //Precondición: que la pila no esté vacía.
        //Devuelve el símbolo superior en la pila y extrae ese
        //símbolo superior de la pila.

        bool vacia( ) const;
        //Devuelve true si la pila está vacía. Devuelve false en caso contrario.
    private:
        PilaEstructuraPtr superior;
    };
}//pilasavitch

#endif //PILA_H
```

CUADRO 15.15 *Programa que utiliza la clase Pila (parte 1 de 2)*

```
//Programa para demostrar el uso de la clase Pila.
#include <iostream>
#include "pila.h"
using namespace std;
using namespace pilasavitch;

int main( )
{
    Pila s;
    char siguiente, resp;

    do
    {
        cout << "Escriba una palabra: ";
        cin.get(siguiente);
        while (siguiente != '\n')
        {
            s.mete(siguiente);
            cin.get(siguiente);
        }

        cout << "Escrita al reves es: ";
        while (! s.vacia())
            cout << s.saca();
        cout << endl;

        cout << "De nuevo? (s/n): ";
        cin >> resp;
        cin.ignore(10000, '\n');
    } while (resp != 'n' && resp != 'N');

    return 0;
}
```

El miembro `ignore` *de* `cin` *se describe en el capítulo 11. Descarta la entrada restante en la línea de entrada actual hasta 10,000 caracteres o hasta que se oprima Intro. También descarta el retorno* (`'\n'`) *al final de la línea.*

CUADRO 15.5 *Programa que utiliza la clase Pila (parte 2 de 2)*

Diálogo de ejemplo

```
Escriba una palabra: popote
Escrita al reves es: etopop
De nuevo? (s/n): s
Escriba una palabra: C++
Escrita al reves es: ++C
De nuevo? (s/n): n
```

CUADRO 15.16 *Implementación para la clase Pila (parte 1 de 2)*

```
//Este es el archivo de implementación pila.cpp.
//Es la implementación de la clase Pila.
//La interfaz para la clase Pila está en el archivo de
//encabezado pila.h.
#include <iostream>
#include <cstddef>
#include "pila.h"
using namespace std;

namespace pilasavitch
{
    //Usa cstddef:
    Pila::Pila( ) : superior(NULL)
    {
        //Cuerpo vacío de manera intencional.
    }

    //Usa cstddef:
    Pila::Pila(const Pila& una_pila)
```

<La definición del constructor de copia es el ejercicio de autoevaluación 11.>

CUADRO 15.16 *Implementación para la clase Pila (parte 2 de 2)*

```
Pila::~Pila( )
{
     char siguiente;
     while (! vacia( ))
        siguiente = saca( );//saca llama a delete.
}

//Usa cstddef:
bool Pila::vacia( ) const
{
     return (superior == NULL);
}

//Usa cstddef:
void Pila::mete(char el_simbolo)
{
     PilaEstructuraPtr temp_ptr;
     temp_ptr = new PilaEstructura;

     temp_ptr->datos = el_simbolo;

     temp_ptr->enlace = superior;
     superior = temp_ptr;
}

//Usa iostream:
char Pila::saca( )
{
     if (vacia( ))
     {
          cout << "Error: no se puede sacar de una pila vacia.";
          exit(1);
     }

     char resultado = superior->datos;

     PilaEstructuraPtr temp_ptr;
     temp_ptr = superior;
     superior = superior->enlace;

     delete temp_ptr;

     return resultado;
}
}//pilasavitch
```

La definición del constructor de copia es un poco complicada, pero no utiliza técnicas que no hayamos visto ya. Los detalles se dejan para el ejercicio de autoevaluación 11.

constructor de copia

La función miembro `saca` primero comprueba si la pila está vacía. Si no está vacía, procede a extraer el carácter superior en la pila. Hace que la variable local `resultado` sea igual al símbolo superior en la pila. Esto se hace de la siguiente manera:

saca

```
char resultado = superior->datos;
```

Una vez que se almacena el símbolo del nodo superior en la variable `resultado`, el apuntador `superior` se desplaza hacia el siguiente nodo en la lista enlazada, con lo cual se elimina en efecto el nodo superior de la lista. El apuntador `superior` se desplaza mediante la siguiente instrucción:

```
superior = superior->enlace;
```

No obstante, antes de que se desplace el apuntador `superior` se posiciona un apuntador temporal llamado `temp_ptr`, de manera que apunte hacia el nodo que está a punto de removerse de la lista. Ahora el nodo puede eliminarse mediante la siguiente llamada a *`delete`*:

```
delete temp_ptr;
```

Cada nodo que se elimina de la lista enlazada mediante la función miembro `saca` se destruye mediante una llamada a *`delete`*. Por ende, todo lo que el destructor necesita hacer es extraer cada elemento de la pila con una llamada a `saca`. Así, cada nodo devolverá su memoria al almacén de memoria libre.

destructor

Ejercicios de AUTOEVALUACIÓN

10. Proporcione la definición de la función miembro `mete` de la clase `Pila` que se describe en el cuadro 15.14.
11. Proporcione la definición del constructor de copia para la clase `Pila` que se describe en el cuadro 15.14.

Colas

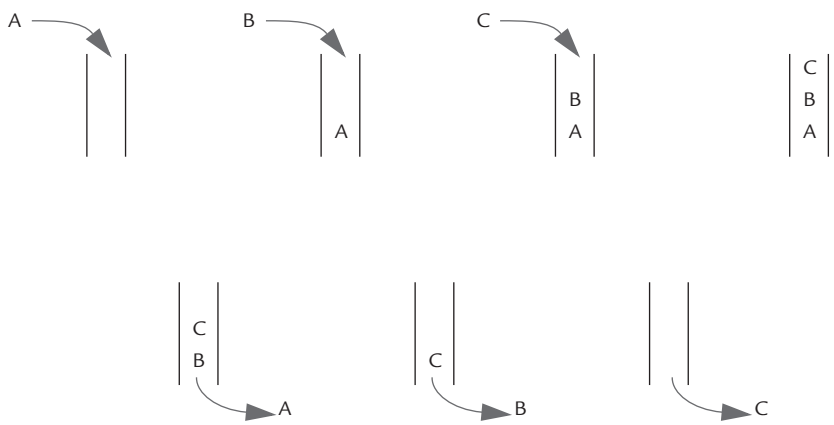
Una pila es una estructura de datos tipo "último en entrar/primero en salir". La **cola** es otra estructura de datos común, la cual maneja datos de la forma "primero en entrar/primero en salir" (PEPS). Una cola se comporta de igual forma que una línea de personas que esperan ser atendidas por el cajero de un banco o para cualquier otro servicio. Las personas se atienden en el orden en el que entran a la línea (la cola). En el cuadro 15.17 se muestra un diagrama de la operación de una cola. Una cola puede implementarse con una lista enlazada, de una forma similar a nuestra implementación de la clase `Pila`. No obstante, una cola necesita un apuntador tanto en la cabeza de la lista como en el otro extremo de la lista enlazada, ya que hay acción en ambas ubicaciones. Es más sencillo extraer un nodo de la cabeza de una lista enlazada que del otro extremo de la lista. Por lo tanto, nuestra implementación extraerá un nodo de la cabeza de la lista (que ahora llamaremos parte **delantera** de la lista) y agregaremos nodos en el otro extremo de la lista, al cual llamaremos parte **posterior** de la lista (o parte posterior de la cola).

cola
primero en entrar/primero en salir
delantera
posterior

Cola

Una `cola` es una estructura de datos del tipo "primero en entrar/primero en salir"; es decir, los elementos de datos se extraen de la cola de la misma forma en que se agregaron.

CUADRO 15.17 *Una cola*



EJEMPLO DE PROGRAMACIÓN
Una clase tipo cola

interfaz

La interfaz para nuestra clase tipo cola se proporciona en el cuadro 15.18. Esta cola específica se utiliza para almacenar datos de tipo `char`. Puede definir una cola similar para almacenar datos de cualquier otro tipo. Existen dos operaciones básicas que se pueden realizar en una cola: agregar un elemento al final de la cola y extraer un elemento de su parte frontal.

programa de aplicación

El cuadro 15.19 muestra un programa simple que ilustra el uso de la clase tipo cola. Este programa lee una palabra letra por letra y coloca las letras en una cola. Después el programa extrae las letras una por una y las escribe en la pantalla. Como los datos se extraen de una cola en el orden en el que entran, la salida muestra las letras de la palabra en el mismo orden en el que el usuario las introdujo. Es bueno comparar esta aplicación de una cola con una aplicación similar que utiliza una pila, como la que vimos en el cuadro 15.15.

implementación

Como se muestra en los cuadros 15.18 y 15.20, nuestra clase tipo cola se implementa como una lista enlazada en la que la cabeza de la lista sirve como la parte delantera de la cola. La variable miembro `delantera` es un apuntador que apunta hacia la cabeza de la lista enlazada. Los nodos se extraen en la cabeza de la lista enlazada. La variable miembro `posterior` es un apuntador que apunta hacia el nodo que está en el otro extremo de la lista enlazada. Los nodos se agregan en este extremo de la lista enlazada.

Una cola vacía es sólo una lista enlazada vacía, por lo que para implementar una cola vacía se asigna `NULL` a los apuntadores `delantera` y `posterior`.

El resto de los detalles de la implementación son similares a lo que hemos visto antes.

CUADRO 15.18 *Archivo de interfaz para una clase tipo cola*

```
//Este es el archivo de encabezado cola.h. Es la interfaz para la clase Cola,
//la cual es una clase para una cola de símbolos.
#ifndef COLA_H
#define COLA_H
namespace colasavitch
{
    struct NodoCola
    {
        char datos;
        NodoCola *enlace;
    };

    typedef NodoCola* NodoColaPtr;

    class Cola
    {
    public:
        Cola();
        //Inicializa el objeto con una cola vacía.

        Cola(const Cola& unaCola);

        ~Cola();

        void agrega(char elemento);
        //Postcondición: se ha agregado un elemento a la parte posterior de la cola.

        char extrae();
        //Precondición: que la cola no esté vacía
        //Devuelve el elemento que está en la parte delantera de la cola y extrae
        //ese elemento de la cola.

        bool vacia() const;
        //Devuelve true si la cola está vacía. Devuelve false en caso contrario.
    private:
        NodoColaPtr delantera;//Apunta a la cabeza de una lista enlazada.
                              //Los elementos se extraen de la cabeza.
        NodoColaPtr posterior;//Apunta hacia el nodo que está en el otro extremo
                              //de la lista enlazada. Los elementos se agregan en
                              //este extremo.
    };

} //colasavitch
#endif //COLA_H
```

CUADRO 15.19 *Programa que utiliza la clase Cola (parte 1 de 2)*

```
//Programa para demostrar el uso de la clase Cola.
#include <iostream>
#include "cola.h"
using namespace std;
using namespace colasavitch;

int main()
{
    Cola q;
    char siguiente, resp;

    do
    {
        cout << "Escriba una palabra: ";
        cin.get(siguiente);
        while (siguiente != '\n')
        {
            q.agrega(siguiente);
            cin.get(siguiente);
        }

        cout << "Usted escribio:: ";
        while ( ! q.vacia() )
            cout << q.extrae();
        cout << endl;

        cout << "De nuevo? (s/n): ";
        cin >> resp;
        cin.ignore(10000, '\n');
    }while (resp != 'n' && resp != 'N');

    return 0;
}
```

En el capítulo 11 hablamos acerca del miembro `ignore` *de* `cin`*, el cual descarta la entrada restante en la línea de entrada actual hasta 10,000 caracteres, o hasta que se oprima Intro. También descarta el retorno (*`'\n'`*) al final de la línea.*

CUADRO 15.19 *Programa que utiliza la clase Cola (parte 2 de 2)*

Diálogo de ejemplo

```
Escriba una palabra: popote
Usted escribio:: popote
De nuevo? (s/n): s
Escriba una palabra: C++
Usted escribio:: C++
De nuevo? (s/n): n
```

CUADRO 15.20 *Implementación de la clase Cola (parte 1 de 2)*

```
//Este es el archivo de implementación cola.cpp
//Es la implementación de la clase Cola.
//La interfaz para la clase Cola está en el archivo de encabezado cola.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "cola.h"
using namespace std;

namespace colasavitch
{
    //Usa cstddef:
    Cola::Cola() : delantera(NULL), posterior(NULL)
    {
        //Vacío de manera intencional.
    }

    Cola::Cola(const Cola& unaCola)
        <La definición de este constructor de copia es el ejercicio de autoevaluación 12.>

    Cola::~Cola()
        <La definición del destructor es el ejercicio de autoevaluación 11.>

    //Usa cstddef:
    bool Cola::vacia() const
    {
        return (posterior == NULL);//delantera == NULL también funciona
    }
```

CUADRO 15.20 *Implementación de la clase Cola (parte 2 de 2)*

```
    //Usa cstddef:
    void Cola::agrega(char elemento)
    {
        if (vacia())
        {
            delantera = new NodoCola;
            delantera->datos = elemento;
            delantera->enlace = NULL;
            posterior = delantera;
        }

        else
        {
            NodoColaPtr temp_ptr;
            temp_ptr = new NodoCola;
            temp_ptr->datos = elemento;
            temp_ptr->enlace = NULL;
            posterior->enlace = temp_ptr;
            posterior = temp_ptr;
        }
    }

    //Usa cstdlib e iostream:
    char Cola::extrae()
    {
        if (vacia())
        {
            cout << "Error: no se puede extraer un elemento de una cola vacia.\n";
            exit(1);
        }

        char resultado = delantera->datos;

        NodoColaPtr descarta;
        descarta = delantera;
        delantera = delantera->enlace;
        if (delantera == NULL) //si se extrajo el último nodo
            posterior = NULL;

        delete descarta;

        return resultado;
    }
}
```

Ejercicios de AUTOEVALUACIÓN

12. Proporcione la definición del constructor de copia para la clase `Cola` que se describe en el cuadro 15.18.
13. Proporcione la definición del destructor para la clase `Cola` que se describe en el cuadro 15.18.

Resumen del capítulo

- Un nodo es un objeto *`struct`* o un objeto de clase que tiene una o más variables miembro que son variables apuntador. Estos nodos pueden conectarse mediante sus variables apuntador miembro para producir estructuras de datos que pueden aumentar y reducir su tamaño mientras un programa se ejecuta.
- Una lista enlazada es una lista de nodos en la que cada nodo contiene un apuntador hacia el siguiente nodo en la lista.
- Para indicar el final de una lista enlazada (o de cualquier otra estructura de datos enlazada) se asigna `NULL` a la variable miembro apuntador.
- Una pila es una estructura de datos del tipo "primero en entrar/último en salir". Una pila puede implementarse mediante una lista enlazada.
- Una cola es una estructura de datos del tipo "primero en entrar/primero en salir". Una cola puede implementarse mediante una lista enlazada.

Respuestas a los ejercicios de autoevaluación

1.
```
Sally
Sally
18
18
```

Observe que `(*cabeza).nombre` y `cabeza->nombre` significan lo mismo. De manera similar, `(*cabeza).numero` y `cabeza->numero` significan lo mismo.

2. La mejor respuesta es

```
cabeza->siguiente = NULL;
```

No obstante, lo siguiente es también correcto:

```
(*cabeza).siguiente = NULL;
```

3.
```
delete cabeza;
```

4.
```
cabeza->elemento = "Orville el hermano de Wilbur";
```

5.
```
struct TipoNodo
{
    char datos;
```

```
    TipoNodo *enlace;
};
typedef TipoNodo* TipoApuntador;
```

6. El valor de apuntador `NULL` se utiliza para indicar una lista vacía.

7. `p1 = p1->siguiente;`

8.
```
Apuntador descarta;
descarta = p2->siguiente;
//ahora descarta apunta al nodo que se va a eliminar.
p2->siguiente = descarta->siguiente;
```

   Esto es suficiente para eliminar el nodo de la lista enlazada. No obstante, si no planea utilizar este nodo para algo más, debe destruirlo con una llamada a *delete*, como se muestra a continuación:

```
delete descarta;
```

9. a) Insertar un nuevo elemento en una ubicación conocida de una lista enlazada extensa es más eficiente que insertarlo en un arreglo extenso. Si va a insertar un elemento en una lista necesita unas cinco operaciones, de las cuales la mayoría son asignaciones a apuntadores, sin importar el tamaño de la lista. Si va a insertar un elemento en un arreglo, en promedio tendría que desplazar casi la mitad de las entradas en el arreglo para insertar un elemento de datos.

   Para listas pequeñas, la respuesta es c), casi lo mismo.

10.
```
//Usa cstddef;
void Pila::mete(char el_simbolo)
{
     PilaEstructuraPtr temp_ptr;
     temp_ptr = new PilaEstructura;

     temp_ptr->datos = el_simbolo;

     temp_ptr->enlace = superior;
     superior = temp_ptr;
}
```

11.
```
//Usa cstddef:
Pila::Pila(const Pila& una_pila)
{
    if (una_pila.superior == NULL)
        superior = NULL;
    else
    {
       PilaEstructuraPtr temp = una_pila.superior; //temp se desplaza
           //a través de los nodos, desde superior hasta el fondo de
           //una_pila.
       PilaEstructuraPtr fin;//Apunta al fin de la nueva pila.

       fin = new PilaEstructura;
       fin->datos = temp->datos;
       superior = fin;
       //El primer nodo se crea y se llena con datos.
       //Ahora se agregan nuevos nodos DESPUÉS de este primer nodo.

       temp = temp->enlace;
```

```
            while (temp != NULL)
            {
                fin->enlace = new PilaEstructura;
                fin = fin->enlace;
                fin->datos = temp->datos;
                temp = temp->enlace;
            }
            fin->enlace = NULL;
        }
}
```

12.
```
//Usa cstddef;
Cola::Cola(const Cola& unaCola)
{

if (unaCola.vacia())
    delantera = posterior = NULL;
else
{
       NodoColaPtr temp_ptr_antiguo = unaCola.delantera;
       //temp_ptr_antiguo se desplaza a través de los nodos
       //desde la parte delantera a la posterior de unaCola.
       NodoColaPtr temp_ptr_nuevo;
       //temp_ptr_nuevo se utiliza para crear nuevos nodos.

       posterior = new NodoCola;
       posterior->datos = temp_ptr_antiguo->datos;
       posterior->enlace = NULL;
       delantera = posterior;
       //se crea el primero nodo y se llena de datos.
       //Los nuevos nodos se agregan ahora, DESPUÉS de este primer nodo.

       temp_ptr_antiguo = temp_ptr_antiguo->enlace;
       //temp_ptr_antiguo ahora apunta al segundo
       //nodo o a NULL si no hay un segundo nodo.

       while (temp_ptr_antiguo != NULL)
       {
            temp_ptr_nuevo = new NodoCola;
            temp_ptr_nuevo->datos = temp_ptr_antiguo->datos;
            temp_ptr_nuevo->enlace = NULL;
            posterior->enlace = temp_ptr_nuevo;
            posterior = temp_ptr_nuevo;
            temp_ptr_antiguo = temp_ptr_antiguo->enlace;
       }
   }
}
```

13.
```
Cola::~Cola()
{
    char siguiente;
    while (! vacia())
        siguiente = extrae();//extrae llama a delete.
}
```

Proyectos de programación

1. Escriba una función `void` que tome una lista enlazada de enteros e invierta el orden de sus nodos. La función tendrá un parámetro de llamada por referencia, el cual será un apuntador a la cabeza de la lista. Después de llamar a la función, este apuntador apuntará a la cabeza de una lista enlazada que tenga los mismos nodos que la lista original, pero en el orden inverso al que tenían en la lista original. Tenga en cuenta que su función no creará ni destruirá nodos. Sólo los reacomodará. Coloque su función en un programa de prueba adecuado.

2. Escriba una función llamada `mezclar_listas` que tome dos argumentos de llamada por referencia que sean variables apuntador, que apunten a las cabezas de listas enlazadas de valores con el tipo `int`. Se supone que las dos listas enlazadas están ordenadas de manera que el número en la cabeza sea el más pequeño, el número en el siguiente nodo sea el siguiente más pequeño, y así sucesivamente. La función devolverá un apuntador a la cabeza de una nueva lista enlazada que contendrá todos los nodos de las dos listas originales. Los nodos en esta lista más grande también se ordenarán de menor a mayor. Tenga en cuenta que su función no debe crear ni destruir nodos. Cuando termine la llamada a la función, los dos argumentos tipo variable apuntador deberán tener el valor `NULL`.

3. Diseñe e implemente una clase cuyos objetos representen polinomios. El polinomio

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$

se implementará como una lista enlazada. Cada nodo debe contener un valor `int` para la potencia de `x` y un valor `int` para el coeficiente correspondiente. Las operaciones de esta clase deberán incluir la suma, resta, multiplicación y evaluación de un polinomio. Debe sobrecargar los operadores +, − y * para la suma, resta y multiplicación.

La evaluación de un polinomio se implementa como una función miembro con un argumento de tipo `int`. La función miembro de evaluación debe devolver el valor obtenido al utilizar su argumento para `x` y realizar las operaciones indicadas. Incluya cuatro constructores: un constructor predeterminado, un constructor de copia, un constructor con un solo argumento de tipo `int` que produzca el polinomio que tiene sólo un término constante, igual al argumento del constructor, y un constructor con dos argumentos de tipo `int` que produzca el polinomio de un término cuyo coeficiente y exponente se proporcionen mediante los dos argumentos. (En la notación anterior, el polinomio producido por el constructor de un argumento es de la forma simple que consiste sólo de a_0. El polinomio producido por el constructor de dos argumentos es de la forma $a_n x^n$, que es un poco más complicada.) Incluya un descriptor adecuado. Incluya también funciones miembro para introducir y mostrar polinomios.

Cuando el usuario introduzca un polinomio, deberá escribir lo siguiente:

$a_n x \hat{} n + a_{n-1} xv \hat{} n - 1 + \ldots + a_0$

No obstante, si un coeficiente a_i es cero, el usuario podría omitir el término $a_i x \hat{} i$. Por ejemplo, el polinomio

$3x^4 + 7x^2 + 5$

puede introducirse como

3x^4 + 7x^2 + 5

También podría introducirse como

3x^4 + 0x^3 + 7x^2 + 0x^1 + 5

Si un coeficiente es negativo, se utiliza un signo negativo en lugar de uno positivo, como en los siguientes ejemplos:

3x^5 − 7x^3 + 2x^1 − 8
−7x^4 + 5x^2 + 9

Un signo negativo en frente del polinomio, como en el segundo de los dos ejemplos, se aplica sólo al primer coeficiente; no hace que todo el polinomio sea negativo. Los polinomios se mostrarán en el mismo formato. En el caso de la salida, no se mostrarán los términos con cero coeficientes.

Para simplificar la entrada, puede suponer que los polinomios siempre se escribirán uno en cada línea y que siempre habrá un término constante a_0. Si no hay un término constante, el usuario introducirá cero para el término constante, como en el siguiente ejemplo:

12x^8 + 3x^2 + 0

4. En este proyecto volverá a realizar el Proyecto de programación 10 del capítulo 10, mediante el uso de una lista enlazada en vez de un arreglo. Como se indica ahí, ésta es una lista enlazada de elementos `double`. Este hecho puede implicar modificaciones en algunas de las funciones miembro. Los miembros son: un constructor predeterminado; una función miembro llamada `agrega_elemento` para agregar un `double` a la lista; una prueba para una lista completa, que es una función de valor Booleano llamada `completa()`; y una función `friend` que sobrecarga el operador de inserción `<<`.

5. Una versión más difícil del Proyecto de programación 4 sería escribir una clase llamada `Lista`, similar al Proyecto 4, pero con todas las siguientes funciones miembro:

   - Constructor predeterminado `Lista();`.
   - `double Lista::delantero();`, que devuelve el primer elemento en la lista.
   - `double Lista::posterior();`, que devuelve el último elemento en la lista.
   - `double Lista::actual();`, que devuelve el elemento "actual".
   - `void Lista::avanza();`, que avanza el elemento que devuelve `actual()`.
   - `void Lista::reinicia();`, para hacer que `actual()` devuelva el primer elemento en la lista.
   - `void Lista::inserta(double despues_demi, double insertame);`, que inserta a `insertame` en la lista después de `despues_demi` e incrementa la variable `private` llamada `cuenta`.
   - `int tamanio();`, que devuelve el número de elementos en la lista.
   - `friend istream& operator<< (istream& ins, double escribeme);`.

   Los miembros de datos privados deberán incluir:

```
nodo *cabeza;
nodo* actual;
int  cuenta;
```

   y tal vez un apuntador más.

   Necesitará la siguiente `struct` (fuera de la clase tipo lista) para los nodos de la lista enlazada:

```
struct nodo
{
     double elemento;
     nodo *siguiente;
};
```

El desarrollo por incrementos es imprescindible para todos los proyectos de cualquier tamaño, sin que éste sea una excepción. Escriba la definición para la clase `Lista`, pero no implemente sus miembros todavía. Coloque la definición de esta clase en un archivo llamado `"lista.h"`. Después incluya `"lista.h"` mediante la directiva `#include` en un archivo que contenga `int main(){}`. Compile su archivo. Encontrará errores de

sintaxis y muchos errores de tipografía que podrían provocar muchas dificultades si usted tratara de implementar los miembros sin esta comprobación. Por ende, es conveniente que implemente y compile un miembro a la vez hasta que tenga lo suficiente como para escribir código de prueba en su función principal (main).

6. Ésta es una versión más difícil del Proyecto de programación 5 que utiliza plantillas (las cuales se vieron en el capítulo 14). Escriba la definición de una clase de plantilla para una lista de objetos cuyo tipo sea un parámetro de tipo. Asegúrese de escribir una implementación completa de todas las funciones miembro y los operadores sobrecargados. Pruebe su lista de plantilla con un tipo *struct*, como se muestra a continuación:

```
struct Persona
{
    string nombre;
    string numero_telefonico;
};
```